A tentative commit protocol for composite web services

Muhammad Younas\textsuperscript{a,*}, Kuo-Ming Chao\textsuperscript{b}

\textsuperscript{a} Department of Computing, Oxford Brookes University, Oxford, UK
\textsuperscript{b} Department of Computer and Network Systems, Coventry University, Coventry, UK

Received 27 February 2005; received in revised form 31 July 2005

Abstract

Composite web services provide promising prospects for conducting cross-organizational business transactions. Such transactions are generally complex, require longer processing time, and manipulate financially critical data. It is therefore crucial to ensure stronger reliability, higher throughput and enhanced performance of transactions. In order to meet these requirements, this paper proposes a new commit protocol for managing transactions in composite web services. Specifically, it aims to improve the performance by reducing network delays and the processing time of transactions. The proposed protocol is based on the concept of tentative commit that allows transactions to tentatively commit on the shared data of web services. The tentative commit protocol avoids resource blocking thus improving performance. The proposed protocol is tested through various simulation experiments. The outcomes of these experiments show that the proposed protocol outperforms existing protocols in terms of transaction performance.

© 2006 Elsevier Inc. All rights reserved.

Keywords: Web services transactions; Commit protocol; Composite web services; Performance; Tentative commit

1. Introduction

A web service is a self-contained software program which can dynamically be discovered and invoked across the Web. Web services are generally developed using software technologies such as WSDL, SOAP, XML, UDDI [21]. Web services are increasingly used to automatically perform a variety of business tasks. They have become predominant means for establishing business-to-business (B2B) collaboration between autonomous organizations. Such collaboration is dynamically established through the composition of component web services that represent organizations' business activities. That is, component web services are composed to develop new composite services that are capable of performing enhanced and cross-organizational B2B activities. To fully realize the benefits of composite web services, various issues need to be examined. These include, for example, security, trust, reliability and efficiency [1–3].

The work presented in this paper concerns the performance of transactions and the commit protocols developed for conducting such transactions in composite web services. Commit protocols have traditionally been employed to enforce transaction reliability, concurrency and consistency in database applications [19]. Due to their intrinsic benefits

* Corresponding author.

E-mail addresses: m.younas@brookes.ac.uk (M. Younas), k.chao@coventry.ac.uk (K.-M. Chao).

022-0000/S – see front matter © 2006 Elsevier Inc. All rights reserved.
they are also employed in (composite) web services so as to provide them with enhanced efficiency, reliability and
data consistency. For instance, component web services such as flight, hotel, and car services can dynamically be
composed into a composite web service while maintaining their reliability and consistency even in the presence of
system and communication failures.

Various frameworks and protocols have been developed for managing transactions in (composite) web services.
These include WS-Transactions [9], OASIS Business Transaction Protocol (BTP) [14], Business Transaction Framework [4], and Tentative Hold Protocol (THP) [16]. These are mainly based on the database transaction models such as
ACID and extended transaction models. ACID (atomicity, consistency, isolation, durability) are standard properties
for classical transactions. Atomicity is based on “all or nothing” policy stating that a transaction must either appear to
execute in its entirety or not at all. Consistency demands that a transaction must maintain data consistency. Isolation
requires that an active transaction cannot expose its intermediate results to other concurrent transactions. Durability
means that the effects of the committed transactions must be made permanent in persistent storages. ACID proper-
ties are implemented using various commit protocols such as two-phase commit (2PC) protocol and its variants, e.g.,
presumed abort (PA), and presumed commit (PC) protocols [7,12,19]. Though ACID model and 2PC protocols are
useful in ensuring data consistency and correctness of transactions they result in serious performance problems. This
is due to the fact that ACID properties are based on the strict isolation policy which requires that a transaction cannot
share its data with other transactions until it is completed. This strictness of isolation results in poor performance, as
transactions have to wait for each other (locking shared resources) until they are completed.

ACID properties are useful for those web services which demand strict atomicity and consistency. However,
they are inappropriate for long running tasks such as business activities. In order to deal with ACID-related issues,
various extended transaction models have been adapted for web services [4,6,9]. For instance, WS-Transaction spec-
ification [9] uses extended transaction models for its business activities. Similarly, OASIS BTP uses an extended
transaction model for long running tasks called cohesions. Extended transaction models mainly relax the strict atom-
icity and isolation policy of ACID properties such that intermediate results of active transactions are visible to other
transactions and component transactions can unilaterally commit irrespective of the commitment of their sibling trans-
actions. Existing work is mainly focused on the specification of web services transactions and their reliability aspects.
In this paper we investigate the performance implications of the composite web services transactions. We propose a
new protocol called Tentative Commit Protocol (TCP) that aims to achieve improved performance by reducing the
network message delay and the processing time of transactions in composite web services.

The paper is structured as follows. Section 2 describes a requirement analysis and reviews related work on web ser-
dices transactions. Section 3 describes the underlying transaction model of the proposed protocol. Section 4 presents
the proposed protocol. Section 5 illustrates the evaluation of the protocol and discusses the experimental results.
Section 6 concludes the paper.

2. Web services transactions

In this section we present a requirement analysis and examine current research efforts on (composite) web services
transactions.

2.1. Requirement analysis

In order to analyze the requirements of composite web services transactions, we consider a scenario for making
travel arrangements for the upcoming Football World Cup 2006. It involves a number of web-based autonomous
businesses that provide specialized services such as flight, car rental, taxi hire, hotel, and bed & breakfast (B&B)
services. Such services can be composed into a composite web service transaction in order to provide a reliable and
integrated service which can be used by users to buy match tickets, book flights, reserve accommodation, arrange
local transportation, and make payments for these services. This scenario provides an adequate context for express-
ing the requirements of composite web services transactions. Such transactions are generally of long duration and
non-prescriptive as they span the boundaries of autonomous businesses which are distributed across the globe. It is
therefore difficult to determine in advance the execution time of such transactions. Further, the processing of these
transactions may be affected by network traffic and system load. These factors severely affect the performance of
transactions and motivate our research to examine such issues and develop appropriate models and protocols.
In the following, we review current work on (composite) web services transactions in order to determine how they have addressed the performance of such transactions.

2.2. Related work

Work on transaction management in web services generally follows ACID and extended transaction models. Papazoglou [4] reports on web services transactions and proposes a Business Transaction Framework (BTF) for web services. This work outlines the requirements and characteristics of business transactions. It also analyzes other transaction related initiatives such as the Business Process Execution Language (BPEL) for Web services, Web Services Transactions (WS-Transactions), Web Services Coordination (WS-Coordination), etc. WS-Transactions [9] and WS-Coordination [10] have been developed by software vendors such as IBM, Microsoft, and BEA Systems. These approaches aim to define frameworks for providing transactional coordination for clients of services offered by multiple autonomous businesses that are based on Web services technologies. They also provide support for long-running business activities in addition to the short-lived (atomic) transactions. Little et al. [11] give a comparative analysis of web services transaction protocols. The analysis pertains to the OASIS Business Transaction Protocol (BTP) and WS-Transactions. It identifies the similarities, differences, strengths and weaknesses of these protocols. However, this analysis does not cover the performance aspects of web services protocols; nor does it consider composite web services transactions. Mikalsen et al. [8] describes a framework called WSTx (Web Services Transactions) for web services, and introduces the concept of transactional attitudes. This approach requires web service clients to declare their transactional requirements and web service providers to declare their individual transactional capabilities and semantics.

The above approaches provide support for ACID criteria, 2PC protocols and extended transaction models. In these approaches, participating web services have to agree on a particular transaction model before starting the execution. However, in composite web services it is difficult to pre-determine a transaction model.

Tai et al. [6] investigate the composition of coordinated web services by combining BPEL4WS with frameworks of WS-Coordination and WS-Transactions. This approach also introduces coordination policies using the WS-Policy framework. Potential advantages include: the definition of Web services composition model for dynamic integration of different coordination protocols and the feasibility of combing BPEL4WS with the WS-Coordination and WS-Transactions frameworks. However, this approach results in a complex middleware which appears to be less efficient. Benlakhal et al. [5] propose a transactional architecture in order to enhance reliability and availability of WS-composition. This architecture is based on a peer-to-peer paradigm that incorporates extended transaction models such as nested transactions, sagas and nested sagas. Bhiri et al. [15] propose a transaction model for web services composition, which allows the designers of the composite web services the flexibility to define acceptable levels for transaction termination. It adapts the property of acceptable termination states which relaxes the atomicity. That is, users can define different levels for the outcomes of a transaction. However, the idea of acceptable termination states is not new. This is essentially based on the traditional workflow transaction management scheme (as in [7]).

Roberts et al. [16] designs a tentative hold protocol (THP) for business transactions (a part of W3C specifications). THP is a message-oriented loosely coupled protocol that is executed prior to the execution of the actual business transaction. The working mechanism of THP is summarized as follows. A client application makes a resource reservation request (e.g., book a flight) to a client coordinator (CC), which in turn submits that request to the resource coordinator (RC). The RC, residing at the resource provider site, is responsible for reserving the resources. The RC receives the client request and performs local processing using a rules integration module (RIM) to determine whether the requested resource can be tentatively held for that client or not (RIM implements the business rules of the service providers that relates to the resource reservation). If RC can tentatively hold a resource then it sends a message to the CC telling it that a resource is tentatively held. Tentative holds are associated with information such as timeout, number of clients holding a particular resource and so on. The main feature of THP is that it allows multiple clients to tentatively hold same resource. For instance, if another client submits a request for a similar resource (e.g., book a flight) then the RC will hold the resource and send notification to that client. If one of these clients buys the resource by executing the actual business transaction then the RC will notify the remaining clients about the situation. That is, it will inform them that their tentative holds on the resource are no longer valid.
THP can be considered as a resource reservation protocol that reserves the resources required by a business transaction. It can be used with existing transaction commit protocols such as: two-phase commit protocols, the transaction Internet protocol, the business transaction protocol, and, other protocols based on extended transaction models. The advantage of THP is that it avoids resource blocking which occurs in the 2PC protocol. It also optimizes transaction throughput and resource consumption. However, such optimization increases message communication and processing overheads thereby resulting in poor performance. Park et al. [18] proposes a framework that is aimed at improving the performance of the tentative hold protocol [16]. This approaches uses 2PC protocol in conjunction with THP protocol to ensure transaction atomicity. The idea of improving the performance of THP is based on the definition of over-hold size and hold duration. Unlike the fixed duration of the original THP, the proposed framework adaptively determines the hold duration on resources in order to achieve improved performance. Though it improves performance it still incurs message and processing overheads similar to the original THP.

The transaction model proposed by Fauvet et al. [17] adapts the original THP protocol in order to tentatively make resource reservation and to avoid resource blocking. This model is aimed at ensuring atomicity in transactional composite web services. Unlike classical atomicity this model defines various levels of atomicity. That is, a transaction can still be committed if some of the component transactions are aborted. This model is similar to the one based on acceptable termination state [15]. It defines a service composition operator which takes into account the minimality and maximality constraints for ensuring flexible atomicity. This approach increases transaction throughput for composite web services. It does not provide any evaluation of the proposed system nor does it address the issue of transaction performance.

It is observed from the above discussion that existing approaches mainly focus on the reliability or throughput of web services transactions. Though it is imperative to increase the reliability and transaction throughput these affect the performance of transactions. Majority of the current approaches do not give attention to the performance of web services transactions especially in the web services composition.

3. The composite web services transaction model

The underlying model of the proposed protocol comprises a set of service providers \( SP = \{sp_1, sp_2, \ldots, sp_n\} \) that provide various services (e.g., hotel reservation, weather forecast) as web services and, a set of service consumers, \( SC = \{sc_1, sc_2, \ldots, sc_m\} \) which request a set of services \( S = \{s_1, s_2, \ldots, s_n\} \). A composite web service, cws, is composed of ‘\( m \)’ (component) services \( s_1, s_2, \ldots, s_m \) (\( 1 < m \leq n \)) provided by the service providers. For instance, a World Cup composite service can be composed of a set of component services \{Flight, Hotel, Car, B&B, Match, Payment\}.

A (composite) web service is considered as a transactional service which is characterized by transactional properties such as (relaxed) atomicity, consistency, and durability. A transactional service allows a service consumer, \( sc_i \), to acquire different services, \( s_1, s_2, \ldots, s_n \), in a single (partially) atomic transaction. That is, all the mandatory services are either acquired or none of them.

A composite web service is modeled as a composite service transaction, CST, and its component services \( s_1, s_2, \ldots, s_n \) are modeled as component service transactions \( ct_1, ct_2, \ldots, ct_n \) (\( \in \) CST). A composite service transaction, CST, is associated with transaction properties of (relaxed) semantic atomicity, consistency, and durability. Semantic atomicity allows the unilateral commit of component service transactions irrespective of the commitment of their sibling component service transactions. Unilateral commit is useful for improving the performance of composite web services which are generally of long duration. With respect to semantic atomicity, consistency of data can be semantically maintained. In the case of failure, semantic atomicity and consistency are maintained through the execution of compensating transactions. Compensating transactions undo the affects of unilaterally committed component service transactions. Durability requires that effects of a committed composite service transaction must be made permanent in data repositories of the web services. These transaction properties are based on the extended transaction models [19] which are widely adapted by the existing web (services) transactions [5,8,11–15].

Component service transactions can be characterized by the following types. A \( ct_i \) is compensatable if its effects can semantically be undone by executing a compensating transaction. It is replaceable if there is an associated alternative service transaction. A \( ct_i \) is retrievable if it can be completed after several finite executions. A \( ct_i \) is said to be a pivot if its affects cannot be semantically undone.

In the following we describe tentative commit (3.1), the states of component service transactions (3.2) and the dependency between component service transactions (3.3).
3.1. Overview of the tentative commit

A $ct_i \in CST^h$ is said to be tentatively committed ($t$-committed) if it tentatively acquires a required service. The underlying principle is that a $t$-committed $ct_i$ can still be canceled if its parent transaction, $CST^h$, is not committed or if another $ct_j \in CST^h$ commits before the actual commit of $ct_i \in CST^h$. The tentative-commit and actual commit are explained as follows:

In actual commit, $ct_i$ acquires a service and updates the required service data (e.g., books a flight). During the commit process, other component transactions cannot update the service data. Once $ct_i$ is committed, then these updates are reflected in the web service data store. Other component service transactions (e.g., $ct_j \in CST^k$) can then access the updated data of a web service.

In tentative commit multiple component transactions can simultaneously update service data at the web service level. However, updates to the original service data are not allowed until a component service transaction is actually committed. For example, two component service transactions can simultaneously update service data such as booking a same flight seat or a same hotel room. After processing, these transactions can tentatively commit but their effects are not reflected in the service data store. Figure 1 diagrammatically shows the tentative commit scenario. In it, two component service transactions $ct_i \in CST^h$ and $ct_j \in CST^k$ are shown to update the same web service data (represented as small circles). Updated service data are shown as small shaded circles. These updates are the consequence of tentative-commit. They are not reflected in the data store of a web service until the component transactions are actually committed. If $ct_i \in CST^h$ performs actual commit then the tentatively-committed $ct_j \in CST^k$ is canceled. Cancellation of $ct_j \in CST^k$ happens iff the correctness of the outcome of $ct_j$ depends on the service data modified by $ct_i$. This is to maintain the consistency of data and the integrity of a transaction. Since the actual data is modified thus the outcome of $ct_j$ may not be correct. However, if the outcomes of $ct_i$ and $ct_j$ do not affect each other then $ct_j$ is not required to be canceled as long as it meets the requirements of the service consumer/provider. It is possible that tentative commit may not be acceptable to some service providers and consumers. We assume that at most three transactions can tentatively commit on the same service data. Increasing the number of tentatively committed transactions will optimize resource blocking as more transactions can access shared service data. However, they will result in excessive aborts.

3.2. States of the component service transactions

The behavior of a component service transaction can be modeled using states and a transition model (as shown in Fig. 2). Each $ct_i$ is associated with a set of states such as initial, waiting, active, failed, aborted, committed, canceled and $t$-committed. During the execution process, a $ct_i$ can make transition from one state to another when a certain event/action is taken place. A $ct_i$ is in initial state when it has not started its execution. It is in waiting state if it has to wait for an action/event before it moves to a next state. Active state represents the processing of a $ct_i$. Failed represents that $ct_i$ has encountered a failure such as system failure or unavailability of a required service. Aborted means that $ct_i$ is aborted due to failure or it has received a request to abort. Committed means that $ct_i$ has received a request to actually commit. tentative-committed represents that $ct_i$ has received a request to tentatively commit. Figures 2(a) and 2(b), respectively, represent the state transition diagrams for the tentative commit and actual commit of a component service transaction.
3.3 Dependencies

Dependencies specify the compositional aspects of component transactions and how these transactions affect the behavior of other transactions during the process of execution of a composite service transaction. Dependencies are enforced by defining various set of rules. Rules of dependencies may vary from one application to another. Following are the example dependencies that generally exist between component service transactions.

**Begin dependency:** A Begin dependency exists between two component service transactions, $ct_1$ and $ct_2$, if the completion of $ct_1$ triggers the begin of $ct_2$. For example, a Hotel service cannot be started until Flight service is booked, i.e., if flight is unavailable then booking a hotel is unnecessary.

**Begin-on-abort dependency:** This states that $ct_2$ cannot be started unless $ct_1$ is aborted. This situation happens, for example, in the case of executing alternative services. B&B service cannot be started unless Hotel service is aborted due to cancellation or failure.

**Abort dependency:** This states that the failure or abort of $ct_1$ results in the abort or cancellation of $ct_2$. For example, if Flight and Match Ticket services are executed in parallel, and Match Ticket service is failed then it will result in the abort or cancellation of Flight service. A user may not want to go if a match ticket is unavailable.

**Compensation dependency:** This states that the failure or abort of $ct_1$ results in the compensation of $ct_2$. For example, if flight is booked but match ticket is unavailable then the flight booking needs to be compensated through a compensating transaction. This is to maintain service consistency.

**Tentative-commit dependency:** This dependency occurs between $ct_i \in CST^h$ and $ct_j \in CST^k$ when they tentatively commit on shared service data. For example, same hotel room can tentatively be booked by $ct_i$ and $ct_j$.

**Tentative-abort dependency:** This dependency occurs from $ct_j \in CST^k$ to $ct_i \in CST^h$ when the commit of $ct_j$ results in the abort of a t-committed $ct_i$. For example, if $ct_j$ and $ct_i$ have tentatively booked a same room (i.e., they are t-committed), then $ct_i$ has to abort when $ct_j$ is actually committed.

4. The tentative commit protocol

The processing of the proposed tentative commit protocol (TCP) is accomplished through the following phases: service selection, tentative commit, and the actual commit.

4.1. Service selection

In web services paradigm, it is common that several component services can provide same capability. For instance, different airline services can provide flight booking facilities (e.g., British Airways, Emirates). In order to process a composite web service transaction, appropriate component services are selected such that they meet functional as well as non-functional (or QoS) properties. QoS properties include, for example: service price, security, reliability, and efficiency [1,3,20]. Unfortunately, there are no standard criteria for web services QoS [3]. Our approach assumes that the component services of a composite service transaction are selected through existing criteria [1,3]. Given various alternatives services, it is possible to construct different plans for the execution of a composite service transaction. For example, plan 1 may include a set of component services {Flight, Hotel, Ticket, Car} while plan 2 include {Flight,
ARTICLE IN PRESS

B&B, Ticket, Car]. These plans can be ranked and selected for execution according their QoS properties. For instance, a user may prefer to book services according to plan 1. However, if plan 1 cannot be accomplished then plan 2 can be executed to provide user with the required services. Once the required service plans are ranked and selected, then the transaction is tentatively committed, as described below.

4.2. Tentative commit process

The proposed tentative commit protocol (TCP) is implemented in an architecture which comprises different components, including: the composite service transaction/application, composite service coordinator (CS-Cr), component web service coordinators (WS-Cr), web service policy module, and the component web service. CS-Cr and WS-Cr control the execution of a composite service transaction and its components service transactions. Each composite service transaction (CST) is associated with one CS-Cr and a number of WS-Cr. The Web Service Policy module implements various business rules under which a web service is processed as part of a transaction. For example, it implements the dependency rules and other business related rules. Figure 3 represents the sequence diagram of a tentative commit process for two different component service transactions, \( ct_i \in CST^h \) and \( ct_j \in CST^k \). It involves the following steps:

1. A composite service coordinator (CS-Cr) receives a transaction request, \( ct_i \) (step 1) and forwards the request to component web service coordinators (WS-Cr) for processing (step 2).
2. WS-Cr has to acquire a required web service using the WS policy module which performs various tasks (steps 3, 4, 5). It checks whether the request of \( ct_i \) conforms to the (business) rules of the service provider. The WS policy module checks the availability of the required services. It also sets the time out period for the tentative commit. That is, if a transaction is not actually committed within that period then its tentative commit is marked as invalid.
3. If a \( ct_i \) fulfills WS policy requirements then WS-Cr processes and tentatively commits \( ct_i \) (step 6). It notifies the CS-Cr of its decision (step 7) which marks \( ct_i \) as tentative-committed (step 8).
4. Steps 9–16 are taken to tentatively commit another component transaction, \( ct_j \) using the same web service as that of \( ct_i \). Details of these steps are not given as they are similar to the above steps 1–8.

Fig. 3. Tentative commit process.
5. Steps 17–20 show that $ct_j$ is actually committed. In this case, the tentatively-committed component transaction $ct_i$ has to be dealt with as follows. If the outcome of $ct_j$ does not affect $ct_i$, then its tentative commit is still valid. If $ct_i$ depends on the outcome of $ct_j$, then its tentative commit will be canceled (steps 21–23). This is to maintain the integrity of $ct_i$ (see Section 3.1). In addition, the following situations also result in the cancellation of tentative commit:

- **Cancellation by service consumer:** A service consumer issuing the transaction can cancel its tentatively-committed component service transaction.
- **Cancellation by service provider:** A service provider is authorized to cancel any outstanding tentative-commit. It will notify the concerned CS-Cr and WS-Cr about the cancellation of its decision.
- **Timeout:** When a component service transaction is tentatively-committed, there is a timeout associated with it. This timeout is generally defined by the service consumer/provider and is implemented through a WS policy module. Once this timeout has expired, the tentative commit is canceled and the respective CS-Cr and WS-Cr are notified accordingly.
- **Cancellation due to failures:** System and network failures may also result in the cancellation of tentative-commit. Such cancellation corresponds to timeout-based cancellation. That is, if a failure occurs and the timeout of the tentative-commit expires, then it will be canceled.

### 4.3. Commit process

The commit process is started after the tentative commit. It comprises the following steps:

1. CS-Cr receives decisions from all the WS-Cr about the $t$-commit of the component service transactions, $ct_i$. If all WS-Cr send $t$-commit decisions, then CS-Cr actually commits the composite service transaction, CST, and sends the commit decision to all WS-Cr.
2. Each WS-Cr actually commits its component service transaction, provided that its $t$-commit is still valid. As described above, actual commit reflects updates in the service data store.
3. Each WS-Cr then sends commit acknowledgment to CS-Cr.
4. When CS-Cr receives all the commit acknowledgments then it marks the composite service transaction as completed.
5. If any of the WS-Cr sends an abort decision and if there is an alternative component service transaction then it will be executed through the same $t$-commit stages as above (Section 4.2). Once it is $t$-committed then its WS-Cr will send $t$-commit decision to CS-Cr, which then decides about its actual commit as above.
6. If a component service transaction is not replaceable then the whole composite service transaction will be aborted. If all the component service transactions are $t$-committed then they will be canceled by sending a cancellation request as described above. In this case no compensation is required, as component service transactions have not updated the actual service data. However, if some component service transactions are actually committed then they have to be compensated in order to ensure atomicity and consistency.

### 5. Experiments and discussion

This section evaluates TCP in comparison to existing protocols which are based on tentative hold protocol (THP) [16]. THP is used with transaction protocols such as classical two-phase commit (2PC) and other protocols which are based on extended transaction models [17]. We evaluate our protocol in comparison with THP-based protocols as these are more relevant and provide appropriate comparative evaluation.

In the following we first illustrate the experimental setup and then conduct various simulation experiments to test the performance of the protocols under consideration.

#### 5.1. Simulation model

To evaluate the performance of the above protocols we have developed a simulation model based on a queuing model of web services. This model is developed using QNAP-2 simulation package [22]. In order to model the
queuing stations in simulation experiments we employ a $M/M/1/N$ queuing systems. In each such system the arrival of service transactions follows the Poisson distribution. It is assumed that the transmission time of service transactions at the service provider site is based on exponential distribution.

Each queuing station in the network is associated with a finite capacity and a single server. The buffer keeps the arriving requests of a transaction. Further, the Poisson distribution is expressed by a single parameter $\lambda$ to represent the arrival rate and the exponential distribution is expressed by $\mu$ to represent the request processing time. Let the system be in state $E_k$ at any instant $k$. Analysis of this system has been carried out based on the birth and death model [23] with coefficients $\lambda_k$ and $\mu_k$ for $k = 0, 1, \ldots, N$, denoting arrival rate and service rates, respectively. To achieve steady state distribution, we assume $\lambda_k = \lambda$ for $k = 0, 1, 2, \ldots, N$ and $\mu_k = \mu$ for $k = 0, 1, 2, \ldots, N$. Thus the probability of the system $P_k$, can be expressed as [23]:

$$P_k = P_0 \left( \frac{\lambda}{\mu} \right)^k, \quad k \leq N,$$

where

$$P_0 = \frac{1 - \lambda / \mu}{1 - (\lambda / \mu)^N + 1}.$$ 

Each queuing station models the processing of a component service transaction that executes on a web service such as Flight web service. The combination of different servers represents the processing of a composite service transaction, CST. Our simulation model is based on the case study of the World Cup composite web service (see Section 2), which comprises 7 component services including Flight, Ticket, Hotel, B&B, Car, Taxi, and Payment services.

To ensure a secure and reliable transmission of data, our simulation has been implemented based on transmission control protocol (TCP) and it does not take into consideration the message losses. A summary of the parameters used in the simulation is given in Table 1. Service$_T$ is the total service time to process a composite service transaction. The calculation of Service$_T$ involves message delay (Msg), tentative commit delay (TC$_t$), tentative hold delay (TH$_t$), actual commit delay (AC$_t$), and the number of component service transaction (N$_{ct}$).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service$_T$</td>
<td>Service time of a composite service transaction, CST</td>
</tr>
<tr>
<td>Msg</td>
<td>Delay taken in communicating a message</td>
</tr>
<tr>
<td>N$_{ct}$</td>
<td>Number of component service transactions (ε CST)</td>
</tr>
<tr>
<td>TC$_t$</td>
<td>Time taken to tentatively commit a component service transaction</td>
</tr>
<tr>
<td>AC$_t$</td>
<td>Time taken to actually commit a component service transaction</td>
</tr>
<tr>
<td>TH$_t$</td>
<td>Time taken to get tentative holds of the required service resources</td>
</tr>
</tbody>
</table>

Various experiments were carried out to compute the total service time of a transaction in the above protocols. Each experiment ran for 1,000,000 simulations time units in order to get the steady state distribution. In the simulation, a composite service transaction is submitted to a composite service coordinator (CS-Cr) which then submits its component service transactions to component web service coordinators (WS-Cr) representing different services such as flight, ticket, hotel, etc. The simulated system uses probability measures to determine the success and failure of component service transactions. As shown in Fig. 4, the system first executes a “ticket service” where there are 90% chances that the component service transaction for buying ticket is successful and 10% chances are that it will fail. That is, it sets the success probability to 0.9 and failure probability to 0.1. The successful request is forwarded to the “flight service” where again it is assumed that the commit chances for component transaction are 90% and abort chances are 10%.

The outcome from this stage is followed by two parallel activities (car renting) and (hotel booking). Same assumptions of 90% success and 10% failure rates are made for these activities. If a car is unavailable then a taxi is booked. Similarly, if a hotel room is unavailable then B&B is booked. It is assumed that a car or taxi service is optional. The transaction will still proceed even if these services fail. We also assume that the payment service will successfully complete. In the case of failure, it will be re-executed until it succeeds. Note that this scenario captures
different execution patterns (sequential and parallel) and includes different types of component service transactions such as replaceable, and compensatable. The execution order of the component service transactions may vary from one application to another.

5.2. Experiments

We conduct three experiments to evaluate the performance of the proposed TCP protocol in comparison to the protocols based on the THP protocol. These include the standard two-phase commit protocol and the protocols based on extended transaction models. In the following experiments THP-2PC represents THP-based 2PC protocols and THP-ETM represents THP-based protocol based on an extended transaction model (ETM).

Experiment 1. This experiment simulates the commit process of the composite web service transaction. It considers the execution of 5 component service transactions on 5 web services such as flight, ticket, car, hotel, and payment service. It does not consider the execution of all 7 component service transactions as it is assumed that a composite service transaction will commit without the execution of alternative service transactions. For instance, hotel accommodation and car are successfully booked. Thus it is unnecessary to execute their alternative service transactions such as B&B and taxi. The outcome of this experiment is shown in Fig. 5.

Experiment 2. This experiment is conducted to determine the total service time of a composite web service transaction by taking into account the alternative service transactions such as B&B and taxi. It therefore simulates the execution of 7 component service transactions. For instance, if hotel service is failed then the accommodation is booked at

Fig. 4. Execution of a composite service transaction.

Fig. 5. Total service time in commit case (5 component services).
Fig. 6. Total service time in commit case (7 component services).

Fig. 7. Total service time in abort case.

B&B. Similarly, if car service is unavailable then taxi is hired for local transportation. Figure 6 shows the results of this experiment.

A total service time for committing a composite service transaction is shown in Figs. 5 and 6. The commit time is computed under various arrival rates of transactions. These experiments show that the proposed TCP significantly improves the total service time as compared to the THP-2PC and THP-ETM. A contributing factor to the improvement of the service time in TCP is that it uses a smaller number of messages, and reduces waiting time and the processing time of component service transactions. THP-ETM performs better than THP-2PC as THP-ETM allows unilateral commit of the component service transactions (in the actual commit stage). Both THP-ETM and THP-2PC incurs same processing time during their first phases (i.e., THP phase—where service resources are tentatively reserved). THP-2PC incurs extra processing time and message delays as every component service transaction has to wait for the completion of their sibling transactions (in the actual commit stage).

**Experiment 3.** This experiment computes the service time in the abort scenario of a composite web service transaction. It considers the situation wherein a transaction is aborted due to failures such as system or network failures or unavailability of a requested service. This experiment simulates the situation wherein component transactions of flight, ticket, and car services are successfully executed. However, component transactions for booking accommodation (in hotel or B&B) are assumed to have failed. The service time of abort case is computed under different arrival rates of the transactions and is shown in Fig. 7. It is shown that TCP performs better than THP-2PC and THP-ETM in the abort scenario. THP-2PC incurs the extra message delay and processing time in the abort scenario. THP-ETM
performs better than THP-2PC. However, due to its reliance THP it still results in extra message delay and processing time required to tentatively reserve resources.

6. Conclusion

This paper presented a new tentative commit protocol in order to improve the performance of composite web services transactions. The tentative commit protocol allows different composite web services transactions to update shared service data. Thus it avoids resource blocking which is an inherent problem in the classical commit protocols. The effectiveness of the proposed protocol is tested through various simulation experiments. These experiments revealed that the proposed protocol improves the performance as compared to existing protocols which are based on tentative hold protocol (THP). Though THP improves the throughput by increasing the commit chances of a composite web service transaction, it results in performance degradation. The proposed TCP is also capable of enhancing the throughput due to the fact that it allows multiple composite service transactions to update shared sources of data. Our future work includes the evaluation of TCP in terms of transaction throughput.

References