SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2012)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2151

Fault-tolerant timestamp-based two-phase commit protocol for RESTful services

Luiz Alexandre Hiane da Silva Maciel and Celso Massaki Hirata*,†

Instituto Tecnológico de Aeronáutica (ITA)—Praca Marechal Eduardo Gomes, 50 Vila das Acácias, CEP 12228-900, São José dos Campos, São Paolo, Brazil

SUMMARY

Service-oriented architecture provides interoperability and weak coupling features for software development. Representational state transfer (REST) is an architectural style that has attracted attention in the SOA domain as it allows the development of Web services based on original principles of the World Wide Web. Unlike Web service specifications, which are based on Simple Object Access Protocol and Web Services Description Language, REST does not provide 'official' standards to address non-functional requirements of services, such as security, reliability, and transaction control. The timestamp-based two-phase commit protocol for RESTful service (TS2PC4RS) algorithm specifies concurrency control of RESTful services during a transaction. An extension of the TS2PC4RS specifies the concurrency control of the Web services considering the update operation to meet some business rules. However, neither algorithm addresses transaction control when failures occur. In long-term transactions, failures can occur and compromise the success of Web service applications. Two common failures traditionally considered in the analysis of protocols are host and connection failures. The aim of this paper is to address fault tolerance for TS2PC4RS and its extension. A fault-tolerant protocol based on timeout and log records is proposed. The fault-tolerant protocol provides support for the host and connection failures that may occur during a transaction execution. The fault-tolerant mechanisms are used to meet the application domain business rules that guide the behavior of RESTful services. We describe the protocol using scenarios when failures occur. Copyright © 2012 John Wiley & Sons, Ltd.

Received 25 September 2011; Revised 15 May 2012; Accepted 23 July 2012

KEY WORDS: fault tolerance; reliability; concurrency control; distributed system; REST; transaction; timestamp; Web services; architectural style

1. INTRODUCTION

Representational state transfer (REST) [1] is an emerging technology that has been gaining attention in the SOA domain because its foundation lies in the original design principles of the World Wide Web. REST provides a new abstraction for publishing information and giving remote access to application systems. The usage of these original principles makes the consumption of the services by clients easier, which allows service providers to attract a larger user community [2, 3].

Web service specifications (WS-*) are a set of specifications for the development of services based on Simple Object Access Protocol (SOAP) [4] and Web Services Description Language [5]. WS-* provides support for non-functional requirements such as security, reliability, and transaction control. The specifications include *WS-Security* [6], *WS-Reliability* [7], *WS-Transaction* [8], and *WS-Coordination* [9]. One drawback of the WS-* standards is their level of complexity, which often hinders their adoption in applications developed to operate on the Web.

*Correspondence to: Celso Massaki Hirata, Instituto Tecnológico de Aeronáutica (ITA)—Praca Marechal Eduardo Gomes, 50 Vila das Acácias, CEP 12228-900, São José dos Campos, São Paulo, Brazil.

†E-mail: hirata@ita.br

On the other hand, most of the non-functional requirements in RESTful Web services are not addressed by an ‘official’ standard. The reason is that REST is an architecture style, so it is used mainly to understand and design Web services [10].

There is still a gap for new proposals to deal with transactional control support ranging from atomicity, consistency, isolation, durability to long-running transactions in the Web service domain. So, to address Web service concurrency control, Maciel and Hirata [11] have proposed the *timestamp-based two-phase commit (2PC) protocol for RESTful services* (TS2PC4RS). The TS2PC4RS algorithm uses a timestamp-based technique [12] with the 2PC protocol [13] to control concurrent accesses to REST resources.

Maciel and Hirata have also proposed an extension for TS2PC4RS [14] to improve the way clients update their ‘reservations’, known as prewrites. The reason for opportunistic updates is the possibility for clients in a long duration transaction to change their minds after starting the transaction. To provide the ability to update the prewrites, the TS2PC4RS extension [14] considers the application domain business rules to extend TS2PC4RS [11].

Web service transaction models [15] deal with the consistency and reliability of loosely coupled Web service applications. The WS-Transaction effort is cited as the *de facto* standard for SOAP-based Web service transactions [15]. TS2PC4WS and WS-Transaction can be applied in different contexts. If the Web services can be abstracted as resources (data items), TS2PC4WS is more appropriate. However, if the services are focused on specific actions of the application domain, the WS-Transaction is more suitable. The decision on which approach to use must consider the business requirements involved, as well as the non-functional requirements that must be met. A detailed comparison between TS2PC4RS and WS-Transaction specification is described in [11].

Concurrency control is one of the key properties that transactions must follow. However, other issues have to be addressed, such as atomicity, isolation, and durability. In TS2PC4RS, atomicity and isolation are relaxed, and durability is not addressed. A fault-tolerant TS2PC4RS that considers the relaxed form of atomicity and isolation and durability is essential to the success of long duration Web service transactions.

In this paper, we propose procedures to deal with failures that may occur during a transaction execution using TS2PC4RS. The aim is to provide host and connection failure tolerance specifically for TS2PC4RS. Despite the fact that our fault-tolerant approach is specific to TS2PC4RS, we believe that the ideas and concepts can be adapted and applied to other protocols and algorithms used to control Web service transactions.

There are two aspects of reliability that must be somehow considered—correctness and availability [13]. It is important that a system behave correctly according to its specifications, and the system is available when necessary. In certain cases, there is a trade-off between correctness and availability [13] because it is possible to increase the availability by agreeing to continue the transaction execution under circumstances that increase the risk of obtaining inconsistent results. On the other hand, using a very cautious strategy, such as halting the transaction whenever anomalies emerge, reduces availability.

In some applications, correctness is an absolute requirement, and information corruption caused by errors is not tolerated (e.g., banking applications). Other applications may tolerate the risk of inconsistencies to achieve greater availability.

In the TS2PC4RS fault-tolerant approach proposed in this paper, the trade-off is originated from the possibility for a client, say client A, to have some period, which can be long, to decide about the commitment or abortion of its transaction. The transaction uses records through read and write operations during its execution. However, because the transaction can be of a long duration, we allow that the records are accessed by read operations of other transaction, for instance, initiated by client B. This possibility increases availability (for read operation); however, it compromises correctness because client B does not know whether client A will commit or abort.

The sections of this paper are organized as follows. Section 2 recalls some of the key aspects of TS2PC4RS and its extension. The description of TS2PC4RS is illustrated through an example. Section 3 describes the general model of the fault-tolerant protocol for TS2PC4RS. We describe the assumptions to use the fault-tolerant TS2PC4RS. Section 4 describes the recovery procedures of host and connection failures through scenarios in which these failures occur. In Section 5, we describe some implementation and a practical experience to show the feasibility of the fault-tolerant TS2PC4RS. In Section 6, we describe how to make the extended version of fault-tolerant TS2PC4RS using the same concepts discussed. The related work is described in Section 7. Section 8 presents a discussion and the conclusions.

2. TS2PC4RS ALGORITHM

Clients are responsible for coordinating the transaction, which can involve various RESTful services. The services must implement the TS2PC4RS algorithm to play the role of a transaction participant/agent.

The client can send *prewrite* operations to the RESTful services, which, on the basis of their business rules, analyze the accomplishment of the client's request. Business rules dictate what the Web services should implement to achieve the business goals. Business rules can be understood, from the information system perspective, as statements that define or constrain aspects of the business. They are intended to assert business structure or to control or influence the conduct of the business. Thus, a business rule expresses specific constraints on the creation, update, and removal of persistent data in an information system [16].

To deal with failures, we recall the main aspects of TS2PC4RS [11].

Section 2.1 recalls some of the key aspects of TS2PC4RS. Section 2.3 recalls the main aspects of the extended algorithm.

2.1. Timestamp-based two-phase commit protocol for RESTful service algorithm

In this subsection, we describe our previous work, the TS2PC4RS algorithm for concurrency control in the RESTful service domain [11].

The TS2PC4RS is a collection of procedures to deal with read, prewrite, and write operations on data items. Every write operation W is preceded by a prewrite operation *PW*. A unique timestamp is assigned to each transaction in its origin. Each read operation R, write operation W, and prewrite operation PW has the transaction's timestamp *TS*.

In the all-or-nothing decision, 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 the services. If the client receives a not-ready message, it aborts the transaction, sending *abort* messages to all the services.

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

Figure 1 describes the exchange of messages between the coordinator and the Web services when implementing TS2PC4RS. The client assumes the role of the *2PC coordinator* responsible for coordinating the transaction. It starts the transaction by sending requests through prewrites to the RESTful services involved in the business process. The RESTful services assume the role of the *2PC agents*. Thus, if the service can perform the operation requested, it replies with a *ready* message to the coordinator-client. Otherwise, the service replies with a *not-ready* message.

In the all-or-nothing case, if the coordinator site receives a ready message from all the agents, the transaction is committed. So the coordinator sends the *commit* message (the decision) to all agents.

Therefore, agents determine whether the prewrites are accepted on the basis of their 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 prewrite conversion into write if the coordinator decides to commit the transaction. Agents discard prewrites if the coordinator does not commit the transaction, by sending abort messages.

The algorithm is reproduced in Listing 2.1. The *LPW* is a list of buffered prewrites on a data item (x) arranged in a timestamp order. If *LPW* is not empty and an agent receives a read operation R with $TS(R) > WTM$ and $TS(R) >= TS(\text{first}(LPW))$, the read operation should not be allowed

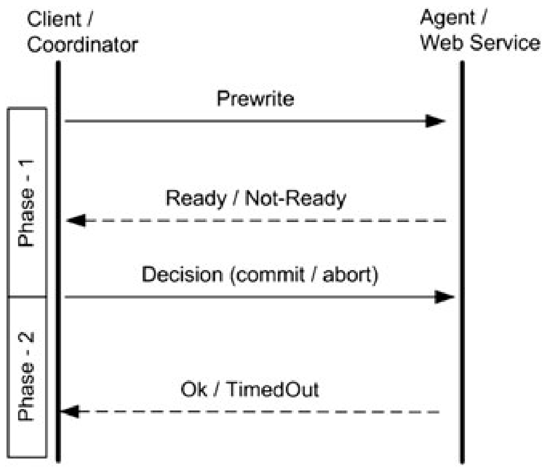


Figure 1. Exchange of messages in timestamp-based two-phase commit protocol for RESTful services.

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 being more responsive and returning some information to the client, relaxing the isolation property. So, in this situation, TS2PC4RS is more flexible because it allows the return of the *data item updated view*, which contains the data item value in the last committed write operation, the *WTM*, and a *LPW* sub-list that contains all *PW*s with timestamp less than or equal to $TS(R)$, that is, the ongoing *PW*s that may impact the read at $TS(R)$.

It is also possible to return some computed value on the basis of the *LPW* sub-list. For example, assuming that all pending prewrites commit, it may be of interest to return the presumable data item value with the updates applied, which will have $WTM = \text{last}(LPW_{\text{sub-list}})$. The client must know that it is a volatile read, as the value read can be changed by ongoing transactions.

Hence, the clients, on the basis of the *data item updated view* and their business rules, can decide what they want to do. For example, depending on what the client is performing, it can abort the transaction, it can try to predict the data item value at the $TS(R)$ (if it is not already calculated by the agent), or it can wait some time 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 removed from the *LPW*.

When the coordinator sends write operations W to the participating agents, the W operation must have an associate *PW* (with the same *TS*) previously sent to the agents. Listing 2.2 shows the procedure that must be accomplished when the agents receive W, indicating the transaction commitment.

A sequence of *PW* marked for update data means that there is an *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 procedure in Listing 2.3 must be performed by the agents. In this case, the *PW* is removed from the *LPW*, and if there is a sequence of *PW* marked for update data, the sequence is executed and the *WTM* is updated with the timestamp of the last *PW* in that sequence.

For more details, the reader can refer to [11].

2.2. *Purchase-of-tickets example*

The transaction is made of two operations: an operation to purchase tickets for a basketball game and an operation to purchase train tickets to go to the city the game is being played in. The client's objective is to buy a certain number of tickets for the game, together with the train tickets. Initially, clients want to buy the same amount of tickets for the game and for train seats. Table I provides an

Listing 2.1 Algorithm for Timestamp-based Two Phase Commit Protocol for RESTful Services. Source: [11].

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

1 - A unique timestamp is assigned to each transaction at its 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:
      WTM(x) - the largest timestamp for a write operation  on x;
      RTM(x) - the largest timestamp for a read operation  on x;
      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 an 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
                 // return "data item updated view"
                 execute read and return the data item value committed
                 at WTM, WTM, and LPW sub-list until TS;
```

Listing 2.2 TS2PC4RS commitment procedure. Source: [11].

```
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 onwards
   Mark the PW for update-data;
```

Listing 2.3 TS2PC4RS abortion procedure. Source: [11].

```
Remove PW from LPW;  // in the case of abortion
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);
```

Table I. 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 ticket number.
Ticket 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).

Source: [11].

overview of the main resources, uniform resource identifiers (URIs), and operations for the service responsible for the game tickets. The train ticket service has similar resources.

As described [11], each REST resource that uses the timestamp concurrency control has the following attributes within its representation.

- The largest write operation timestamp (*WTM*).
- The largest read operation timestamp (*RTM*).
- The list of buffered prewrites (*LPW*).

The RESTful services implement the 2PC *agents* that control access to the data items. The REST clients implement the 2PC *coordinator*. In this paper, the terms Web service, RESTful service, and service have the same meaning and play the role of the 2PC agent; the terms coordinator, client, and coordinator-client have the same meaning and play the role of the 2PC coordinator.

The client sends read operations (R) through GET messages and 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 about the conflict. For example, if R is rejected because of the *WTM*, the value of *WTM* is added in the response message to allow the client to increase its timestamp.

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 decide on the transaction.

Figures 2–4 illustrate the purchase-of-tickets scenario described in [11]. The filled arrows represent the requests made by clients. The dashed arrows represent the responses sent by the servers

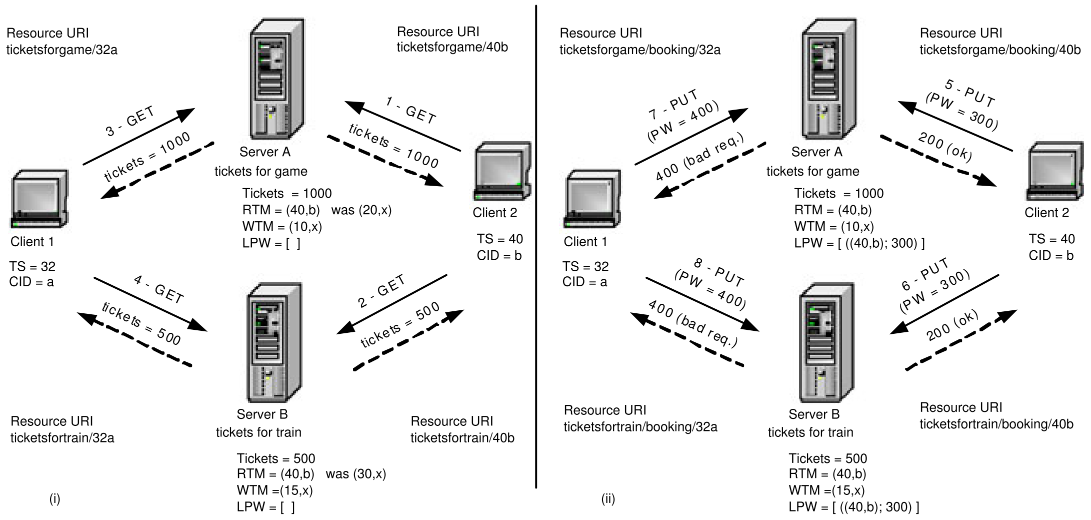


Figure 2. (i) Each client retrieves the account representations. (ii) Only client 2 succeeds in sending prewrites. Source: [11]. CID, coordinator identifier; TS, timestamp; URI, uniform resource identification.

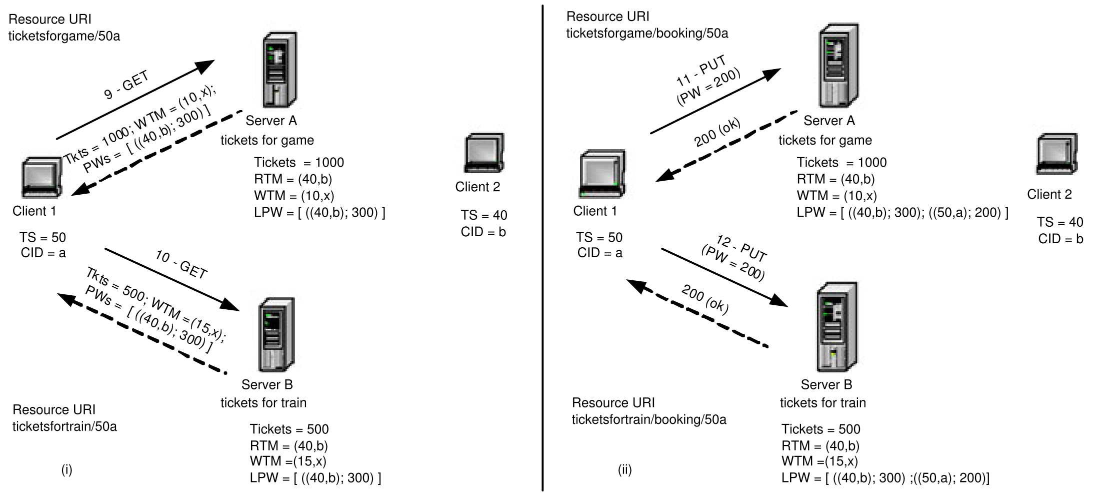


Figure 3. (i) Client 1 restarts and requests the resource representations again, but there are transactions in progress. So some more information is sent back to client 1. (ii) Client 1 now succeeds in sending prewrites. Source: [11]. CID, coordinator identifier; TS, timestamp; URI, uniform resource identification.

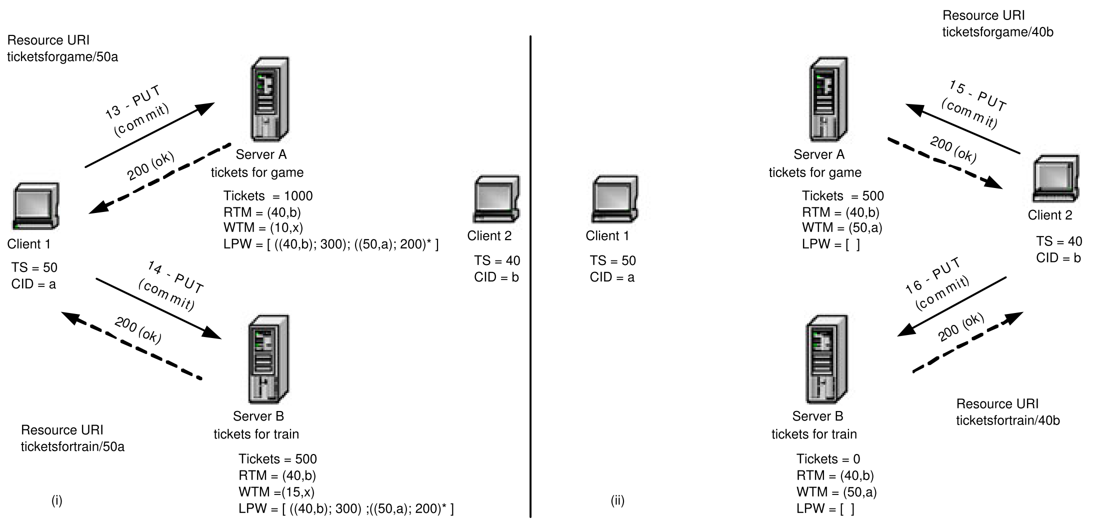


Figure 4. (i) Client 1 commits its transaction. (ii) Client 2 commits its transaction. Source: [11]. CID, coordinator identifier; TS, timestamp; URI, uniform resource identification.

to clients. The requests are numbered to indicate the order of execution, and the responses indicate whether the requests are successful or not. Client 1's goal is to buy 400 tickets in both services; and Client 2's goal 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. Each coordinator has a unique ID, and a total order among all CIDs is assured. Client 1's initial timestamp is $(32, a)$, and client 2's initial timestamp is $(40, b)$.

The first step for each client is to obtain the representation of resources involved by sending an HTTP GET request to the servers. The request URI is `ticketsforgame/timestamp` to server

A and `ticketsfortrain/timestamp` to server B. Both clients obtain the representations of available game and train tickets (Figure 2i).

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

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

A response to the preceding request is illustrated as follows:

```
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 *WTM*s, 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 $\max(RTM, TS)$.

With the representations, clients check 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 in Figure 2ii, the URI `ticketsforgame/booking/timestamp` is used to send a prewrite to server A, and the URI `ticketsfortrain/booking/timestamp`, to server B.

At the time of sending the prewrites to the servers, client 1 cannot achieve a successful prewrite because the *RTM*s of both resources are greater than client 1's timestamp. Client 1, therefore, receives two *not-ready* messages in return to its prewrites, and client 1 must restart its transaction with a higher timestamp (Figure 2ii). On the other hand, client 2 successfully books their tickets and receives *ready* messages from the servers. So 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 unchanged. 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 unchanged.

A request content to book 300 tickets for the game may be as follows.

```
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, they restart their transaction with a higher timestamp, $TS = (50, a)$ and send HTTP GET requests to retrieve the resource representations again. Now, however, as shown in Figure 3i, the *LPW*s are not empty, they contain one prewrite with $TS(PW) = (40, b)$. So neither server can 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 a timestamp less than $(50, a)$. In this situation, the client must decide what to do, considering that some transactions have not been committed yet.

Client 1, with the resource updated views, can decide on the basis of their business rules. Let us assume client 1 notes that there are transactions in progress which, if they are committed, 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.

Thus, client 1 successfully books its 200 tickets and receives ready messages from the servers. As illustrated in Figure 3ii, 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 Listing 2.2. As, in servers A and B, the *PW*s to be committed are the second in the *LPW*s, they are just marked for update data in each resource. In Figure 4i, the asterisk represents the update data mark.

When client 2 commits, according to the commitment procedure (Listing 2.2), the resources execute the write operation, remove the committed *PW* from *LPW*, and update *WTM* with $TS(PW_{\text{removed}})$. The resources also remove the sequence of *PW*s 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 4ii.

Up to this point, we described the interactions between the coordinator and the Web services as proposed in our previous work—-TS2PC4RS [11]. This is necessary to clarify the context in which we are proposing the fault-tolerant approach, which is the aim of this paper.

2.3. *Timestamp-based two-phase commit protocol for RESTful services extended algorithm*

During the transaction execution, the client may also want to change their request to increase the chances of success with all the services, so the processing of prewrite is extended to accept updates [14].

The extended TS2PC4RS algorithm [14] is a previous work of ours that allows the prewrite updates on the basis of the application business rules. The business rules cover everything that the business claims must be evaluated in data item manipulation, including the assurance of consistency maintenance during the data item states transfer.

The term *prewrite update* refers to the second prewrite onwards sent using the same timestamp *TS* to change the first accepted prewrite.

In this paper, we use TS2PC4RS without the waiting lists proposed in [14]. The update request is analyzed by the Web service, and it is accepted or rejected. The reason for not addressing the waiting lists is the difficulty in assuring consistency in the event of host failure for the service.

Thus, if the prewrite is being updated, but the update does not corrupt any business rule, the update is made by replacing the prewrite in the *LPW* [14]. For more details about the extended TS2PC4RS, the reader can refer to [14].

3. TIMESTAMP-BASED TWO-PHASE COMMIT PROTOCOL FOR RESTFUL SERVICE FAULT TOLERANCE MODEL

In the timestamp 2PC protocol [13], the goal of the first phase is to reach a common decision, and the goal of the second phase is to implement the decision. The basic idea is to determine a unique decision for all agents with respect to committing or aborting all local sub-transactions.

In TS2PC4RS [11], the coordinator-client can partially commit the transaction, that is, the client is able to commit only the accepted prewrites. The result of the transaction has other options besides the 2PC all-or-nothing possibility in which either all the agents commit or all the agents abort. In the extended TS2PC4RS [14], the client can make updates until they reach the decision for the transaction.

Figure 1 illustrates an overview of the message exchange in TS2PC4RS. As described in Section 2.1, the coordinator starts the transaction phase 1 by sending prewrite requests to the RESTful services that the coordinator selects to participate in the transaction. A RESTful service that accepts the prewrite replies with a *ready* message to the coordinator. The service replies with a *not-ready* message if it cannot accept the prewrite. Once the coordinator receives the responses from all the services and the coordinator takes the decision, phase 2 is started. Afterwards, the coordinator sends the commit or abort messages to the corresponding services.

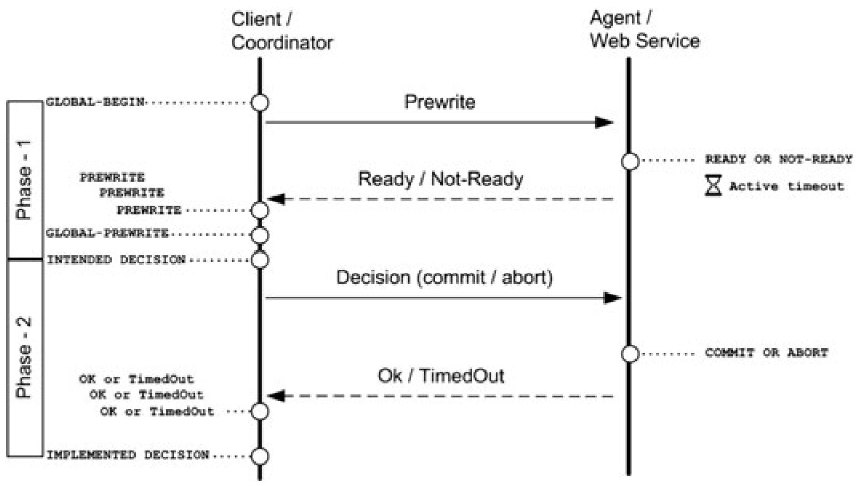


Figure 5. Exchange of messages and log records in the fault-tolerant timestamp-based two-phase commit protocol for RESTful services.

To deal with failures, we use a log file to record the TS2PC4RS main events. If an error occurs during the transaction execution, the transaction can be recovered from the error by using the saved information. Figure 5 complements Figure 1 with the main events that must be logged during the execution of a transaction to allow recovery from failures. The main events of the coordinator and Web services are represented by small circles and are detailed in Section 4. Briefly, the events that must be logged are the start/end of the first phase, the start/end of the second-phase, and the individual responses of the Web services in each phase. We assume that the end of the first phase and the start of the second occur at the same time.

Some restrictions must be assumed to allow the use of log records to provide objective recovery procedures. In this paper, we consider the following conditions to successfully finish a transaction.

c1. All the Web services accept the prewrites; otherwise, the transaction is aborted. If the coordinator does not receive a *ready* message from all the Web services in the TS2PC4RS phase 1, the coordinator decides to abort the transaction.
c2. The Web service is responsible for deciding when the timeout to receive the coordinator's decision is reached; however, it must inform the coordinator of the deadline (timeout). The coordinator follows the Web service decision.
c3. The coordinator decides on the basis of its business rules before the earliest deadline (the earliest Web service timeout). The Web service timeout should be long enough to allow the coordinator to reach and transmit the decision to all the Web services. The timeout also should enable the execution of a recovery procedure if an unexpected failure occurs.

Each Web service has a timeout mechanism to impose a time limit to receive the coordinator decision. If the Web service does not receive the decision before the timeout expiration, the Web service cancels the prewrite that has already been accepted. All the Web services that reach their timeouts cancel the accepted prewrite.

Our proposed protocol is resilient to failures in which no log information is lost. Log information is recorded in each Web service, as well as in the coordinator/client. The use of logs is the basic technique for implementing transactions in the event of failures [13].

A log record contains information for undoing or redoing all the actions performed during a transaction execution [13]. The undo actions must allow the reconstruction of the previous state of the system, that is, before the transaction execution. The redo actions must perform the transaction operations again.

4. FAULT TOLERANCE FOR TIMESTAMP-BASED TWO-PHASE COMMIT PROTOCOL FOR RESTFUL SERVICES

In this section, we describe the fault tolerance approach for TS2PC4RS. First, we describe the structure and the information in the TS2PC4RS' main events log records.

In our proposed protocol, the variables (*RTM*, *WTM*, *LPW*) controlled by the Web service providers must be logged to reconstruct them in the event of failures. After the TS2PC4RS variables during the restart are reconstructed, the recovery procedures in Sections 4.2 and 4.3 are executed.

The undo and redo operations must be idempotent [13]. Therefore, performing them several times must be equivalent to performing them once. This is important because the recovery process might fail and be restarted more than once. So the coordinator and the Web services must recognize and correctly process the repetitions demanded by the recovery procedures.

On the coordinator-client side, a log record for TS2PC4RS is composed of the following information:

1. the transaction timestamp and
2. the Web service uniform resource locators used in the transactions or some other Web service identification.

The main events that the coordinator/client must log are as follows:

- GLOBAL-BEGIN: The start of the first phase, when the coordinator selects the Web services to be used in the transaction.
- PREWRITE: The success in receiving a ready message from each Web service.
- ABORT: The decision to abort the transaction due to a not-ready message or an exception during phase 1.
- GLOBAL-PREWRITE: The end of the first phase, when it successfully transmits the prewrites to all participating Web services and receives the corresponding ready message.
- INTENDED DECISION: The start of the second phase, when the coordinator-client reaches a transaction decision, indicating which Web services must commit and which must cancel the prewrite.
- OK: Success in receiving the OK message from a Web service.
- TIMEOUT: Success in receiving a TIMEOUT message from a Web service.
- IMPLEMENTED DECISION: The end of the second phase, when the coordinator-client terminates the transaction.

The coordinator-client events are shown in Figure 5 on the client side represented by small circles. The TS2PC4RS protocol detailing the actions executed by the coordinator-client in phases 1 and 2 is described in Listing 4.1.

If the coordinator receives all the replies (ready or not ready messages), the coordinator intended decision is reached on the basis of its business rule restrictions. As the business rules depend on the application domain, it is not possible to predict the coordinator's exact behavior to reach the decision. The coordinator intended decision can be made on all-or-nothing or partial commitment (or partial abortion) of Web services.

On the Web services side, a log record for TS2PC4RS is composed of the following information:

1. the transaction timestamp;
2. the TS2PC4RS variables that are being updated;
3. the type of operation being executed (read, prewrite, and write (commit or abort));
4. the old TS2PC4RS variable values; and
5. the new TS2PC4RS variable values.

The main events that each Web service must log are the following:

- READY or NOT-READY: The prewrite acceptance or rejection.

Listing 4.1 TS2PC4RS Protocol for the coordinator (client).

```
// Phase-1 - Coordinator/Client sends requests to agents/web services and
//           receives replies with information.
Log GLOBAL-BEGIN;
Try{
   For each web service{
      Send prewrite message to and receive ready (or not ready) message
       (with deadline) from web service;
      If ready then{
         Log PREWRITE;
         Compute EarliestDeadline;
      }
      If not-ready then{
         Halt the For-iteration;
         Log ABORT;
         // Send abort message to the accepted prewrites or
         // Do nothing with the web services (the web services that
         // accepted prewrites will abort by timeout).
         End transaction;
      }
   }
   Log GLOBAL-PREWRITE; // All Prewrites successfully sent
}Catch (Connection Exception Or Server Exception){ // The phase-2 is not
                                                   // necessary!
   Halt the For-iteration;
   Log ABORT;
   // Send abort message to the accepted prewrites or
   // Do nothing with the web services (the web services that
   // accepted prewrites will abort by timeout).
   End transaction;
}

//---------------------------------------------------------------------------
//Phase-2 - Coordinator/Client reaches an intended decision and implements it
//          before the earliest deadline;
Log the INTENDED DECISION for all web services;
While there is web service to be resolved {
   Try{
      // specific decision for each web service can be COMMIT or ABORT;
      Send the decision message to and receive OK or TIMEOUT message from
       the web service;
      Log OK or TIMEOUT for the web service;
   } Catch (Connection Exception Or Server Exception){
      Log the caught exception;
      Try later;
   }
}
Log IMPLEMENTED DECISION; // no further action is required
```

- CANCEL: The prewrite autonomous cancel due a timeout or an exception.
- COMMIT or ABORT: The prewrite commit or abort, when it receives the coordinator intended decision.

The Web service events are shown in Figure 5 on the Web service side represented by small circles. The TS2PC4RS protocol detailing the actions executed by the Web services in phases 1 and 2 is described in Listing 4.2.

In this paper, we investigate procedures that allow the TS2PC4RS to deal with failures.

To support the fault recovery process, the access to *remote recovery information* can be used if necessary. The faulty Web service in its restart procedure can ask the coordinator about the transaction outcome. The access to *remote recovery information* is explained through the example scenarios in Sections 4.2 and 4.3.

Listing 4.2 TS2PC4RS Protocol for the agent (web service).

```
// Phase-1 - Agent/Web Service processes prewrite message;
Receive the prewrite request;
If the prewrite can be accepted according to
 the web service business rules then{
   Process the prewrite;
   Log READY;
   Activate timeout; // set the deadline to coordinator
   Try{
      Send READY message; //send the deadline
   }Catch (Connection Exception or Client Exception){// The phase-2 in not necessary!
      Cancel the corresponding prewrite;
      Log CANCEL DUE EXCEPTION;
   }
}Else{
   Log NOT READY;
   Try{
      Send not-ready message;
   }Catch (Connection Exception or Client Exception){// The phase-2 in not necessary!
      //  Do nothing (There is no prewrite to cancel)
   }
}

// Waits for the client's decision and time is running...

//-----------------------------------------------------------------------------
// Phase-2 - Agent/Web Service:
// Two threads:

// Thread 1
// Condition to run:Timeout expired
Cancel the corresponding prewrite;
Log CANCEL DUE TIMEOUT;
Set TimedOut to TRUE;

// Thread 2:
// Condition to run: receive the INTENDED DECISION from the coordinator
If TimedOut then{
   Try{
      Send TIMEOUT message to Coordinator;
   }
   Catch (Connection Exception or Client Exception){
      // Do nothing. The coordination will retry.
   }
}Else{
   Try{
      Log DECISION (ABORT or COMMIT);
      Execute the corresponding procedure;
      Send OK to Coordinator;
   }Catch (Connection Exception or Client Exception){
      // Do nothing. The coordination will retry.
   }
}
```

The purchase example in Section 4.1 introduces the timeout mechanism used in the recovery procedures proposed in this paper. In Section 4.2, the scenarios considering coordinator and Web service faults are described. Section 4.3 describes the scenarios where communications problems must be dealt with when using TS2PC4RS.

4.1. Reliability in purchase-of-tickets example

The purchase-of-tickets example described in Section 2.2 and originally proposed in our previous works [11, 14] is used to facilitate the understanding of the issues about reliability.

To introduce the timeout mechanism in the example, let us consider the scenario with just one client that has succeeded in sending its prewrites, as illustrated in Figure 6a. Client 1, whose timestamp is $(50, a)$, has succeeded in sending its *PW* message to book 200 tickets. So the *LPW* of both servers has one entry representing the prewrites for client 1.

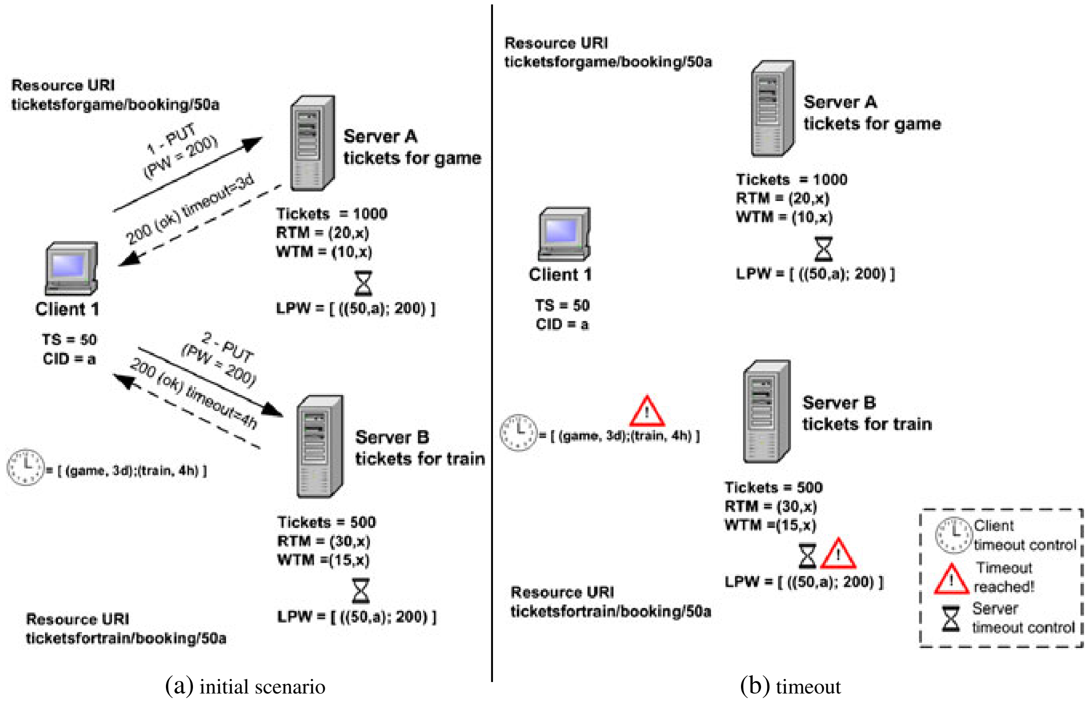


Figure 6. (a) Client 1 succeeds in sending its prewrites. (b) Server B reaches and processes timeout.

Each service returns additional information, the timeout, that the service will wait for a message from the client committing or aborting the accepted prewrite. In the example, server A declares a timeout of 3 days (timeout = 3d in Figure 6a), and server B declares a timeout of 4 h (timeout = 4h in Figure 6a).

In Figure 6a, the symbol used to represent service timeout control is an hourglass, and the symbol used to represent the client timeout control is a clock. When a timeout is reached, the service is authorized to cancel the associated prewrite. Thus, the client monitors all the timeouts informed by the services to commit or abort the operations properly.

Starting from the described scenario and assuming that client 1 does not make a decision in 4 h, server B reaches its timeout. So server B can cancel the client 1 prewrite. Figure 6b shows the occurrence of timeout in server B. The symbol with an exclamation mark in Figure 6b identifies the timeout occurrence.

Client 1, on the basis of its own control of the prewrite timeouts, autonomously processes the server B timeout and gives up the train tickets for server B. Client 1 must exclude the server B accepted prewrite from the transaction. However, if client 1 sends a commit message to server B after the deadline (server B is timed out), server B returns a timeout message to client 1, as described in the TS2PC4RS protocol of Listing 4.2 and illustrated in Figure 7.

At this point, client 1 can, for example, find another company to buy train tickets from, find another way to go to the game, send another prewrite request to server B, or decide to buy only the tickets for the game. If client 1 chooses to commit the purchase of game tickets, they just send a commit message to server A, as shown in Figure 7.

4.2. *Site failure scenarios*

The site failures are coordinator-client failures and RESTful service failures. The recovery procedures for client and Web service failures are described for each situation of the logs described in the TS2PC4RS protocol for coordinator (Listing 4.1) and service (Listing 4.2).

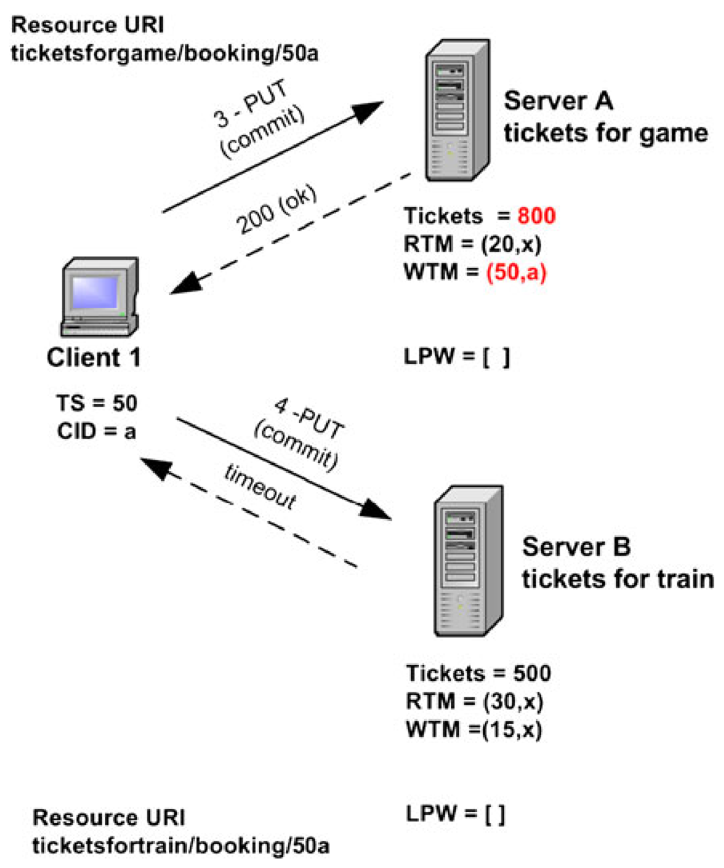


Figure 7. Client 1 decides to commit only the purchase of game tickets.

Figure 5 illustrates the interactions between coordinator/client and Web services/agents, identifying the main log events that mark the progress of the TS2PC4RS protocol (represented by small circles).

4.2.1. Coordinator crashes before GLOBAL-PREWRITE. As in the first scenario, we assume that the coordinator-client fails after logging the GLOBAL-BEGIN, but before recording the GLOBAL-PREWRITE. In this case, the following procedures are executed.

1. All the Web services that accepted the prewrites answering READY and that did not reach the timeout wait for the recovery of the coordinator.

   The restart procedure for the coordinator resumes the protocol from the beginning, reading the identity of Web services from the GLOBAL-BEGIN record in the log and sending the prewrite requests to them again. If the coordinator has logged some PREWRITEs, this information can be used to identify the Web services that already answered ready, excluding them from the restart procedure. In this case, the corresponding timeouts must be computed considering the period that the coordinator failed.

   Each Web service that still has the prewrite in *LPW* must recognize that the prewrite message is a repetition. The Web service replies with the remaining timeout to the coordinator.
2. The Web services that reach their timeout cancel the accepted prewrites. As the coordinator does not make a decision, no service receives the coordinator commit or abort, so the Web services can cancel without corrupting the transaction progress. The coordinator restarts as described in the previous item (1). Each Web service that cancels the accepted prewrite processes the re-sent prewrite as a new one.
3. If the coordinator logs ABORT, the restart procedure is not necessary. The coordinator does not need to re-send any message. The Web services cancel by timeout.

4.2.2. Coordinator crashes after the GLOBAL-PREWRITE and before the INTENDED DECISION. After the coordinator-client logs the GLOBAL-PREWRITE, indicating that all Web services returned the ready message, the coordinator can take some time to reach a decision. The period is demanded by the application business rules. The period necessary for the coordinator to take a decision must be short enough to commit the Web services before the earliest timeout.

In this situation, in the restart procedure, the coordinator resumes the protocol from the GLOBAL-PREWRITE, restarting the process to reach the intended decision as described in Listing 4.1.

4.2.3. Service crashes before recording READY. Following the sequence of TS2PC4RS protocol, the Web service fails before recording READY (or NOT-READY) in the log. There are two situations: the service crashes before receiving the prewrite and the services crash after receiving the prewrite and before logging READY.

In the first situation, the coordinator catches the corresponding exception and aborts the transaction. The service does not keep any information on the prewrite. On restart, the service proceeds without the prewrite.

In the second situation, the prewrite may be processed and inserted in *LPW* or not. However, because the *LPW* is not saved in the log because of the crash, the prewrite is lost. On restart, the service proceeds without the prewrite. Therefore, the consistency between the coordinator and service is kept in both situations.

The coordinator catches a service exception when the service starts to process the prewrite but crashes before finishing and before recording READY (or NOT-READY) in the log. The coordinator catches an exception if the service is not operational before the coordinator sends the prewrite.

As the condition (c1.) states that the coordinator must receive ready from all the services to start the TS2PC4RS phase 2, the coordinator approach is to ABORT the transaction. The coordinator can choose to send an abort message to the accepted prewrites, despite the Web services that accepted the prewrite cancel due to a timeout. See the catch clause for phase 1 in Listing 4.1. The coordinator does not need to start phase 2.

4.2.4. Service crashes after recording READY in the log. This subsection describes the failure of service after recording READY in the log. In this situation, we assume that the corresponding prewrite is in the *LPW*.

The operational Web services proceed to obey the coordinator. They correctly terminate the transaction (commit or abort), after receiving the coordinator intended decision in the TS2PC4RS phase 2.

The coordinator catches a service exception when it tries to send the intended decision to the failed Web service and records this fact in the log. The coordinator tries to send the decision until the Web service successfully recovers.

The failed Web service, through the restart procedure, asks the coordinator about the outcome of the transaction and performs the corresponding commit or abort.

When a failure occurs after the log is recorded and before the READY message is sent, the coordinator and all other operational Web services behave as described in Section 4.2.3—the transaction is aborted. The failed Web service performs the restart as described earlier.

Considering the initial scenario in Figure 8, where both clients succeed in sending their prewrites (200 tickets for client 1 and 300 tickets for client 2), a scenario in which the server B fails is described.

Server B crashes after logging READY, so afterwards, it accepts the prewrites. Figure 9a illustrates client 2, timestamp $(40, b)$, committing its transaction. Server A commits the corresponding prewrite, updating its *WTM*, *LPW*, and the available quantity of tickets. On the other hand, client 2 receives a server exception when it tries to send the commit message to server B. Then, client 2 logs the exception and keeps trying to send the decision to server B. Server B, in its restart procedure, resets the corresponding timeouts and asks client 2 about the decision. Client 2 resets the timeout.

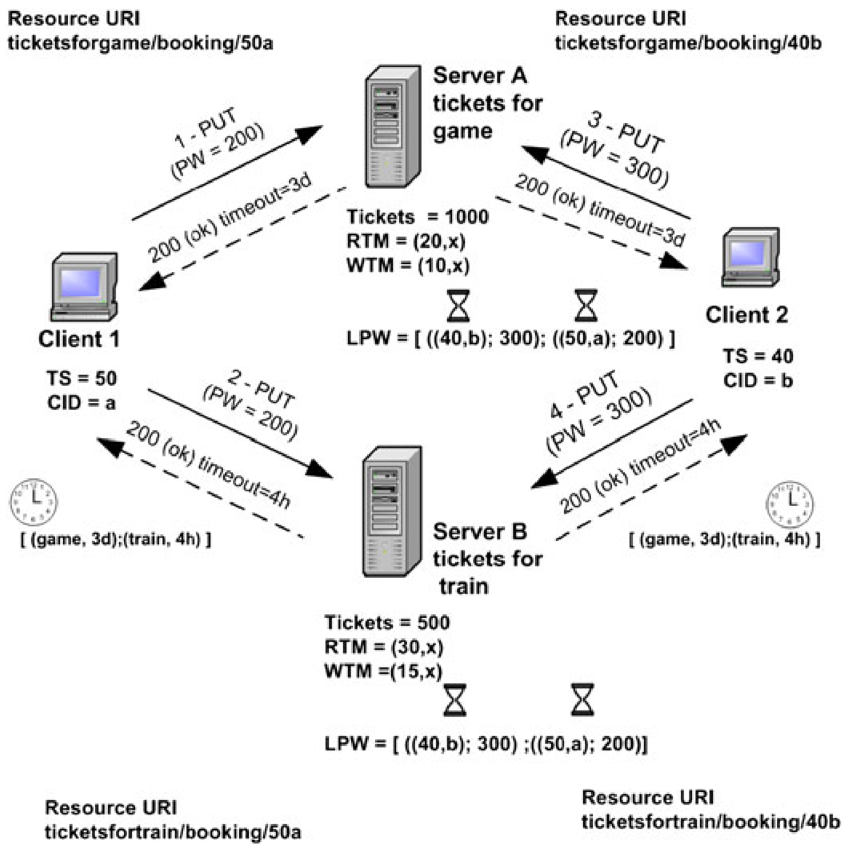
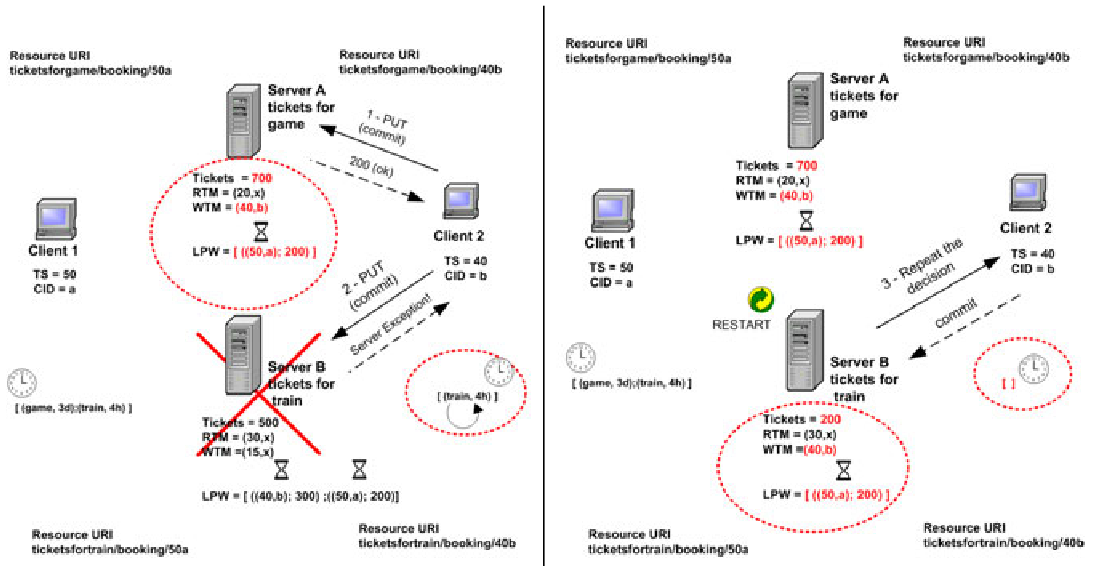


Figure 8. Clients 1 and 2 have succeed in sending their prewrites.



(a) Server crash after READY

(b) Server restart procedure

Figure 9. (a) Server B crashes after logging ready. Client 2 tries to commit but catches an exception because Server B is not operational. (b) Server B restarts and asks client about the decision.

Figure 9b shows server B's message exchange in the restart procedure. At restart, server B asks client 2 about the decision, and it accomplishes client 2's decision, in this case, committing the corresponding prewrite, updating its *WTM*, *LPW* and the available quantity of tickets. At this point, client 2 successfully commits its transaction with timestamp $(40, b)$ and logs the IMPLEMENTED DECISION.

Server B also asks client 1 about the decision. However, as client 1 did not send its decision, server B rebuilds the client 1 prewrites in the *LPW* and restarts the corresponding timeout.

4.2.5. Coordinator crashes after logging the INTENDED DECISION. Following the coordinator protocol in Listing 4.1, we analyze a coordinator failure after logging the INTENDED DECISION, but before logging the IMPLEMENTED DECISION (in TS2PC4RS phase 2). In this case, the following procedures are executed.

1. If the coordinator successfully recovers itself before the Web services' timeout expirations, the coordinator, at restart, sends the intended decision to the Web services again. The Web services that have not received the coordinator decision and have not reached their timeouts wait for the coordinator recovery. The Web services must recognize that the decision message is a repetition.
2. On the other hand, if the coordinator cannot recover itself before some Web services' timeout expirations, the Web services that have accepted the prewrite and not reached their timeouts stay in a state where they are uncertain about the coordinator decision.

   On the server side, as availability is desirable, it is not of interest to keep the prewrites for too long in the *LPW*, as the *WTM* updates progress too slowly and *LPW* may become too large to be effectively maintained.

   In this situation, the services that have reached their timeouts cancel the corresponding prewrites and execute the TS2PC4RS abort procedure [11]. The service logs the cancellation due to the timeout. See the TS2PC4RS protocol for the Web service described in phase 2 (thread 1) of Listing 4.2.

   When the coordinator finally recovers from the failure, it sends the intended decision to the Web services again. The Web services that have canceled the prewrites because of time-out return a TIMEOUT message to the coordinator. The coordinator logs the Web service TIMEOUT message, and after receiving the responses (OK or TIMEOUT) from all the Web services, the coordinator logs the IMPLEMENTED DECISION.

   If the coordinator has logged some responses (OK or TIMEOUT) before it crashes, this information can be used to identify the Web services that have already replied, excluding them from the restart procedure. The coordinator does not need to re-send the decision to the Web services that have logged answers (OK or TIMEOUT).

   We think that the situation described earlier barely occurs. However, the proposed solutions are acceptable, taking into account that the transactions are executed in a Web service domain that demands implicit service autonomy and availability.

4.2.6. Coordinator crashes after logging the IMPLEMENTED DECISION. Finally, following the TS2PC4RS protocol described in Listings 4.1 and 4.2, the scenario where the coordinator fails after recording the IMPLEMENTED DECISION is described. This is a simple situation, as the transaction is already concluded, and no action is required at restart.

Next, the failures related to communication problems are described.

4.3. Connection failures scenarios

Connection failures that may occur during a transaction execution are described in this subsection. We consider that TS2PC4RS executes over a connection-oriented transport layer protocol, and some connection problems may occur. So, in Figure 5, every request–response message uses a synchronous connection. The request messages are illustrated with solid lines, and the response messages are illustrated with broken lines.

We assume that failures of connection are detected by both client and server. See the exception handling described in the TS2PC4RS protocol of Listings 4.1 and 4.2. This way, both can execute the recovery procedures described in the following subsections.

As with the coordinator-client and Web service failures discussed in Section 4.2, the procedures for connection failure recovery are described from log information.

4.3.1. Web service cannot receive the prewrite. As in the first scenario to deal with communication failure, we assume that a prewrite message cannot be received by a Web service.

The result is the same as the situation described in Subsection 4.2.3, where a prewrite answer message (READY or NOT-READY) is not received by the coordinator because the service crashes before processing the prewrite. Thus, the coordinator adopts a similar behavior to that described in Section 4.2.3.

The coordinator catches a connection exception when it tries to send the prewrite message. In this case, the coordinator decides to abort the transaction because it does not succeed in sending a prewrite to all the Web services. The coordinator has two options: (i) send the abort message to the Web services that have accepted the prewrite; and (ii) choose not to send the abort message because the operational Web services will cancel the accepted prewrite by timeout. In either case, the transaction is aborted.

No Web-service-specific procedure is necessary in this case as the Web service that could not be reached may not even know of the communication failure. The Web service remains waiting for prewrite messages.

4.3.2. Coordinator cannot receive the prewrite answer message. The scenario where a prewrite answer message (READY or NOT-READY) from a Web service cannot be received by the coordinator-client because of communication problem is analyzed.

The coordinator-client catches the connection failure and proceeds as described in the previous item (4.3.1)—-the transaction is aborted. From the coordinator's viewpoint, this situation is similar to a Web service's failure before processing the prewrite (Section 4.2.3).

On the service side, if the service has logged READY, but it cannot send the answer-message to the coordinator, an exception is caught by the service. In this situation, the service has two options: (i) cancel the corresponding prewrite and log cancel because of exception; (ii) choose to do nothing, as the prewrite will be canceled by timeout. In both options, the Web service cancels the corresponding prewrite.

If the service has logged NOT-READY, it does not have to do anything because the prewrite was not accepted.

4.3.3. Web service cannot receive the INTENDED DECISION. This situation is similar to the one presented in Section 4.2.5. The destination Web service remains uncertain about the decision.

The coordinator logs the INTENDED DECISION and catches a communication exception when it tries to send the commit or abort message to one or more Web services. Thus, the connection failure occurs after the coordinator logs the INTENDED DECISION and before it logs the IMPLEMENTED DECISION (Figure 5).

In this case, the coordinator logs the caught exception and tries to re-send the decision later. On the service side, the connection failure occurs before the service logs the decision received from the coordinator. The Web service does not receive the decision message. Thus, it keeps waiting for the coordinator decision, until the service reaches its timeout. The services that have reached their time limits cancel the corresponding prewrite and log CANCEL DUE TIMEOUT, as described in TS2PC4RS protocol (Listing 4.2).

In this case, it is assumed that the communication channel will eventually be restored, allowing the coordinator to send the intended decision to the remaining Web services. When the coordinator logs all the Web service responses (OK or TIMEOUT), it logs the IMPLEMENTED DECISION, completing the transaction. The coordinator behavior is described in TS2PC4RS protocol of Listing 4.1 (phase 2).

4.3.4. Coordinator cannot receive the service answer message. The procedure to deal with the scenario where an answer message confirming the commit/abort or informing about the timeout cannot be received by the coordinator is described.

The Web service successfully receives the coordinator intended decision and does the following: (i) commits or aborts the accepted prewrite accordingly; or (ii) verifies that the accepted prewrite was canceled due to timeout. The failure occurs when the service sends the reply message back to the coordinator because of a connection problem.

In this situation, the coordinator catches the connection failure when it transmits the message containing the intended decision. Thus, the coordinator logs the caught exception and tries to re-send the decision later to receive the OK or TIMEOUT message from all the services. When the coordinator receives all the Web service answers, it logs the IMPLEMENTED DECISION (see phase 2 of Listing 4.1).

The Web service also catches the connection failure when answering the coordinator; however, it does not need to execute a specific procedure. As described in the TS2PC4RS protocol of Listing 4.2 (phase 2), in this case, the Web service site just waits for the coordinator retry message. The service then identifies the repetition and answers with the corresponding message (OK or TIMEOUT).

5. PRACTICAL EXPERIENCE

In this section, a practical experience is described to verify the fault tolerance approach proposed for TS2PC4RS. We show, through the experiments, that the connection and host exceptions can be caught by the coordinator and the Web services allowing them to recover from failures using the procedures described in Sections 4.2 and 4.3.

The experiments were implemented using Java version 1.6.0, the Jersey [17] version 1.0.3. Jersey is the open-source reference implementation of the JAX-RS—-Java API for RESTful Web Services (JSR 311). The Web server—servlet container—used is the Grizzly version 1.9.15b [18].

We chose two scenarios of host failure and one of connection failure to illustrate the behavior of the proposed protocol. The two host failure scenarios are described in Sections 4.2.3 and 4.2.4. The connection failure scenario is described in Section 4.3.2. Other scenarios are very close to the chosen scenarios, and they are not described here because of lack of space. They differ only in the procedures that must be executed after the detection of an exception.

The execution results of the scenario in which the service crashes before recording READY (Section 4.2.3) are summarized in Table II.

Listing 5.1 shows the logs recorded by the coordinator and by the game ticket server. The logs are numbered in the sequence in which they are recorded. The coordinator logs the GLOBAL-BEGIN within its timestamp and the selected Web service uniform resource locators. After receiving ready from the game ticket server, the coordinator logs the corresponding PREWRITE. When the coordinator tries to send the prewrite to train ticket server B, it catches an exception with a `java.net.ConnectException: Connection refused`. So the coordinator logs CANCEL DUE EXCEPTION.

The game ticket server logs READY when it accepts the prewrite, but as it does not receive a commit or abort decision in the defined time limit, the game ticket server logs cancel by timeout.

The execution results for the scenario in which the service crashes after recording READY (Section 4.2.4) are summarized in Table III. In the experiment, the train ticket server fails after it

Table II. Results for the scenario: service crashes before recording READY.

Host	Status	Verified behavior
Coordinator	Operational	Caught a `java.net.ConnectException` and aborts the transaction
Server A (Game)	Operational	Cancels the accepted prewrite by timeout
Server B (Train)	Not operational	—

Listing 5.1 Logs for the coordinator and for the game tickets server A.

```
# ---------------------- Coordinator Logs ----------------------------------
01 GLOBAL-BEGIN;50;http://hostname:9998/;http://hostname:9997/
03 PREWRITE;50;http://hostname:9998/
04 ABORT DUE EXCEPTION;50;http://hostname:9997/

# ---------------------- Tickets for Game Logs -----------------------------
02 (50;123); READY; tickets=1000; rtm=20; wtm=10; lpw=[(50;123;0);]
05 CANCEL; Timeout; (50); tickets=1000; rtm=20; wtm=10; lpw=[]
```

Table III. Results for the scenario: service crashes after recording READY.

Host	Status	Verified behavior
Coordinator	Operational	Caught a `java.net.ConnectException` and keeps retrying to send the commit message
Server A (Game)	Operational	Commits the prewrite
Server B (Train)	Operational, Not-Operational (fail) Operational (recovered)	After accepting the prewrite, server B fails, but it recovers and successfully accepts a coordinator commit retry-message.

Listing 5.2 Logs for the scenario: service crash after accepting the prewrite.

```
# ---------------------- Coordinator Logs ----------------------------------
01 GLOBAL-BEGIN;50;http://hostname:9998/;http://hostname:9997/
03 PREWRITE;50;http://hostname:9998/
05 PREWRITE;50;http://hostname:9997/
06 GLOBAL-PREWRITE;50;http://hostname:9998/;http://hostname:9997/
07 INTENDED_DECISION:commit;50;http://hostname:9998/;http://hostname:9997/
09 decisionOK;50;http://hostname:9998/
10 attempt 1/100 to resend;50
11 attempt 2/100 to resend;50
12 attempt 3/100 to resend;50
13 attempt 4/100 to resend;50
15 decisionOK;50;http://hostname:9997/
16 IMPLEMENTED_DECISION:commit;50;http://hostname:9998/:true;http://hostname:9997/:true

# ---------------------- Tickets for Game Logs -----------------------------
02 (50;123);READY; tickets=1000; rtm=20; wtm=10; lpw=[(50;123;0);]
08 COMMIT(50); tickets=877; rtm=20; wtm=50; lpw=[]

# ---------------------- Tickets for Train Logs -----------------------------
04 (50;123);READY; tickets=500; rtm=30; wtm=15; lpw=[(50;123;0);]
14 COMMIT(50); tickets=377; rtm=30; wtm=50; lpw=[]
```

has accepted the prewrite. The train ticket server is restarted in time to receive a coordinator commit retry message. So the train ticket server recovers from the recorded logs and commits the prewrite.

Listing 5.2 shows the logs recorded by the coordinator and game and train ticket servers. The logs are numbered in the sequence in which they are recorded. The coordinator logs the GLOBAL-BEGIN, one PREWRITE for each service, GLOBAL-PREWRITE, INTENDED-DECISION, and the commit confirmation (OK) received from the game server. Then, the coordinator tries to send the commit to the train ticket server; it catches a `java.net.ConnectException` and keeps trying to send the commit message. On the fourth attempt, the coordinator receives the OK from the train ticket server, and so it records the IMPLEMENTED-DECISION.

The game ticket server logs READY when it accepts the prewrite and COMMIT when it receives and processes the decision message sent by the coordinator. The train ticket server has similar logs. The difference is that the train server fails after it logs READY. However, it successfully restarts from the READY log information and records COMMIT after it receives and processes the retry decision message sent by the coordinator.

Table IV. Results for the scenario: coordinator cannot receive the prewrite answer message.

Host	Status	Verified behavior
Coordinator	Operational	Caught a `java.net.NoRouteToHostException` and aborts the transaction.
Server A (Game)	Operational	Cancels the accepted prewrite by timeout
Server B (Train)	Operational	After accepting the prewrite, server B cannot send the ready message because of a connection failure Server B caught a `java.io.IOException` and cancels the accepted prewrite.

Listing 5.3 Logs for the scenario: Coordinator cannot receive the prewrite answer message.

```
# ---------------------- Coordinator Logs ---------------------------------
01 GLOBAL-BEGIN;50;http://hostname:9998/;http://hostname:9997/
03 PREWRITE;50;http://hostname:9998/
05 ABORT DUE EXCEPTION;50;http://hostname:9997/

# ---------------------- Tickets for Game Logs -----------------------------
02 (50;123);READY; tickets=1000; rtm=20; wtm=10; lpw=[(50;123;0);]
06 CANCEL;Timeout;(50); tickets=1000; rtm=20; wtm=10; lpw=[]

# ---------------------- Tickets for Train Logs -----------------------------
04 (50;123);READY; tickets=500; rtm=30; wtm=15; lpw=[(50;123;0);]
05 CANCEL; Exception sending answer message;(50); tickets=500; rtm=30; wtm=15; lpw=[]
```

The execution results for the scenario in which the coordinator cannot receive a prewrite answer because of a connection failure (Section 4.3.2) are summarized in Table IV. In the experiment, we use two computers connected by an Ethernet network cable. One computer executes the client/coordinator, and the other executes the game and train ticket servers. When the train ticket server is processing the prewrite, we disconnect the cable to insert a connection failure. The coordinator/client caught a `java.net.NoRouteToHostException` and aborts the transaction. The train ticket server caught a `java.io.IOException` with the message ‘An existing connection was forcibly closed by the remote host’. So the train server cancels the corresponding prewrite.

Listing 5.3 shows the logs recorded by the coordinator and game and train ticket servers. The logs are numbered in the sequence in which they are recorded. Records with the same sequence number indicate that the events occur at the same time. The coordinator logs the GLOBAL-BEGIN, PREWRITE (of the game server), and ABORT DUE EXCEPTION.

The game ticket server logs READY when it accepts the prewrite, but as it does not receive a commit or abort decision in the defined time limit, the game ticket server logs CANCEL due to timeout.

The train ticket server also logs READY when it accepts the prewrite. However, the train server logs CANCEL due to a connection exception when it tries to send the ready message to the coordinator, canceling the just-accepted prewrite.

6. FAULT TOLERANCE FOR THE EXTENDED TIMESTAMP-BASED TWO-PHASE COMMIT PROTOCOL FOR RESTFUL SERVICES

In this section, we describe the fault-tolerant protocol for the extended TS2PC4RS [14]. As mentioned, the extended TS2PC4RS algorithm allows prewrite updates on the basis of the application business rules.

Phase 2 is the same as the TS2PC4RS protocol described in Listings 4.1 and 4.2. Phase 1 is extended to consider the update messages. Phase 1 has two sub-phases: phase 1 described in Section 4 (Listings 4.1 and 4.2) and a new sub-phase to deal with updates.

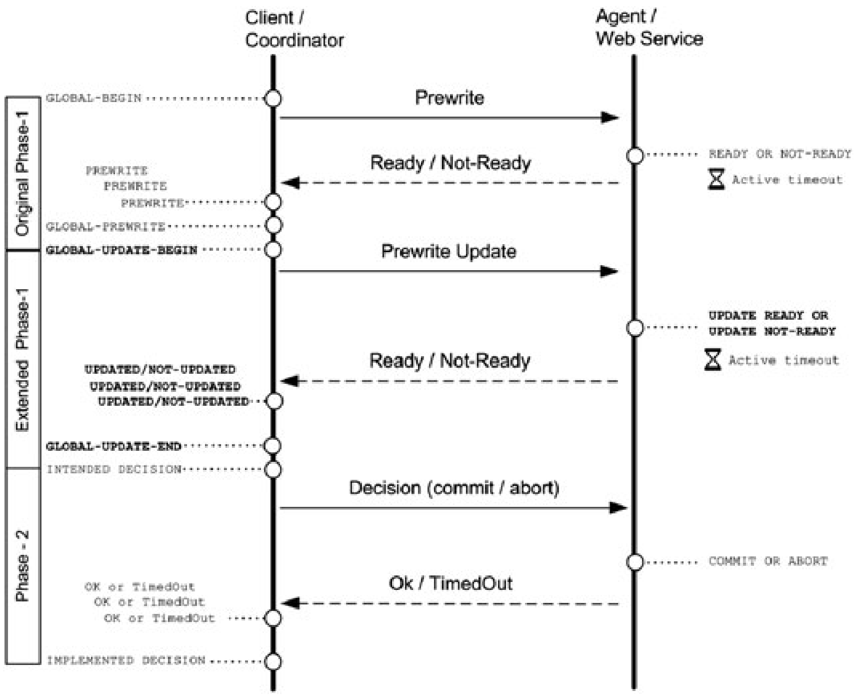


Figure 10. Exchange of messages and log records in the fault-tolerant protocol for the extended timestamp-based two-phase commit protocol for RESTful services.

The new sub-phase is called phase 1 extension and is only executed if the coordinator, after logging the GLOBAL-PREWRITE, has to update a prewrite. The phase 1 extension is optional. The two-phase adapted protocol is shown in Figure 10.

The additional events that the coordinator/client must log are described below. The new coordinator records are shown in Listing 6.1 and in Figure 10 (the additional log events are in bold).

- GLOBAL-UPDATE-BEGIN: The start of the phase 1 extension, when the coordinator selects the Web services (prewrites) to be updated.
- UPDATED: The success in updating the prewrite on the selected Web service.
- NOT-UPDATED: The failure in updating the prewrite on the selected Web service;
- GLOBAL-UPDATE-END: The end of the phase 1 extension, when the coordinator knows the successful updates.

There is only one additional event that each Web service must log, the UPDATE READY or UPDATE NOT-READY, indicating the prewrite update acceptance or rejection. The new Web service records are shown in Listing 6.2 and in Figure 10 (the additional log events are in bold).

The failure recovery procedures for *phase 1* and *phase 2* are the same as those described in the procedures for site failures (Section 4.2) and connection failures (Section 4.3). The failure recovery procedures for *phase 1 extension* are described in what follows.

On the coordinator side, the service or connection failures are detected in the phase 1 extension. The coordinator logs NOT-UPDATED, stating that the corresponding update has not been processed. The coordinator considers the previous accepted prewrite, sent before the update attempt. Thus, the intended decision must be based on the GLOBAL-PREWRITE and GLOBAL-UPDATE-END.

On the service side, when a coordinator-client or connection failure is detected during sending of the ready message in the phase 1 extension, the service cancels the corresponding update and records this fact in the log. We can note that procedures to address failures for the extended TS2PC4RS are straightforward and the prewrite updates do not significantly affect the proposed protocol.

Listing 6.1 Two-phase adapted TS2PC4RS for the coordinator. Phase-2 and Phase-1 are the same of Listings 4.1.

```
// Phase-1 - Identical to Phase-1 of the TS2PC4RS Protocol
// for the coordinator.

//-----------------------------------------------------------------------------
// Condition to start the Phase-1 Extension: The GLOBAL-PREWRITE is in the log.

// Optional Phase-1 extension to deal with the prewrite updates of the Extended
// TS2PC4RS. Coordinator/Client sends UPDATE requests to agents/web services and
// receives replies with information.

// The Coordinator-client selects the updates.
Log GLOBAL-UPDATE-BEGIN;
For each prewrite update{
     Try{
       Send update message to and receive ready (or not ready) message
        (with the same deadline) from web service;
       If ready then{
          Log UPDATED;
       }
       Else then{
          Log NOT-UPDATED;
          // Do nothing with the web services. The coordinator must
          // consider the previous accepted prewrite  (before the update attempt).
       }
    }Catch (Connection Exception Or Server Exception){
       Log NOT-UPDATED;
       // Do nothing with the web services. The coordinator must
       // consider the previous accepted prewrite (before the update attempt).
   }
}
Log GLOBAL-UPDATE-END; //  END of the Phase-1 extension.


//-----------------------------------------------------------------------------
//Phase-2 - Identical to Phase-2 of the TS2PC4RS Protocol for the coordinator.
```

7. RELATED WORK

Buys *et al.* [19] proposed a context-aware fault tolerance for SOA applications to improve approaches based on redundancy strategies. The redundancy-based fault-tolerant strategies are used as a means to avoid disruptions in services despite of the occurrence of failures in the underlying components. The redundancy-based strategies are usually static predefined and immutable; they do not necessarily result in improvement in application dependability when they are applied in highly dynamic, distributed computing systems such as Web services executing over the Internet. SOA systems exhibit highly dynamic characteristics, and changes in the operational status of Web services, in particular their availability and response time, are likely to occur frequently. In this scenario, Buys *et al.* [19] proposed a dependability strategy for supporting a redundancy management that allows the autonomous tuning of its configuration in view of changes in the operational status of any of the components involved, that is, the context.

Avizienis [20] proposed using replicas to implement fault tolerance for Web services based on the n-version programming (NVP). The NVP approach uses the independent generation of $n > 1$ functionally equivalent programs from the same initial specification. The idea is for the NVP to function as a client-transparent replication layer in which all n programs (versions or replicas) receive a copy of the user input and are orchestrated to independently perform their computations in parallel. Then, NVP uses a decision algorithm to determine a result from the individual outputs of the employed versions. Examples of decision algorithms include majority, plurality, and consensus voting [21].

Buys *et al.* [19] proposed improvements for NVP by adding a dynamic perspective through providing a context-aware fault-tolerant strategy. However, their proposal has to deal with the task to

Listing 6.2 Two-phase adapted TS2PC4RS for the web service. Phase-2 and phase-1 are the same of Listing 4.2.

```
// Phase-1 - Identical to Phase-1 of the TS2PC4RS Protocol
// for the web service.

//---------------------------------------------------------------------------
// Condition to start the Phase-1 Extension: The READY of the prewrite is
// in the log.

// Optional Phase-1 extension to deal with the prewrite updates of the
// Extended TS2PC4RS. Agents/web services process the prewrite update.

Receive the update request;
If the update can be accepted according to
 the web service business rules then{
   Process the update;
   Log UPDATE READY;
   Try{
      Send READY message; // The deadline is the same of the prewrite.
   }Catch (Connection Exception or Client Exception){
      Cancel the corresponding update; // only the update is cancelled.
      Log UPDATE CANCEL DUE EXCEPTION;
   }
}Else{
   Log UPDATE NOT READY;
   Try{
      Send not-ready message;
   }Catch (Connection Exception or Client Exception){
      // Do nothing
   }
}

//---------------------------------------------------------------------------
//Phase-2 - Identical to Phase-2 of the TS2PC4RS Protocol for the web service.
```

collect information about the context, the environment in which the Web services operate. Some examples of contextual information are the amount of redundancy (the number of versions) currently employed, the evolution of voting outcomes (which versions can be rewarded and which can be penalized), and the operational status of each of the available resource versions (or replicas), such as dependability, load, and execution time. Thus, a continuous monitoring of changes in the operational status of any replica is necessary. The monitoring demands the definition of a protocol among all the replicas used in the fault tolerance strategy. Suitable monitoring for all kind of domains in which the fault tolerance is necessary cannot be guaranteed.

In TS2PC4RS, the Web services of different providers can be used as a kind of 'replica'. However, the client knows about them and can adopt the strategy to send concurrent prewrite requests to all providers of the same resource (data item). However, in the second phase, the client must send the commit to each provider in sequence and conclude the transaction when it receives the first commit confirmation. The other prewrites for the same resource (data item) must be aborted.

In this way, different decision algorithms (for approaches such as NVP) must be defined for the first and second phases of TS2PC4RS. In the first phase, the prewrite requests can be sent concurrently for the selected replicas; however, in the second phase, the commit request must be sent in sequence for each replica. The TS2PC4RS's second phase finishes when the first commit confirmation is received by the decision algorithm from each different Web service used in the transaction. Each Web service used in the transaction must have its own set of replicas.

However, Buys *et al.* [19] did not address the use of transactions or how to apply the proposed context-aware fault tolerance strategy within a Web service transaction. A transaction usually uses multiple Web services to achieve the transaction objective. The transaction must be executed as a unique overall activity.

Zheng and Lyu [22] proposed an adaptive QoS-aware fault tolerance strategy for Web services. The idea is to improve service reliability using subjective user requirements and objective system performance. The proposed fault tolerance strategy is context aware and aims to be dynamically and automatically reconfigured to meet different user requirements and changes in the environment status. QoS-aware middleware is required to promote user participation and collaboration through the sharing of user's individually obtained QoS information of the invoked Web services.

To identify Web services with similar or identical functionalities developed independently by different organizations, Zheng and Lyu [22] used a service community to define a common terminology that must be followed by all participants. The idea is that the Web services, which are developed by different organizations, can be described in the same interface, following common terminology. Before joining the community, a Web service has to follow the interface definition requirements of the community and register with the community coordinator. The use of the service community concept imposes interfaces for the organizations' Web services, which can make the use of the proposed adaptive QoS-aware fault tolerance strategy difficult. Another drawback is that the users need to contribute their own individually collected Web service QoS information via a service community coordinator. The sharing of information imposed cannot be reached in all Web service applications as the user organization may have some policies involving security or confidential restrictions, which avoid information sharing with other users or organizations.

Zheng and Lyu's work [22] does not address the use of transactions or how to apply the proposed adaptive QoS-aware fault tolerance strategy within a Web service transaction. However, we believe that QoS-aware fault tolerance can be used jointly with our fault tolerance proposal, in a similar manner already explained earlier. For the replicas managed by the QoS-aware middleware to be used in a TS2PC4RS transaction, the first phase of TS2PC4RS can use a parallel fault tolerance strategy (sending the prewrite requests at the same time), and the TS2PC4RS second phase must use a sequential fault tolerance strategy (the next replica is invoked only if the primary Web service fails) because the commit must be executed by only one replica. It is necessary because the user/client wants to commit only once. For example, in the ticket-purchase example, the client wants to buy the amount of tickets for the game only once, despite the prewrite request (reservation) being sent to all the replicas at the same time.

There is another approach that uses replication, such as smart proxies [23], which proposes that a middleware run the client side, providing replication transparency for client applications. The middleware is responsible for selecting the best replica and invoking the selected replica provider from the client side. The middleware must adapt the replica provider interface considering the interface required by the client. The concepts of this client-side middleware can also be used with our TS2PC4RS fault tolerance approach to avoid interweaving between the functional code of the client and the control code necessary to implement the fault-tolerant TS2PC4RS.

Another approach that can contribute to implementing client-side middleware is taken by the guidelines proposed by Mendonça *et al.* [24] to help application developers in identifying a replication server selection policy that best suits a particular Web service replication scenario. In this way, the middleware that runs on the client side can transparently and automatically select the best Web service—from the perspective of a particular client application—from a collection of functionally equivalent Web service instances.

In both approaches [23, 24], it is necessary to evaluate the use of such client-side middleware when the client application assumes the role of a coordinator of a Web service transaction as is the case for TS2PC4RS.

A replication framework for SOAP-based Web services is proposed by Salas *et al.* [25]. The framework allows the deployment of a Web service in a set of sites to increase its availability. The replication is made exclusively using SOAP to interact across sites. As TS2PC4RS uses RESTful Web services, the replication framework cannot be directly applied in TS2PC4RS. Although, the proposed concepts such as the use of multicast to communicate with the replicas of a Web service can be used in a replication framework for TS2PC4RS.

In general, the frameworks based on Web service replication can be used with the TS2PC4RS fault tolerance approach proposed in this paper. However, for the replication in Web service applications using TS2PC4RS to be used, it is necessary to correctly choose the RESTful service invocation mode, that is, if the framework can concurrently invoke the replicas or not. For example, in the purchase-of-tickets scenario, if the client wants to buy tickets for a game only once, that is, the client does not want to buy the amount of tickets asked in all the replicas applied in the fault tolerance strategy, only one service must effectively process the request to sell the game tickets. In this case, despite the replicas (similar Web services) being considered to be functionally equivalent programs, they can be provided by different organizations, which have their own stock of tickets to sell. Each organization may have its own market strategies, logistics, and interests.

The TS2PC4RS fault tolerance approach can be used jointly with approaches for other areas, such as testing. Chakrabarti and Kumar [26] proposed an approach to test RESTful Web services for a large information technology organization infrastructure. The testing approach can help to discover failures originating from system faults or connection failures over the Internet [26]. So the testing proposal can be used by the developers of RESTful services that implement TS2PC4RS with our fault tolerance proposal to execute functional and non-functional tests.

The relaxed atomicity and isolation proposed by TS2PC4RS [11] is a specific extended transaction model based on timestamp and 2PC. So our proposal in [11] is different from the one proposed by Houston *et al.* [27] who have described a general-purpose event-signaling mechanism to support various extended transaction models. Despite this difference, we think that the TS2PC4RS model can be analyzed to adapt it to the general-purpose event-signaling mechanism.

The TS2PC4RS can be adapted to be used in other application domains, such as mobile database systems. To use a minimum number of wireless messages, Kummar *et al.* [28] proposed a transaction commit protocol based on timeout, that is, if the coordinator does not receive a failure message from a participant node within a predefined timeout period, then the coordinator commits the transaction. One of the drawbacks of the protocol is to find the most appropriate value of a timeout. This is not easy because it depends on a number of system variables. It is possible to adapt TS2PC4RS to use the timeout commit protocol proposed by Kummar *et al.* We can change the protocol of the TS2PC4RS timeout mechanism described through the scenarios (Sections 4.2 and 4.3) in a way that if the service does not receive the coordinator decision within a predefined timeout, the service commits the prewrite instead of canceling it. However, a detailed analysis of fault tolerance is required.

Reinke *et al.* [29] proposed a transaction model for sensor networks. They applied standard commit protocols such as 2PC [13] and transaction commit on timeout [28] to analyze the feasibility of the usage of the protocols in wireless sensor network domains. As the fault-tolerant TS2PC4RS is based on 2PC protocol and timestamp, we believe that some ideas of the fault-tolerant TS2PC4RS can be used in the wireless sensor networks. However, some investigation is also required to evaluate its applicability.

The Paxos algorithm [30] is by definition a fault-tolerant distributed system. In this way, Paxos algorithm does not need extensions to deal with failures. Paxos uses different roles. The agents are responsible to execute client, acceptor, proposer, learner, and leader roles [30]. The fault-tolerant TS2PC4RS uses two roles: client-coordinator and Web services-agents. Paxos assumes the use of a asynchronous communication [30] to exchange messages, and TS2PC4RS assumes a synchronous communication as it uses Web services available through the HTTP over a transport layer protocol/ Internet Protocol network protocol. Despite the differences, both approaches use durable logs to record main events for recovery.

8. ANALYSIS AND CONCLUSIONS

As mentioned, TS2PC4RS [11] provides concurrency control for applications that involve various RESTful services based on the timestamp and 2PC protocol. An extension to TS2PC4RS was proposed to improve business rule support through the usage of prewrite updates [14]. However, neither TS2PC4RS nor its extension addresses failures that may occur during a distributed transaction executing over a Web service infrastructure. In this paper, we present failure recovery procedures to improve the reliability of TS2PC4RS and its extension. The reliability provided by the fault tolerance mechanism is essential to the successful implementation of Web service transactions, which can take a long time and are subject to failures.

The timeout mechanism is used in the management of the list of prewrites (*LPW*) whose transactions can take too long. In this situation, the updates of *WTM* may progress slowly and *LPW* may get large. The timeout mechanism can be used to address this problem. If the coordinator-client neither commits nor aborts the transaction within a period, then the Web services cancel the accepted prewrites by timeout. It is an important feature, as it allows the clearing of the prewrites that are not valid.

The reliability leveraged by the fault tolerance mechanism improves business rule support on the server side. The timeout mechanism allows the providers to maintain a level of concurrent accesses. The servers are able to concurrently serve a high number of clients through better control of the quantity of prewrites being processed.

It is worthy to note some of the best practices recommended for the fault tolerance mechanism. The coordinator/client has to follow the time limit imposed by the Web services to reach a decision in due time. The coordinator computes the earliest timeout from the Web services and tries to send the intended decision before the earliest timeout.

However, if the coordinator cannot reach an intended decision before a Web service timeout, this should be detected by the coordinator's own timeout control, and the coordinator then disregards such a service from the transaction in progress and considers this fact in the decision. If the coordinator does not detect the timeout and sends a commit message to a Web service that has already reached its time limit, the service replies with a timeout message, as described in the TS2PC4RS protocol of Listing 4.2.

On the service side, it is recommended that the effective timeout implemented by the Web services be longer than the timeout returned to the coordinator-client. In this way, if the coordinator encounters difficulties related to connection or site failures, it has some additional time to execute the recovery procedures, as described in Sections 4.2 and 4.3.

A best practice in using the timeouts is illustrated through a brief example: a Web service informs a client of a timeout of 4 h. The client must decide within 4 h; otherwise, the service cancels the client's request. However, in fact, the Web service waits for 5 h, although it states a timeout of 4 h. Therefore, the service waits longer than the agreed period. Using this recommendation, we can mitigate the risk of an unexpected failure causing a negative impact for the client expectation.

We can note that relaxed atomicity and isolation of TS2PC4RS and durability are dealt in the proposed protocol. The proposed fault-tolerant protocol preserves the relaxed atomicity and isolation of TS2PC4RS, and durability is achieved by the use of logs.

The scenarios in which TS2PC4RS must deal with failures that emerge during a transaction execution have been described. Host and connection failures are addressed by the proposed fault-tolerant protocol. The log records are the milestones that allow the execution of recovery procedures. So the main events of TS2PC4RS must be recorded in a stable storage system to allow effective recovery.

The use of TS2PC4RS needs some form of contract, which might consider the necessary conditions to make the interactions between clients and RESTful services clear and unambiguous [11]. Thus, services and client behaviors in the event of failures must also be stated in contracts. The deadlines due to a timeout mechanism must likewise be described in the contract, stating that the service is canceled if the timeout is reached. The coordinator-client must consider the risks of executing a transaction in the event of site and connection failures.

In general, RESTful services are well suited for basic, tactical, and *ad hoc* integration over the Web, and SOAP-based services are preferred for professional enterprise application integration scenarios with a longer life span and advanced non-functional requirements, such as transactions, security, and reliability [31]. In spite of such suggestions for use, both approaches have advantages and disadvantages, so it is up to the developer to make the decision on which approach is more suitable for each particular case. In this way, although TS2PC4RS was proposed to work with RESTful services, it can also be implemented using the SOAP-based Web services. The exposed SOAP services must obey some restrictions. The fundamental restriction is that the Web services must be exposed using a standard interface with at least three operations: create, update, and retrieve. So, create and update are used to support the prewrite and write TS2PC4RS operations, and retrieve is used to support the read TS2PC4RS operation. The Web services must implement the TS2PC4RS algorithm to correctly deal with the variables *RTM*, *WTM*, and *LPW*. The REST architectural style better fits in TS2PC4RS, as REST has a uniform interface as one of its architectural restrictions. This restriction is usually implemented using the HTTP methods: PUT, POST, GET, DELETE. So the REST uniform interface restriction maps directly to the interface needs of TS2PC4RS.

The comparison between ‘big’ Web services (WS-*) and RESTful Web services described in [31] can help the objective assessment of the two integration styles to select the one that best fits the application needs in the implementation of TS2PC4RS.

The fault-tolerant TS2PC4RS can be used in other approaches, such as distributed simulation implemented using RESTful services [32], which propose general middleware to expose services as URIs to the external world using the REST architectural style to make the middleware independent of the simulation formalism or a specific simulation engine [32]. To use the fault-tolerant TS2PC4RS, the middleware architecture must be analyzed to verify where the use of Web service transactions is necessary. TS2PC4RS [11] can contribute better in approaches that need the execution of Web service transactions and not in isolated Web services.

Currently, the fault-tolerant TS2PC4RS has the following limitations: (i) the definition of an appropriate value for the Web service timeout. This can be difficult as it should consider the communication delay, processing power of client hosts and service hosts, and the workload on the client’s side. (ii) The proposed procedures address single failures only. The result of a scenario in which multiple host and connections failures occur is unpredictable. (iii) The log records of TS2PC4RS must be recorded in a stable storage to allow an effective recovery. The fault-tolerant TS2PC4RS works only if no log information is lost. The lost of log information is catastrophic.

As future work, we consider two alternatives, which are both based on contracts. The first is the drawing up of a specification language that describes the contracts that implement the business rules. The second is to include security requirements in the contracts. The idea is that both business rules and security requirements be mapped onto components that implement the fault-tolerant TS2PC4RS.

REFERENCES

1. Fielding R. Architectural styles and the design of network-based software architectures. *PhD in Computer Science*, University of California, Irvine, 2000.
2. O’Reilly T. REST vs. SOAP at Amazon April 2003. Available from: http://www.oreillynet.com/pub/wlg/3005 [last accessed November 2010].
3. Vinoski S. Serendipitous reuse. *IEEE Internet Computing* 2008; **12**(1):84–87. Available from: http://dx.doi.org/10.1109/MIC.2008.20 [last accessed March 2011].
4. World Wide Web Consortium. Simple object access protocol (SOAP) 1.1. *Technical Report*, May 2000. Available from: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/ [last accessed January 2011].
5. World Wide Web Consortium. Web services description language (WSDL) 1.1, 2001. Available from: http://www.w3.org/TR/2001/NOTE-wsdl-20010315 [last accessed January 2011].
6. Organization for the Advancement of Structured Information Standards. Oasis Web services security (WSS) TC, Feb 2006. Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss [last accessed February 2011].
7. Organization for the Advancement of Structured Information Standards. Oasis Web services reliable messaging (WSRM) TC. *Technical Report*, Nov 2004. Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm [last accessed February 2011].
8. Organization for the Advancement of Structured Information Standards. Oasis Web services transaction (WS-TX) TC, Feb 2009. Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx [last accessed January 2011].
9. Organization for the Advancement of Structured Information Standards. Web services coordination version 1.2. *Technical Report*, Feb 2009. Available from: http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html [last accessed January 2011].
10. Costello RL. Building Web services the REST way. *Technical Report 200-?*. Available from: http://www.xfront.com/REST-Web-Services.html [last accessed November 2010].
11. Maciel LAHS, Hirata CM. A timestamp-based two phase commit protocol for Web services using REST architectural style. *Journal of Web Engineering* September 2010; **9**(3):266–282.
12. Lamport L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 1978; **21**(7):558–565.
13. Ceri S, Pelagatti G. *Distributed Databases, Principles and Systems*. McGraw-Hill: New York, 1985.

14. Maciel LAHS, Hirata CM. Extending timestamp-based two phase commit protocol for RESTful services to meet business rules. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), Taichung, Taiwan, March 21–24, 2011*, Chu WC, Wong WE, Palakal MJ, Hung CC (eds). ACM, 2011; 778–785, DOI: 10.1145/1982185.1982354.
15. Wang T, Vonk J, Kratz B, Grefen PWPJ. A survey on the history of transaction management: from flat to grid transactions. *Distributed and Parallel Databases* 2008; **23**(3):235–270.
16. The Business Rules Group. Defining business rules. *Technical Report*, 2000. Available from: http://www.businessrulesgroup.org [last accessed July 2011].
17. Javanet.Jersey.Java technology related community website, 2012. Available from: jersey.java.net/ [last accessed March 2012].
18. Javanet.Grizzly.Java technology related community website, 2012. Available from: http://grizzly.java.net/ [last accessed March 2012].
19. Buys J, De Florio V, Blondia C. Towards context-aware adaptive fault tolerance in SOA applications. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based System,* DEBS '11. ACM: New York, NY, 2011; 63–74, DOI: 10.1145/2002259.2002271.
20. Avizienis A. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering* 1985; **11**(12):1491–1501.
21. Lorczak PR, Caglayan AK, Eckhardt DE. A theoretical investigation of generalized voters for redundant systems. In *International Symposium on Fault-Tolerant Computing (FTCS '89).* IEEE Computer Society Press: Washington, D.C., 1989; 444–453.
22. Zheng Z, Lyu M. An adaptive QoS-aware fault tolerance strategy for Web services. *Empirical Software Engineering* 2010; **15**:323–345.
23. Junior JGR, Carmo GTS, Valente MTO. Invocation of replicated Web services using smart proxies. In *Proceedings of the 12th Brazilian Symposium on Multimedia and the Web,* WebMedia '06. ACM: New York, NY, 2006; 138–147, DOI: 10.1145/1186595.1186613.
24. Mendonça NC, Silva JAF, Anido RO. Client-side selection of replicated Web services: an empirical assessment. *Journal of Systems and Software* 2008; **81**(8):1346–1363. Available from: http://www.sciencedirect.com/science/article/pii/S0164121207003068 [last accessed April 2012].
25. Salas J, Perez-Sorrosal F, Patiño Martínez M, Jiménez-Peris R. Ws-replication: a framework for highly available Web services. In *Proceedings of the 15th International Conference on World Wide Web,* WWW '06. ACM: New York, NY, 2006; 357–366, DOI: 10.1145/1135777.1135831.
26. Chakrabarti S, Kumar P. Test-the-rest: an approach to testing restful Web-services. *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World*, 2009; 302–308, DOI: 10.1109/ComputationWorld.2009.116.
27. Houston I, Little MC, Robinson I, Shrivastava SK, Wheater SM. The Corba activity service framework for supporting extended transactions. *Software: Practice and Experience* 2003; **33**(4):351–373.
28. Kumar V, Prabhu N, Dunham M, Seydim A. TCOT—a timeout-based mobile transaction commitment protocol. *IEEE Transactions on Computers* 2002; **51**(10):1212–1218.
29. Reinke C, Hoeller N, Neumann J, Groppe S, Linnemann V, Lipphardt M. Integrating standardized transaction protocols in service-oriented wireless sensor networks. In *Proceedings of the 2009 ACM Symposium on Applied Computing,* SAC '09. ACM: New York, NY, 2009; 2202–2203, DOI: 10.1145/1529282.1529768.
30. Lamport. Paxos made simple. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)* 2001; **32**(4):51–58.
31. Pautasso C, Zimmermann O, Leymann F. Restful Web services vs. big Web services: making the right architectural decision. *Proceedings of the International World Wide Web Conference:* [*s.l.:s.n.*], 2008; 805–814. Available from: http://www2008.org/ [last accessed January 2011].
32. Al-Zoubi K, Wainer GA. Performing distributed simulation with RESTful Web-services. In *Winter Simulation Conference*, Dunkin A, Ingalls RG, Yücesan E, Rossetti MD, Hill R, Johansson B (eds). WSC: Austin, 2009; 1323–1334. Available from: http://dx.doi.org/10.1109/WSC.2009.5429650 [last accessed May 2012].