Extending Timestamp-Based Two Phase Commit Protocol for RESTful Services to Meet Business Rules

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

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

ABSTRACT

Service Oriented Architecture allows development of software with requirements of interoperability and weak coupling. Nowadays, REST is an architectural style that has been gaining attention in the SOA domain. REST allows the development of web services based on concepts simpler than WS-*, however, REST, as an architectural style, does not provide "official" standards to address some non-functional requirements of services, such as, security, reliability, and transaction control. The Timestamp-based Two Phase Commit Protocol for RESTful Services (TS2PC4RS) algorithm proposes a REST-based technique to support the web services transactional control implementation. This paper proposes to extend the TS2PC4RS algorithm to improve the satisfaction of business rules. The goal is met in the way the clients can update their prewrites on the ongoing transactions, so that the clients do not need to start a new transaction in order to implement the desired updates. The update of prewrites takes into account the application domain business rules which guide the RESTful services behavior. Thus the business rules are also considered in the algorithm extension. An example was used to describe the TS2PC4RS extension for updates.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

DESIGN, EXPERIMENTATION

Keywords

Architectural style, concurrency control algorithm, REST, transaction, timestamp, web services, business rules

1. INTRODUCTION

REST [5] (Representational State Transfer) is an emerging technology that has been gaining attention in the SOA (Service Oriented Architecture) domain due to the fact it has its foundation on the original design principles of the World Wide Web. The usage of these original principles eases the consumption of the services by the clients what allows the services providers attract a larger user community [16, 19].

In the REST architectural style the resource is the key abstract information. Every resource is associated with a an URI[20] (Universal Resource Identifiers), which is the resource name and the address. The resources are manipulated through a uniform interface which maps to HTTP methods [18]. GET retrieves information about a resource. PUT creates a new resource when new URI is provided by the client. PUT also updates an existing resource. POST creates a new resource without providing new URI. DELETE erases an existing resource.

In general, RESTful services are well suited for basic, tactical, ad hoc integration over the Web [17]. On the other hand, WS-* Web services, which are based on SOAP [21] and WSDL [22], are preferred for professional enterprise application integration scenarios with a longer lifespan and advanced QoS requirements (e.g., transactions, security, reliability) [17].

In spite of such suggestions of use, both approaches have advantages and disadvantages so that it is up to the developer to make the decision of which approach is more suited for each particular case.

WS-* web services are supported by a set of specifications for the development of services based on SOAP and WSDL, which can be used when advanced QoS requirements are necessary. The specifications include *WS-Security* [11], *WS-Reliability* [10], *WS-Transaction* [12], *WS-Coordination* [13].

RESTful web services are considered simpler to adopt and understand than the WS-* web services, however, most of the non-functional (QoS) requirements in RESTful web services are not addressed by an "official" standard. The reason it that REST is an architecture style, and so, it is used mainly to understand and design web services guided by that style [4].

Nonetheless, while REST is not a standard, the concrete implementation of REST does use standards like, HTTP, URL and for resource representation: XML, HTML, GIF, JPEG and so on. Some non-functional requirements are also addressed by standards used in concrete REST architecture implementations. For instance, security can be reached by using OAuth (secure API authorization) [14], OpenId (single sign-on protocol) [15] or HTTP basic and digest access authentication [6], reliability can be reached using the HTTP features, particularly the idempotent verbs.

Despite the standards used in concrete REST architectures, there still is a lack of support for the QoS requirements in the REST domain. Thus, in order to provide the transaction control for the REST domain, Hiane and Hirata have proposed TS2PC4RS to address concurrency control with RESTful services [9]. The TS2PC4RS (*Timestamp-based Two Phase Commit Protocol for RESTful Services*) algorithm uses timestamp technique [8, 3] with the two phase commit (2PC) protocol to control concurrent access to REST resources.

In the TS2PC4RS algorithm the client is allowed to send read (using GET), prewrite (using PUT), write (using PUT) operations to the RESTful services involved in the transaction context. To execute a transaction, the client first sends the prewrite messages (prepare) to all participating RESTful services and based on the participating responses the client sends the write message (commit or abort) to complete the transaction.

Without the ability to update prewrites, if a client desires to change a prewrite already accepted by the RESTful service, the normal procedure is to execute two operations, first the client must abort the previous prewrite (the one the client wants to change) and then it must send a new prewrite containing the desired changes. However, before the client sends the new prewrite, some other client may succeed in sending its prewrites, which can make the new prewrite of the first client inconsistent due to the restrictions imposed by the business rules. The RESTful service can also reject the new prewrite due to the timestamp rules.

Therefore, it is necessary to improve the way the clients can update their prewrites, so that the prewrite update operation can be performed in the current transaction without starting a new one. In order to provide the ability to update the prewrites, it is important to consider how the application domain business rules are taken into account to extend the TS2PC4RS algorithm.

For the purpose of this work, business rules can be understood, from the information system perspective, as statements that define or constrain aspects of business. They are intended to assert business structure, or to control or influence the behavior of the business. Thus, a business rule expresses specific constraints on the creation, updating, and removal of persistent data in an information system [2].

In this paper we investigate how business rules can be met when using TS2PC4RS algorithm by addressing the updates of prewrites already accepted. More specifically, we aim to extend the TS2PC4RS algorithm [9] in order to provide the clients with the ability to update its prewrites taking into account the business rules.

Sections of this paper are organized as follows. Section 2 recalls some of the key aspects of the TS2PC4RS algorithm and describe the extensions necessary to support the prewrite updates. The TS2PC4RS extensions are described through the purchase of tickets example. Section 3 presents an analysis of the proposed TS2PC4RS extensions pointing out new possibilities of interactions between clients and RESTful services. Section 4 presents the conclusions and proposals for future work.

2. TIMESTAMP CONCURRENCY FOR REST EXTENDED

Clients are allowed to send *prewrite update* operations to the RESTful services, which, based on their business rules, accomplish the prewrite update immediately or, if it cannot be performed, the prewrite update can be stored to be accomplished in the future. The interactions between clients and servers are better discussed in Section 2.2.

Section 2.1 recalls some of the key aspects of the original algorithm. Section 2.2 exposes the issues that emerge from the capability to update prewrites already accepted by the RESTful services, considering the application domain business rules. Section 2.3 presents the main necessary extensions to the algorithm to support prewrite updates.

2.1 TS2PC4RS Original Algorithm

In this subsection, we reproduce the original TS2PC4RS algorithm for concurrency control in RESTful services domain presented in [9].

```
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;
```

In this article, we are dealing mainly with step 4 of this algorithm, i.e., step 4 is extended to accept updates and to take into account the application domain business rules.

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 removed from the LPW.

The commit procedure [9] that must be accomplished when the agents receive the write operation W indicating the transaction commitment follows.

```
Search for 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;
```

First the prewrite being committed is found in the LPW and if it is the first one in LPW, then the prewrite is removed from LPW, the write operation is executed and WTM is updated with the timestamp of the removed prewrite.

Thereafter, if there is a sequence of prewrites (a LPW sublist) that are marked to be removed from LPW – with the update-data mark – immediately after the removed prewrite, then the sequence is removed from LPW, the corresponding write operations are executed and WTM is updated with the timestamp of the last prewrite of the sequence. However, if the prewrite being committed is the second forth, the prewrite is just marked to be removed as soon as possible.

If the transaction is aborted, the following procedure [9] must be performed by the agents that receive the write operation W indicating the transaction abortion.

```
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);
```

Primarily, the prewrite being aborted is removed from LPW and if it is the first one in LPW, we have to verify if there is a sequence of prewrites that are marked to be removed from LPW immediately after the removed prewrite. If the sequence is found, then it is removed from LPW, the corresponding write operations are executed and WTM is updated with the timestamp of the last prewrite of the sequence.

The client (2PC coordinator) is responsible to coordinate all the transaction. It starts the transaction by sending requests through prewrites to all RESTful services (2PC agents) which are involved in the business process being executed. If all services can perform the operations requested, they reply with a *ready* message to the coordinator client. Otherwise the services reply with a *not-ready* message.

So, in the all-or-nothing case, the client evaluates all the reply messages received from the RESTful services and if they are all ready, the client commits the transaction by sending the *commit* message to all services. If the client receives a not-ready message, it aborts the transaction sending *abort* messages to all services.

If the business rules allow, the client may wish to partially commit the transaction. In this case, even receiving some not-ready messages, the client may want to commit the requests which are ready by sending the commit message to the corresponding RESTful services.

The client may also want to change his/her request seeking to increase the chances of success with all services. With the extended TS2PC4RS proposed in this paper and the *updated view of the data item* concept proposed by the original TS2PC4RS algorithm, the client is allowed to make a decision and update the prewrite on the data item (resource) of interest using a transaction in progress.

In order to allow prewrite updates by clients, ensuring compliance with all involved business rules, the RESTful services should relax the control over the LPW. So, the TS-2PC4RS algorithm needs some extensions as described in next Sections.

2.2 Purchase of Tickets Example

We use the purchase of tickets example [9] to facilitate the understanding of the issues about supporting prewrite updates taking into account the involved business rules. The example considers two operations: purchase of tickets for a basketball game and purchase of train tickets to go to the city (place) of the game.

The client's objective is to buy a certain number of tickets for the game together with the train tickets to go to the place of the game. Initially, clients want to buy the same amount of tickets for the game and for the train seats. Table 1 provides an overview of the main resources, URIs, and operations for the service responsible for the game tickets. The train tickets service has similar resources.

It is a good practice to expose the transaction concept as a resource [1, 7, 18, 23]. In this way, we decide to expose transaction information in its own resource (`/booking/{TS}`) for each involved server.

As described in [9], each REST resource that uses the timestamp concurrency control has the following additional attributes within its representation: the largest write operation timestamp (WTM), the largest read operation timestamp (RTM), and the list of buffered prewrites (LPW). The RESTful services implement the two-phase-commit (2PC) agents that control the access to the data items. The REST clients implement the 2PC coordinator.

The client sends read operations (R) through GET messages; prewrite (PW) and write (W) operations through 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 409 status code (Conflict) and information for the client to try to fix the problem. For example, if R is rejected because of the WTM, the value of WTM must appear in the message body in order to permit the client to increase its timestamp and try again.

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 409 status code and information for the client to try to fix the problem.

Each client can have its own logical clock to assign the timestamps for its transactions. The client's logical clock time is updated (increased) through the information returned in the HTTP 409 messages. Another option is to centralize the timestamp assignment into an entity which can work similarly a transaction manager [1, 7, 23].

In the purchase of tickets example, server A hosts the RESTful service responsible for the game tickets, and the initial values for its attributes are number of 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 number of tickets = 500, WTM = (15, x), RTM = (30, x), and LPW = [ ]

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 representation within the available tickets number.
Tickets Booking	`/ticketsforgame/booking/{TS}`	PUT	Create or update a booking at TS. Also used to commit or abort the booking.
	`/ticketsforgame/booking/{TS}`	GET	Retrieve the status of the booking created at TS (e.g., pending, aborted, completed).

Consider the scenario where both clients have succeeded in sending their prewrites. Client 1, whose timestamp is (50,a), has succeeded in sending its PW message to book 200 tickets through a PUT to the URIs `ticketsforgame/booking/50a` and `ticketsfortrain/booking/50a`. Client 2, whose timestamp is (40,b), has succeeded in sending its PW message to book 300 tickets through a PUT to the URIs `ticketsforgame/booking/40b` and `ticketsfortrain/booking/40b`. Thus, the LPW of both servers have two entries, one for each client.

Starting from the above scenario and assuming that the main objective is to maximize the sale of tickets and clients satisfaction, the general business rules [2] of servers can be stated as follows. (i) The sum of already accepted reservations (prewrites) must not exceed the current amount of tickets available. (ii) If it is possible at the moment of the request receiving, the number of tickets already reserved should be kept as large as possible. (iii) Potential clients should be kept in waiting lists to be notified when its purchase requests can be met – potential client is someone who can possibly purchase the tickets. (iv) The reservations are ordered by timestamp and must be committed in the same order – so, clients are served in timestamp order.

It is worth noting that if the only rule that must hold is the general rule (i) and overwrite of prewrites already entered in LPW is allowed, the original TS2PC4RS algorithm works.

However, if there are more business rules like the general rules (ii) and (iii), the TS2PC4RS algorithm needs some extensions, in addition to the permission to accept overwrite of prewrites already accepted. To describe the extensions, let us consider some business rules with more detail.

The client of the purchase of tickets example can send update requests to increase or decrease the amount of reserved tickets. When the client wants to buy a larger amount of tickets, it sends an increase update request. Otherwise, when the client wants to buy a lesser amount of tickets, it sends a decrease update request.

(a) If the client wants to increase the tickets amount requested, but the general rule (i) rejects the change, the client can request that its prewrite update request be stored to run as soon as possible (general rule (iii)), e.g., after an abortion of some other transaction, which increases the number of tickets available.

    The server may store the increase update request in a waiting list called *update-wait-list* (UWL). In this case, when the amount requested becomes available, the server updates the request in the LPW, removes it from the UWL, and notifies the client.

(b) If the client wants to decrease the tickets amount requested, we should follow the general rule (ii).

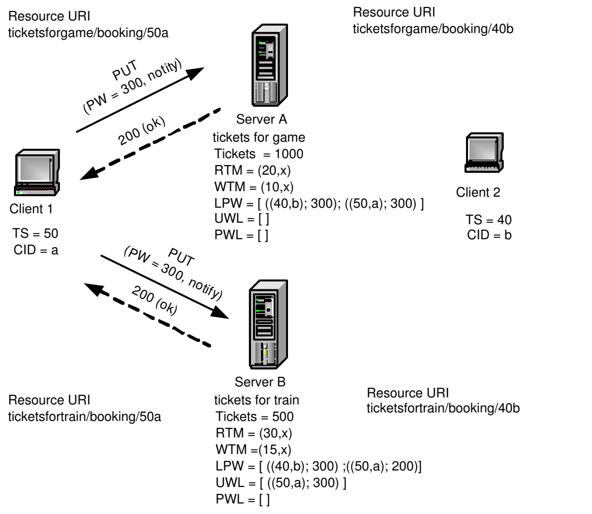

Figure 1: Client 1 updates its reservation request.

For this purpose the server can maintain a list of potential clients who had their first reservation request denied and had asked to be notified when tickets became available (general rule (iii)). Let us call this list the *prewrite-wait-list* (PWL). When the amount requested becomes available, the server accepts the prewrite request and notifies the client.

Therefore, when the client requests to decrease the requested amount of tickets, the server evaluates the PWL in order to keep the number of tickets reserved as large as possible (general rule (ii)) by trying to match the decrease request with the prewrite requests in the PWL.

The server can also check the UWL to find requests that can be used to match the decrease request with the increase prewrite requests in the UWL. However, nothing can be made if both lists, UWL and PWL, are empty upon the decrease request receiving, and so, the decrease prewrite request is just accepted – the corresponding PW in the LPW is decreased.

At client side it is not interesting to break the prewrite update into an abort and a new prewrite because the client may lose his/her advantage over the other clients as described in Section 1. Moreover, if the client has to restart its transaction with a new timestamp, its new prewrite will probably have a worse position in the LPW than the aborted one by the general rule (iv).

Continuing the above example scenario, let us assume that the client 1 decides to increase its reservation of tickets up to

300. Client 1 sends a prewrite update request to the server B through a HTTP PUT as illustrated in Figure 1. The client 1 also asks the server that if its request cannot be implemented immediately, the server can insert the request in the UWL. This is made through the *notify* parameter. By general rule (i), the server B cannot update the client 1's prewrite immediately. Thus, server B puts the prewrite update request in the UWL, keeping him/her as a potential client (general rule (iii)).

Client 1 also sends a prewrite update to the server A, who proceeds the update immediately as the available tickets amount is sufficient to obey general rule (i) (Figure 1).

On the other hand, client 2 decides to decrease its reservation of tickets down to 150. Client 2 sends prewrite update requests to the server A and B as illustrated in Figure 2. Both servers proceed the update immediately. However, server B has one increase update in UWL that can be implemented due to the availability of new tickets caused by the client 2's decrease request. Thus, server B moves the request ((50,a);300) from UWL to LPW and notifies client 1 about the accomplishment of its increase update request as it can be seen in Figure 3 (general rule (ii)).

At this point, either client can commit its transaction by setting the booking state to "completed" in the same way already described in the original example in [9].

In order to demonstrate the use of the PWL, let us consider a new but similar scenario where the client 2, whose timestamp is (40,b), has succeed in sending its PW message to book 300 tickets to both servers. Thus, the LPW of both servers have one entry.

Suppose that the client 3, whose timestamp is (60,c), needs to buy only train tickets because client 3 already has tickets for the game. So, client 3 sends its first prewrite request to buy 300 train tickets, which cannot be implemented at the moment because of general rule (i). Then, as client 3 has asked to be notified if tickets become available, server B puts client 3's prewrite request in the PWL – keeping a potential client (Figure 4).

Consider that client 2, based on some event, wants to decrease its reservation of tickets down to 200. Thus, after processing the client 2 decrease update request, the server B identifies the client 3's prewrite in the PWL, moves the prewrite from PWL to LPW and notifies the client 3 as shown in Figure 5. After the server B's notification, client 2 or client 3 can commit its transaction.

It is worth noting that the UWL and PWL concepts can be combined in favor of maintain a unique waiting list. If the business requires, the original prewrites should be distinguishable from the prewrite updates.

2.3 TS2PC4RS Extended Algorithm

Considering the issues discussed in section 2.2, the step 4 of the original TS2PC4RS algorithm is extended as follows in order to allow the prewrite updates based on the application business rules. The business rules cover everything that the business claims that must be evaluated in the data item manipulation, including the assurance of the consistency maintenance during the data item states transfer.

The RESTful services are responsible for the business rules verification since the RESTful services control the access to the data items.

The TS2PC4RS algorithm extension is described below.

```
4 - For prewrite operations:
```

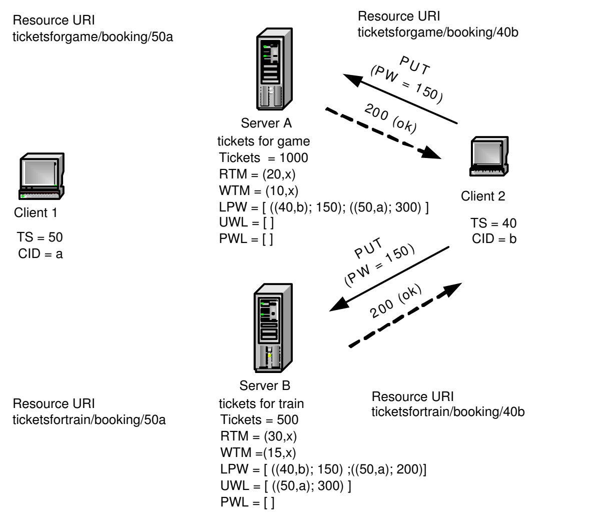

Figure 2: Client 2 updates its reservation request.

```
If TS < RTM(x) or TS < WTM (x) then
   reject the PW operation and restart the transaction;
else
   If the PW operation is not already in LPW then
      If the PW can be executed obeying
      all the business rules set then
         put the PW operation and its TS
         into the LPW; //first PW at this TS
      else
         If the client has asked for notification then
            put the PW operations and its TS into PWL;
         else
            reject the PW operation and restart
            the transaction;
   else // it is an prewrite update.
      If the PW can be executed obeying
      all the business rules set then
         replace the PW operation in the LPW with this
         new PW; //the TS must be the same
      else
          If the client has asked for notification then
             put the PW operations and its TS into UWL;
          else
             reject the PW operation and restart
             the transaction;
```

The step 4 is extended to take into account the application domain business rules. The cases dealing with the timestamp verification remain as the TS2PC4RS original algorithm, i.e., if the transaction timestamp (TS) is less than WTM or RTM the transaction must be aborted.

With this extension, each REST resource – in addition to WTM, RTM and LPW – has the following additional attributes within its representation: its *update-wait-list (UWL)* and its *prewrite-wait-list (PWL)*.

If the prewrite is not in the LPW – the prewrite is the first at the corresponding timestamp – and if the prewrite does not corrupt any business rule, it is inserted into the LPW. Otherwise if the prewrite is not in the LPW, but the prewrite corrupts some business rule, the RESTful service puts the prewrite in the PWL after checking that the client had asked to be notified if tickets become available.

If the prewrite is already in the LPW and the prewrite does not corrupt any business rule, the prewrite update is made replacing the PW already in the LPW. Otherwise if the prewrite is already in the LPW, but the prewrite cor-

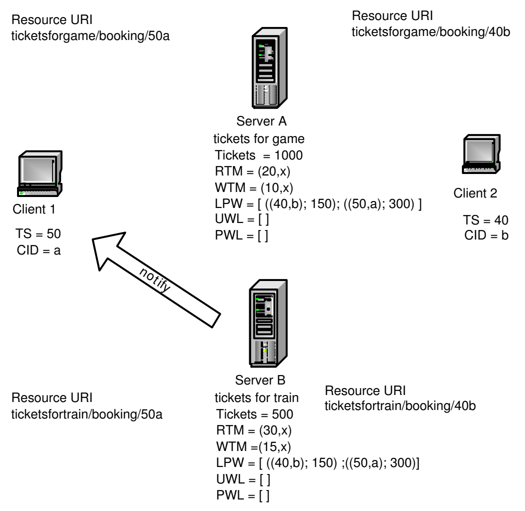

Figure 3: Server B identifies and implements a increase update request.

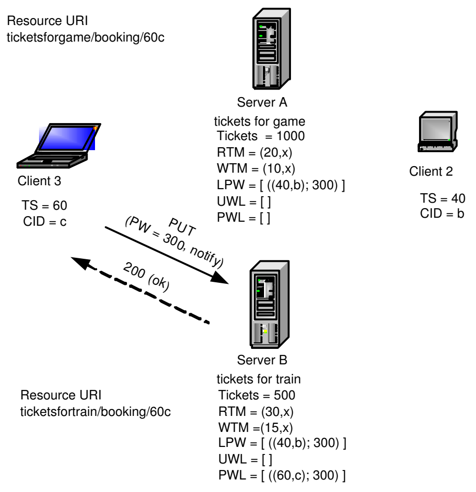

Figure 4: Client 3 requests a prewrite to buy 300 train tickets.

rupts some business rule, the RESTful service puts the prewrite in the UWL after checking that the client asked to be notified if tickets become available.

In the cases that the prewrite is put into PWL or UWL, the RESTful service must also check the possibility of the prewrite be accepted due to future events. For the example, if a client aborts the transaction or decreases the amount of tickets requested, other clients can have their prewrites eventually accepted. Thus, the prewrites can be put into PWL or UWL only if they can eventually be accepted.

The commitment and abort procedures of the original TS-2PC4RS algorithm have to be extended to evaluate UWL and PWL. When a commit/abort is requested, all UWL and PWL references to the prewrite being committed or aborted must be canceled and then the original commitment or abort procedure can be made as described in TS2PC4RS [9].

The abort procedure has a new particularity. When a prewrite is aborted, some tickets are made available again. Thus, when executing an abort, UWL and PWL must be checked due the number of tickets that became available. So, UWL and PWL are used to soften the purchase loss by the corresponding server.

3. ANALYSIS

In the situation explained in the example described in [9], when a REST client interacts with two RESTful services within a transaction, it must send prewrite requests to both services. If client's requests are accepted only by one of the services, the client identifies that it cannot complete the transaction successfully and then, in the all-or-nothing case, it aborts the transaction by sending an abort message to the RESTful service that has accepted the prewrite. The client can also partially commit the transaction by committing only the successful prewrite, if its business rules allow it.

However, through the ability to update the prewrites already accepted by the servers, the client does not need to abort the whole transaction or to partially commit the transaction, leaving the unsuccessful prewrite. The client can update its successful prewrite in order to increase the chances of success with the unsuccessful prewrite. Thus, the transaction execution can be completed in a more productive way to the client.

For example, if the client – in the purchase of tickets example – manages to send prewrite only to server A, because server B does not have the requested tickets amount. The client does not need to abort the transaction. The client, after evaluating the situation, may decide to decrease the tickets amount requested to both servers, and so, in the same transaction, the client sends an update prewrite to the server A and a usual prewrite to server B. So, the client increases the chances of transaction success.

The ability to improve the business rules support in the client-server interactions in the REST domain provides advantages to both clients and servers. As shown in the example, the servers enhance their selling capability through the waiting-lists (PWL, UWL). The sales that would be lost due to lack of available tickets, can be softened by putting the prewrites in these waiting-lists in order to store potential clients. The clients enhance their consumption power through the same waiting-lists, i.e., they have the option to stay in a queue in order to be served as soon as possible.

Through the TS2PC4RS extension, we have introduced both abilities in the original TS2PC4RS algorithm: prewrite update and business rules improvement support. So, the TS-2PC4RS can be extended to absorb the nuances of various application domains. The TS2PC4RS extensibility eases the adjustment to different environments as has been demonstrated through the purchase of tickets example, reaching the TS2PC4RS Extended algorithm in Section 2.3.

Based on the general and detailed business rules stated in Section 2.2, we deal only with increase prewrite update requests in the UWL. So, the decrease prewrite update requests are not put in the UWL due to the stated rules which have as main objective the maximization of tickets sold and clients satisfaction.

With the stated rules, the clients have no interest in waiting for having their decrease prewrite requests accepted, but if we consider a new general rule (v): Apply penalties to clients that update their already accepted prewrites to de-

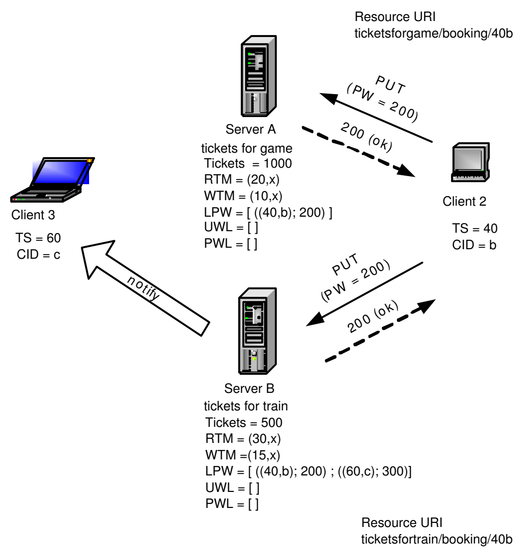

Figure 5: Client 2 updates its reservation request and Server B identifies and implements the prewrite request in the PWL.

crease the tickets amount requested; the storage of decrease update requests in PWL becomes interesting to the clients.

In order to fulfill the general rule (v), the server can store the decrease update requests that cannot be softened in the UWL. The increase and decrease requests must be distinguishable in the UWL. As already described in Section 2.2, in some cases, the decrease update requests cannot be softened because UWL and PWL are empty and so the attempt to keep the quantity of tickets already reserved as large as possible (general rule (ii)) cannot succeed at the moment of the decrease request receiving, and so, the decrease request may be put in the UWL, if the client asked to put its request in a waiting-list to avoid the penalties of general rule (v).

The behavior would be very similar to that described in item (a) of the detailed business rules for the prewrite operation (Section 2.2), because it is also an prewrite update request. The difference is that it is necessary to wait for some other client that wants to increase its prewrite or that is sending its first prewrite request, in order to match the decrease update requests in the UWL with the increase requests or the new prewrite requests. In this way, the clients avoid the penalties stated by the possible general rule (v) and the chances to keep the number of tickets already reserved as large as possible increase (general rule (ii)).

The additional attributes used in the REST resources (RTM, WTM, LPW, PWL, UWL) in order to enable the extended TS2PC4RS are used to control the data access in transactions. They have nothing to do with the resource original purpose. An intermediary component can be introduced to control the additional attributes. With an intermediary component in the server side, it is possible to free the original resource to deal with control data. The RTM update caused by a read operation (R) sent through a GET request can be viewed as a log record – functionality often handled by an intermediary component.

The use of the transaction timestamp (`/{TS}`) in the resources URIs allows the usage of caching intermediary components. The cache entry may be invalidated after a prewrite, write (commit or abort) or a prewrite update.

The RTM update through a GET request does not completely obey the HTTP semantics, but considering the separation of concepts: original REST resource and control data, this issue may be an acceptable relaxation in order to obtain the transaction control benefit, as the RTM update can be implemented in a transparent way to the original resource. The RTM update does not cause a state change in the resource, it changes only the access control data.

As for the original TS2PC4RS algorithm [9], our proposal may need some mechanism of prewrite cleaning if the transactions take too long. In this situation, the updates of WTM may progress too slowly and LPWs, UWLs and PWLs 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 (and its updates). The agent has to have the permission from the coordinator beforehand.

The original TS2PC4RS [9] can be compared to the Baker and Charlton's work [1] as it traditionally deals with the two-phase-commit protocol. Baker and Charlton use resources as units of work, i.e., they define resource for the transaction manager and for each transaction. They do not deal with resource specific behavior at the server side.

The TS2PC4RS[9] and its extensions guide the behavior of REST resources used in a transactional context. One goal is that TS2PC4RS can be applied to existing resources without causing major impacts on the resource provider implementation.

Through timestamp ordering, it is possible to verify that the "version" which the client wants is not influenced by ongoing transactions. Thus, the client can read the resource and continue his/her activity with the assurance that the resource state at the stated timestamp will not be changed. If the "version" which the client wants is affected by the ongoing transactions, it is possible to return the *updated view of the data item*, leaving to the client the decision of what he/she wants to do.

In summary, the use of timestamp allows the sequencing of the clients requests at each resource provider, even if they arrive out of order. Thus, before the transaction end, the clients requests can be prioritized using the timestamp order.

Despite, in the extended TS2PC4RS, the client performs the role of the transaction manager (TM), we could also use a TM apart as described in the references [1, 23], i. e., the client would ask TM to create a transaction that would be identified by the timestamp returned to the client. However, some considerations should be taken to assure that the TM meets the business goal of the client appropriately.

For example, the TM can receive all the indications of the client's needs to guide the transaction previously. Another possibility is to receive the client indications during the transaction. In this case, the TM exposes the current situation to the client and asks what him/her wants to do.

4. CONCLUSIONS

The application domain business rules have a essential role to deal with transactions using the TS2PC4RS algorithm in the REST domain. Each application domain has its own rules which guide the software execution, in this case, the RESTful services.

In this work, we exemplify the manipulation of prewrite updates through some business rules of the purchase of tickets example, and it resulted in an extended version of the TS2PC4RS algorithm. Despite the usage of the purchase of tickets example, the proposed extensions to the TS2PC4RS algorithm are general enough to deal with different applications domains. The proposed waiting-lists, PWL and UWL, are loosely-coupled with the example, and so, they can be used in a variety of domains.

As the business rules depend on the application domain where the RESTful services are used, the concrete solutions for each case can be slight different in order to absorb the nuances of the application domains. Thus, the TS2PC4RS extensibility proves to be an essential characteristic.

Moreover, despite the uniform interface – one of the REST architectural style's constraints – increases the overall system simplicity, the uniform interface constraint has as the primary trade-off the likelihood to decrease the clients efficiency due to having to deal with more general data formats [19].

Thus, the use of the extended TS2PC4RS algorithm needs some form of contract, which might predicts the necessary information to make the interactions between clients and RESTful services clear and unambiguous. Some of the information that may be included in the contract are the message payload format – which can use XML, JSON, plain text, or any other desired format –, the supported TS2PC4RS operations (read, write, prewrite, update prewrite) which, as presented, map gracefully to the HTTP methods, and any further relevant information to the client-server interaction. The RESTful services can provide a timeout system to allow control of how long a update prewrite remains in the waiting list (UWL, PWL). Clients can send such an information along with the update prewrite request. Such a use of a timeout system should also be described in the contract.

However, it is not about to turn the uniform interface simplicity into a specialized contract. The aim is to maintain the reusability increased by uniform interface constraint, but with the necessary documentation to clarify the allowed interactions between clients and servers which use the extended TS2PC4RS algorithm. So it is not recommended to put vendor-specific restrictions or features that go beyond the REST constraints into the contract, because it can lead to interoperability problems.

As future work we plan to investigate the recovery model for TS2PC4RS. We hope to allow the TS2PC4RS to deal with host failures (client and RESTful services) and with communication failures (e.g. loss of message) during the transactions.

5. REFERENCES

[1] M. Baker and S. Charlton. The web: Distributed objects realized! OOPSLA, 2007. http://www.slideshare.net/StuC/oopsla-2007-the-web-distributed-objects-realized.

[2] BRG. Defining business rules - What are they really? The Business Rules Group, 2000. http://www.businessrulesgroup.org.

[3] S. Ceri and G. Pelagatti. *Distributed Databases, Principles and Systems.* McGraw-Hill, 1985.

[4] R. L. Costello. Building web services the REST way, s.d. http://www.xfront.com/REST-Web-Services.html access March 2008.

[5] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, USA, 2000.

[6] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and Digest access authentication. RFC 2617, June 1999. http://www.ietf.org/rfc/rfc2617.txt.

[7] S. Jacobs. A question about REST and transaction isolation, February 2004. http://www.stylusstudio.com/xmldev/200402/post30270.html.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[9] L. A. H. S. Maciel and C. M. Hirata. A Timestamp-Based two phase commit protocol for web services using REST architectural style. *Journal of Web Engineering*, 9(3):266–282, September 2010.

[10] OASIS. Oasis web services reliable messaging (WSRM) TC, November 2004. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm.

[11] OASIS. Oasis web services security (WSS) TC, February 2006. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

[12] OASIS. Oasis web services transaction (WS-TX) TC, July 2007. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx.

[13] OASIS. Web services coordination (WS-Coordination), July 2007. http://docs.oasis-open.org/ws-tx/wscoor/2006/06.

[14] OAUTH. An open protocol to allow secure api authorization in a simple and standard method from desktop and web applications, April 2010. http://oauth.net/.

[15] OIDF. Openid is a safe, faster, and easier way to log in to web sites, 2010. http://openid.net/.

[16] T. O'Reilly. REST vs. SOAP at Amazon, April 2003. http://www.oreillynet.com/pub/wlg/3005.

[17] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. big web services: Making the right architectural decision. In *17th International World Wide Web Conference (WWW2008)*, pages 805–814, Beijing, China, April 2008 2008.

[18] L. Richardson and S. Ruby. *RESTful Web Services.* O'Reilly & Associates, Sebastopol, California, May 2007.

[19] S. Vinoski. Serendipitous reuse. *IEEE Internet Computing*, 12(1):84–87, 2008.

[20] W3C. Naming and addressing: Uris, urls, ... http://www.w3.org/Addressing/URL/uri-spec.html access date March 2008.

[21] W3C. Simple Object Access Protocol (SOAP) 1.1, May 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[22] W3C. Web Services Description Language (WSDL) 1.1. Note, March 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[23] J. Webber, S. Parastatidis, and I. Robinson. *REST in Practice: Hypermedia and Systems Architecture.* O'Reilly Media, Inc., 2010.