Design and Analysis of an e-Transaction Protocol Tailored for OCC *

Paolo Romano, Francesco Quaglia and Bruno Ciciani
DIS, Università di Roma "La Sapienza"

Abstract

In this work we present a protocol ensuring the e-Transaction guarantee (i.e. a recently proposed end-to-end reliability guarantee) in a Web based, three-tier transactional system. The protocol does not need any coordination among the replicas of the application server, thus exhibiting negligible overhead in normal behavior. Additionally, it achieves highly efficient fail-over especially for the case of back-end database employing Optimistic Concurrency Control (OCC), namely a type of concurrency control well suited for data access performed via Web. We also present a comparative discussion with existing solutions and a quantitative analysis of the proposed protocol, which clearly quantifies its benefits, in terms of reduced user perceived latency, especially when employed in combination with OCC.

1 Introduction

The concept of "e-Transaction" (exactly-once Transaction) has been recently introduced in [5] as a reasonable form of end-to-end reliability guarantee for a three-tier transactional system. In this paper we consider the case of three-tier systems with centralized back-end database, and propose an e-Transaction protocol that introduces negligible overhead in failure free computation, and achieves highly efficient fail-over especially in case the back-end database employs Optimistic Concurrency Control (OCC). The latter aspect gives our protocol a practical relevance deriving from that OCC is expected to be more adequate than Pessimistic Concurrency Control (PCC) for data access via Web. Specifically, PCC is attractive when (i) there is non-minimal data contention (i.e. the cost of protecting data through locking is less than the cost of rolling back transactions due to data conflicts) and (ii) time intervals locks are applied are relatively short, so that data are promptly available for other transactions [14]. This is typical of environments where data access is done by an array of terminal operators, who perform operations on the database on behalf of end users. On the other hand, Web applications make every browser a data entry clerk, therefore they allow the final user to directly access personal or financial information, or specific information of interest. As such a user will likely access its own data, contention is likely to be low. In addition, the time interval data are being accessed is typically increased, as compared to what happens in case of data access performed by a terminal operator, especially in case of access through wireless devices, such as palms and WAP enabled devices, due to the reduced transmission rate on wireless channels. Therefore, OCC (which does not employ locks), looks generally more attractive. These are just some of the reasons why some recent databases and platforms specifically tailored for Web applications (e.g. Tamino XML Server, Solid FlowEngine, Adabas D and Microsoft ADO .NET) encourage the use of OCC for "long-running activities" such as when users are remotely interacting with data via Web.

Our protocol mainly relies on the use of recovery information, locally manipulated at the database side, to guarantee that the transaction associated with a client request is committed exactly one time. Manipulation of the recovery information does not require coordination among the replicas of the application server, which do even not need to know each other existence. Our proposal is therefore inherently scalable, and well suited for both local and geographic distribution of the application server replicas themselves. With respect to the latter point, we recall that geographic distribution, with a very high degree of replication of the application servers hosting the transactional logic, is representative of recent Internet infrastructures referred to as Application Delivery Networks (ADNs), such as those provided by Sandpiper, Akamai or Edgix. These infrastructures are a natural evolution of classical Content Delivery Networks (CDNs), where the edge server has not only the functionality to enhance the proximity of contents to clients, but also to enhance the proximity between clients and the application (business) logic. Hence, the fact that our protocol reveals adequate for this type of settings further contributes to its practical relevance.

Beyond providing the description of the protocol, we present a comparative discussion with existing proposals in support of reliability and a quantitative analysis highlighting advantages from our protocol in terms of user perceived latency for a wide range of system settings.

The remainder of this paper is structured as follows. In Section 2 the three-tier system model we consider is presented. Section 3 is devoted to the introduction of our e-Transaction protocol. The comparative discussion with existing proposals is provided in Section 4. Section 5 is devoted to the quantitative analysis.

2 System Model

We consider a three-tier system where processes communicate through message exchange and can fail by crashing. Communication channels between processes are reliable, therefore each message is eventually delivered unless either the sender or the receiver crashes during the transmis-

*This work was partially funded by the WEB-MINDS project supported by the Italian MIUR under the FIRB program.

sion.

Application servers have a primitive `compute`, which embeds the transactional logic for the interaction with the database. This primitive is used to model the application business logic while abstracting the implementation details, such as SQL statements, needed to perform the data manipulations requested by the client. `compute` executes the updates on the database inside a transaction that is left uncommitted, therefore the changes applied to data are not made permanent as long as the database does not decide positively on the outcome of the transaction. The result value returned by the primitive `compute` captures the output of the execution of the transactional logic at the database, which must be communicated to the client. The primitive `compute` returns, together with the output of the execution of the transactional logic, the identifier assigned by the database server to the corresponding transaction.

The system back-end consists of a database server which eventually recovers after a crash. The database server has a primitive `decide` which can be used to invoke the commitment of a pending transaction. `decide` returns commit/rollback depending on the final decision the database takes for the transaction, and any exception possibly raised during the decision phase. Also, as in conventional database technology, if the database server crashes while processing a transaction, then, upon recovery, it does not recognize that transaction as an active one. Therefore, if the `decide` primitive is invoked with an identifier associated with an unrecognized transaction in input, then the return value of this primitive is rollback.

The database offers a transaction abstraction called "testable transaction", originally presented in [4]. With this abstraction, the database stores recovery information that can be used to determine whether a given transaction has already been committed. Specifically, each transaction is associated with an identifier, which is stored within the database as a part of the transaction execution itself, together with the result of the transaction. Hence, if the identifier is stored within the database, the corresponding transaction has already been committed. As in [4], we assume the testable transaction abstraction is supported through a primitive `insert`, available at the application servers, allowing them to ask the database server to write the identifier within the database together with the result obtained by the execution of the `compute` primitive. Finally, an additional primitive `lookup` is used to retrieve the logged recovery information.

3 The Protocol

The protocol we present ensures the following two properties synthesizing the e-Transaction problem as introduced in [4, 5]: (i) The back-end database does not commit more than one transaction for each client request (*Safety* - at most once). (ii) If a client issues a request, then, unless it crashes, it eventually receives a commit outcome for the corresponding transaction, together with the result of the transaction (*Liveness* - at least once) ([1]).

[1] According to the specification of liveness guarantees as proposed in [4, 5], an e-Transaction protocol is not required to ensure liveness in the presence of client crash. This is because the e-Transaction framework deals with thin clients having no ability to maintain recovery information. This reflects a representative aspect of current Web-based systems where access to persistent storage at the client side can be (and usually is) precluded for a variety of reasons. These range from privacy and security issues (e.g. to contrast malicious and/or intrusive Web sites invasively delivering cookies) to constraints on the available hardware (e.g. in case of applications accessible through cell phones).

```
issue(request_content req){
1.    generate a new id;
2.    select an application server AS;
3.    set outcome=ROLLBACK;
4.    send Request[req,id] to AS;
5.    while (outcome is not COMMIT){
6.       await receive Outcome[outcome,res,id] or TIMEOUT;
7.       if (TIMEOUT or outcome is not COMMIT){
8.          select an application server AS;
9.          send Request[req,id,check] to AS;
10.      } /* end if */
11.   } /* end while */
12.   return [COMMIT,res];
13. }
```

Figure 1. Client Behavior.

For space constraint we cannot report the protocol correctness proof wrt to the previously stated safety and liveness properties. The interested reader can find it in [9].

3.1 Client Behavior

The pseudo-code defining the client behavior is shown in Figure 1. Within the function `issue`, the client generates an identifier associated with the request, selects one application server and sends a Request message to this server, together with the request identifier. It then waits for the reply. In case it receives commit as the outcome for the corresponding transaction, `issue` simply returns. In any other case, it means that something wrong might have occurred. Specifically: (i) Timeout expiration means that the application server and/or the database server might have crashed. (ii) Rollback outcome means instead that the database could not commit the transaction, for example because of decisions of the concurrency control mechanism. In both cases, `issue` re-selects an application server (possibly different from the last selected one) and re-sends the Request message to that application server. The Request message is actually associated with the already selected request identifier. Upon successive timeout expirations, the client keeps on re-sending the Request message (with that same identifier) until it receives the commit outcome. Each time the client re-sends the request, it associates with the Request message an additional parameter, namely $check$, which notifies to the application server that we are in the presence of a re-transmission.

3.2 Application and Database Server Behaviors

The application server behavior is shown in Figure 2. Two execution paths are possible depending on the parameters associated with the client request. If $check$ is not included in the Request message, then the application server invokes the primitive `compute` to start a transaction on the back-end database. The application server then attempts to make changes on the database permanent by invoking `TestableTransaction`. Within this function, the application server first executes `insert`, in order to store the client request identifier within the database, together with the result of the transaction. It then sends a Decide message to the database server and waits for the outcome. This

```
Application Server:
1.    result_type res;
2.    transaction_identifier tid;
3.    while(true){
4.       cobegin
5.       ::    await receive Request[req,id] from client;
6.             [res,tid]=compute(req);
7.             outcome=TestableTransaction(res,id);
8.             send Outcome[outcome,res,id] to client;
9.       ::    await receive Request[req,id,check];
10.            if ((res=lookup(id))!=nil){
11.               send Outcome[COMMIT,res,id] to client;
12.               end thread;
13.            } /* end if */
14.            [res,tid]=compute(req);
15.            outcome=TestableTransaction(res,id);
16.            send Outcome[outcome,res,id] to client;
17.   } /* end while */

outcome TestableTransaction(result_type res, request_identifier id){
18.   insert(res,id); /* where id is a primary key */
19.   repeat{
20.      send Decide[tid] to the database server;
21.      await receive Outcome[outcome,exception,tid] or TIMEOUT;
22.   }until(message received);
23.   if (exception.type = duplicated_primary_key_exception){
24.      set res=exception.result;
25.      return COMMIT;
26.   } /* end if */
27.   return outcome;
28. }
```

Figure 2. Application Server Behavior.

same message is periodically re-sent in case of subsequent timeout expirations.

We assume the client request identifier to be a primary key for the database, which is the mechanism we adopt to guarantee the safety property. Therefore, any attempt to commit multiple transactions associated with the same client request identifier is rejected by the database itself, which is able to notify the rejection event by rising an exception. This makes the client request for updating data within the database an idempotent operation, i.e. the request can be safely re-transmitted multiple times to different application servers ([2]).

Upon the receipt of the Outcome message in reply from the database server ([3]), the flag $exception$ is checked to determine whether the same request identifier was already in the database. In the positive instance, a transaction associated with that same client request has already been committed. As a result, the exception allows the application sever to return an Outcome message with the commit indication to the client together with the already established result. In any other case (i.e. $exception$ is not raised), the outcome received by the database server is sent back to the client. The outcome might be rollback, e.g., due to decisions of the concurrency control mechanism.

A slightly different behavior is triggered at the application server upon the receipt of a Request message which includes the parameter $check$. In this case, the application server knows that we are in the presence of a retransmission of the same request from a client. Therefore, we might take performance benefits by exploiting the fact that a previous transmission of that same client request might have originated a transaction that has already been committed. To discover whether the transaction was already committed, the identifier of the client request is used by the primitive `lookup` that simply verifies whether the identifier itself has been already stored within the database. In the positive case, `lookup` returns the result of the transaction, stored together with the identifier, and the application server sends an Outcome message with commit to the client together with the associated result. This might help saving time and resources since neither `compute` nor `TestableTransaction` are executed. Otherwise, the application server processes the request as if the parameter $check$ were absent, and returns to the client the outcome of its interaction with the database.

[2] We note that assuming the client request identifier to be a primary key is a viable solution in practice. In case we can modify the database schema, this primary key can be easily added. In case the schema is predetermined and not modifiable (e.g. legacy databases), as suggested in [4] while describing supports for the testable transaction abstraction, an external table can be used.

[3] Given that the database server eventually recovers after a crash, a reply is eventually delivered to the application server.

```
Database Server:
1.    while(true){
2.       await receive Decide[tid] from an application server;
3.       [outcome,exception]=decide(tid);
4.       send Outcome[outcome,exception,tid] to the application server;
5.    }
```

Figure 3. Database Server Behavior.

The behavior of the database server is shown in Figure 3. For simplicity we only show the relevant operations related to transaction commitment, while skipping the data manipulation associated with the business logic. This server waits for a Decide message from an application server which asks to take a final decision for a transaction associated with a given tid, and then attempts to make the transaction updates permanent through the `decide` primitive. The final result (commit/rollback) is then sent back to the application server, together with the $exception$, possibly indicating the attempt to duplicate a primary key (i.e. the identifier of the client request) within the database. Recall that, as stated in Section 2, in case the parameter tid passed in input to the primitive `compute` is associated with an unrecognized transaction (recall this might happen in case the database server has crashed after the activation of that transaction and then has recovered before receiving the Decide message from the application server), this primitive returns a rollback outcome.

4 Related Work and Discussion

A typical solution for providing reliability consists of encapsulating the processing of the client request within an atomic transaction to be performed by the middle-tier (application) server [6]. This is the approach taken, for example, by Transaction Monitors or Object Transaction Services such as OTS or MTS. However, this solution does not deal with the problem of loss of the outcome due, for example, to middle-tier server crash. The work in [7] tackles the latter issue by encapsulating within the same transaction both processing and the storage of the outcome at the client by means of cookies. This solution imposes the use of a distributed commit protocol, i.e. two-phase commit (2PC), since the client is required to be included within the boundaries of a distributed transaction. Therefore, it relies on the exchange of prepare/vote messages among parties, thus exhibiting larger communication/processing overhead as compared to our protocol, in fact we do not include the client within the transaction boundaries and we do not make use of 2PC.

Several solutions based on the use of persistent queues have also been proposed in literature [1, 2], which are com-

monly used in industrial mission critical applications and supported by standard middleware technology (e.g. JMS in the J2EE architecture, Microsoft MQ and IBM MQ series). However, persistent queues are transactional resources, whose updates must be performed within the same transactional context in which the application data are accessed. This needs coordination among several transactional resources just through a distributed commit protocol (e.g. 2PC). Therefore, compared to our protocol, also in this case the communication/processing overhead is higher.

Message logging has also been used as a mean to recover from failures in multi-tier systems [8]. A client logs any request sent to the server, which also logs any request received. This allows the server to reply to multiple instances of the same request from a client without producing side effects on the back-end database multiple times. The server also logs read/write operations on the database, in order to deal with recovery of incomplete transaction processing. Differently from our proposal, this solution primarily copes with statefull client/middle-tier applications, e.g. like CAD or work-flow systems. Also, this solution needs to provide high availability of recovery information (i.e. the logs of received requests and of read/write operations) over the middle-tier to handle the fail-over. Given that this is typically obtained through replication of the recovery information, this protocol imposes some form of overhead and exhibits reduced scalability because of the handling of the coherency of the replicated information. We avoid such a problem by maintaining the application servers (i.e. the middle-tier) stateless.

Frolund and Guerraoui have presented three different e-Transaction protocols [3, 4, 5]. The solutions in [3, 5] are based on an explicit coordination scheme among the replicas of the application server, so they have to pay an additional overhead due to coordination. As a consequence, they are mainly tailored for the case of replicas of the application server hosted by a cluster environment, where the cost of coordination can be kept low thanks to low delivery latency of messages among the replicas. Since coordination among the replicas is not required in our protocol, we can avoid that overhead at all, with performance benefits especially in case of high degree of replication of the application server and distribution of the replicas on a geographical scale, e.g. like in ADNs.

Actually, the protocol in [4] is the closest one to our solution since it does not need coordination among the replicas of the application server and relies on recording some recovery information (having the same content as the one used in our proposal) at the back-end database during the processing of the transaction. However, the relevant difference between this proposal and our protocol is that we use part of the content of the recovery information (i.e. the client request identifier) as a primary key. This feature, in its turn, leads to a strong difference in the fail-over phase of the two protocols. Specifically, to maintain safety, the solution in [4] does not allow the client to simply re-submit its request to a possibly different replica of the application server, as instead we admit in our protocol. The proposal in [4] ensures safety via a so called "termination" phase, to be executed upon timeout expiration at the client side. During this phase, the client sends, on a timeout basis, terminate messages to the application servers until it receives a reply indicating whether the transaction associated with the last issued request message has been committed or has been rolled back. In case the client is notified of a rollback outcome, it can safely start a new round of interaction by re-sending the request message with a different identifier to whichever application server. On the other hand, upon the receipt of a terminate message, an application server forces a rollback operation on the database in order to ensure the abort of the corresponding transaction, in case it were still uncommitted. At this point the application server determines whether the transaction was already committed by checking if the recovery information associated with the transaction is stored within the database. In the positive case, the application server retrieves the transaction result to be sent to the client.

For what concerns the impact of the different structure of the two protocols on the fail-over, we need to consider the effect of the type of concurrency control used at the back-end database. With PCC, the termination phase executed by the protocol in [4] might help system responsiveness. This is because forced termination of a transaction, possibly left pending due to crash of the application server taking care of it, allows releasing the acquired locks. As a consequence, a new instance of the transaction will not be temporarily blocked by a previous instance accessing the same data within the database. Given that our protocol does not rely on any forced termination, a new instance of the transaction associated with the client request could be blocked by a pending previous instance until it experiences lock timeout for deadlock detection at the back-end database. On the other hand, with OCC our protocol is expected to provide better responsiveness while handling the fail-over. Specifically, a new instance of the transaction, originated by the retransmission of the client request upon timeout, can access data within the database with no additional delay caused by a previous instance possibly left pending due to crash of the application server taking care of it (this is because OCC allows each transaction to execute without blocking data being read/written). Therefore, in this circumstance, we pay no penalty possibly caused by the presence of a previous pending instance of the transaction, and no penalty due to the end-to-end additional interaction required to support the termination phase, as instead occurs for the protocol in [4].

As a last point, we note that forced rollback of pending transactions, required by the protocol in [4] during the termination phase, implies that explicit transaction demarcation must be performed at the database server side. By the admission of the authors, this should be done through the XA standard API [12]. However, XA specifications prescribe that upon a rollback operation of a transaction associated with a given identifier, namely XID in the XA terminology, the database system can reuse that XID value for a successive transaction activation. Hence, if a terminate message were processed before the corresponding request message in the protocol in [4], the latter message could possibly give rise to a transaction that gets eventually committed. On the other hand, upon the receipt of the reply to a terminate message indicating the rollback of the previously

issued request, the client would activate a new transaction, with a different XID, which could eventually get committed, thus leading to multiple updates at the database and violating safety. To prevent this problem, the authors suggest to delay the processing of the terminate messages at the application servers, so to enforce correct processing order at the database (i.e. a rollback operation must be executed after the corresponding transaction was already activated). Unfortunately, delaying the processing of terminate messages would penalize the user perceived system responsiveness during the fail-over phase.

5 Performance Analysis and Results

In this section we concentrate on a quantitative comparison between our protocol and the solution presented in [4], which, as previously discussed, is the closest one to our proposal. We focus on the analysis of the user perceived latency. This is done through the introduction of relatively simple analytical models suitable for comparing the two protocols in a wide range of environmental settings. While modeling protocol behaviors, we follow a bottom-up approach. Specifically, we first present a schematization of the main client-initiated interactions allowed by the two protocols (e.g. a termination interaction in case of the protocol in [4]). Latency models for those interactions are used as building blocks for the construction of complete models for the expected end-to-end latency at the client side. We derive the models assuming the back-end database provides OCC, which, as already pointed out in Section 4, is expected to exalt the features of our proposal. This allows a quantification of its potential when employed with settings it reveals tailored for.

Models for Basic Client-Initiated Interactions. In this paragraph, we provide latency models for the different basic client-initiated interactions allowed by the protocols. These models express mean latency values for interactions successfully completed with no timeout expiration at the client side (the effects of timeouts will be included while composing these models to evaluate the whole end-to-end protocol latency perceived by the end user).

The protocol in [4] is based on a request transmission interaction, as schematized in Figure 4.a ([4]), and on a request termination interaction, as schematized in Figure 4.b ([5]). Note that the request termination interaction can end with either a commit or a rollback indication to the client, depending on whether the transaction was already committed upon the issue of the rollback request by the application server to the database server. Actually, the real outcome is discovered by using the recovery information at the database, which is accessed via a lookup phase. Our protocol is based on a request transmission interaction, analogous to the one of the protocol in [4], shown in Figure 4.a, and on a request retry interaction, shown in Figure 4.c. The retry interaction includes the $check$ parameter. This allows checking, again through a lookup phase, whether the transaction has been already committed before activating any new instance. In the negative case, the new instance is activated, and the outcome is reported to the client. We denote as P_{commit} the probability that the application server finds the transaction already committed during either the request termination interaction in Figure 4.b for the protocol in [4] or during the request retry interaction in Figure 4.c for our protocol. In other words, P_{commit} indicates the probability that the lookup phase returns with an already established result for the transaction.

We can now derive expressions for the expected latency of the request transmission interaction in Figure 4.a, proper of both protocols, which we denote as T_{req}, and the expected latency of both the request termination and retry interactions (in Figure 4.b and in Figure 4.c, respectively), each one proper of a specific protocol, which we denote as T_{term} and T_{retry}. These expressions are:

$$T_{req} = RTT_{CL/AS} + RTT_{AS/DB} + T_{compute} + T_{ins} \quad (1)$$

$$T_{term} = RTT_{CL/AS} + RTT_{AS/DB} + T_{rollback} + T_{lookup} \quad (2)$$

$$T_{retry} = RTT_{CL/AS} + RTT_{AS/DB} + T_{lookup} + \\ + (1 - P_{commit})[T_{compute} + T_{ins}] \quad (3)$$

where: (i) $T_{compute}$ is the average time required to execute the transactional business logic. (ii) T_{ins} is the average time required to log the recovery information at the database. (iii) $RTT_{CL/AS}$ and $RTT_{AS/DB}$ represent, respectively, the average latency for a request/response interaction between a client and an application server, and between an application server and the database server. (iv) $T_{rollback}$ is the time required for handling a forced rollback request for a transaction. (v) T_{lookup} represents the time for performing a lookup operation in the table maintaining the recovery information.

We note that the expression for T_{term} does not depend on P_{commit}. This is because the termination phase for the protocol in [4] has the same pattern independently of whether the transaction the application server attempts to terminate through forced rollback was already committed or not. Anyway, as we shall show, the parameter P_{commit} plays a role in the expression for the whole end-to-end latency provided by the protocol in [4] since, in case the termination phase finds the transaction not committed, a new instance of request needs to be transmitted by the client.

End-to-end Latency Models. To build complete end-to-end latency models for the two protocols we need to consider timeout expiration at the client side. Actually, the timeout mechanism can give rise to false failure suspicions. The accuracy of such an approach to failure detection can be affected by a large number of factors, e.g. the choice of

[4] We consider the case of transactional logic activated at the database via a single message from the application server, e.g. like in stored procedures. This is done, without loss of generality, in order to avoid the introduction of an arbitrary delay in the model for the request transmission interaction caused by an arbitrary number of message exchanges between application and database servers for the management of the transactional logic. Similarly, the insertion of the recovery information, i.e. the request identifier and the result of SQL manipulations, is also managed within the stored procedure at the database side.

[5] Note that no delay has been introduced within this interaction for the processing of the terminate request at the application server, which, as discussed in Section 4, is required by the protocol in [4] to ensure safety. This choice derives from that no clear indication has been provided by the authors on the delay value. However, we underline that omitting this delay even favors the protocol in [4] in the comparative analysis.

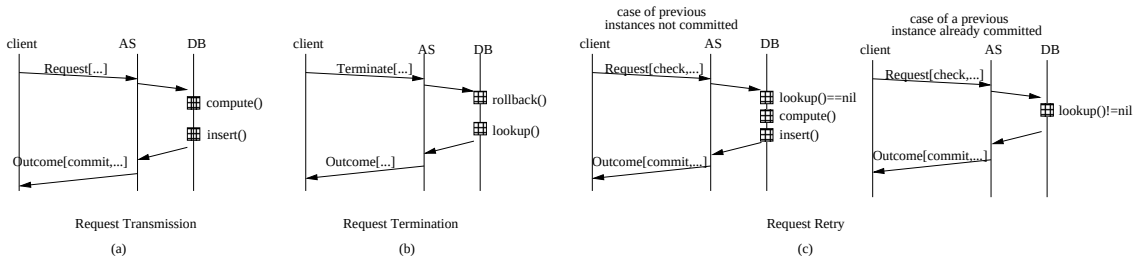

Figure 4. Basic Client-Initiated Interactions.

the timeout value with respect to the average system speed, as well as the variance of the average system speed, and the probability of failure of any process involved in the interaction. For simplicity, we abstract over these details, and model the effects of timeout expiration by a single parameter P_{TO}, namely the probability for a client to experience a timeout during any client-initiated interaction.

As a last preliminary observation, we note that the typical behavior of a Web browser (contacting the Web/application server through HTTP(S)) is to close the underlying TCP connection in case of timeout [11]. Hence, while deriving end-to-end latency models, we consider the case in which the client-initiated interaction during which timeout is experienced does not get eventually completed (just because the channel for the reply to the client is closed upon timeout).

On the basis of the above considerations, we can express the average user perceived latency for the considered protocols, which we denote as T^{fg} and T^{our_prot}, as follows:

$$T^{fg} = (1 - P_{TO})T_{req} + P_{TO}(TO + T^{fg}_{failover}) \quad (4)$$

$$T^{our\_prot} = (1 - P_{TO})T_{req} + P_{TO}(TO + T^{our\_prot}_{failover}) \quad (5)$$

where: (i) TO is the timeout value at the client side and (ii) $T^{fg}_{failover}$ and $T^{our\_prot}_{failover}$ represent the expected latency for the fail-over phase of the two protocols.

By expressions (4) and (5), the timeout latency TO and the latency for fail-over operations are experienced at the client side only in case of timeout expiration, i.e. with probability P_{TO}. In case of no timeout, the user perceived latency simply consists of the time for a request transmission interaction T_{req} as expressed in (1).

To complete the models, we have now to derive expressions for $T^{fg}_{failover}$ and $T^{our\_prot}_{failover}$. Actually, these expressions can be derived by thinking that fail-over is supported by the two protocols by simply composing client-initiated interactions, among those modeled in the previous paragraph, on a timeout basis. Specifically, the protocol in [4] lets the client activate request termination interactions on a timeout basis until an outcome is notified to the client. In case the outcome is rollback, the client selects a new request identifier and regenerates its initial behavior by activating a new request transmission interaction. Instead, our protocol lets the client simply activate request retry interactions until one of them is eventually completed with positive outcome for the transaction. As a consequence, the expected fail-over latencies can be expressed as follows:

$$T^{fg}_{failover} = (1 - P_{TO})[T_{term} + (1 - P_{commit})T^{fg})] + P_{TO}(TO + T^{fg}_{failover}) \quad (6)$$

$$T^{our\_prot}_{failover} = (1 - P_{TO})T_{retry} + P_{TO}(TO + T^{our\_prot}_{failover}) \quad (7)$$

By means of simple algebraic transformations and replacements, we finally obtain the following expressions for the end-to-end latency provided by the two protocols:

$$T^{fg} = \frac{(1 - P_{TO})T_{req} + P_{TO}(TO + T_{term} + \frac{P_{TO}TO}{1-P_{TO}})}{1 - (1 - P_{commit})P_{TO}} \quad (8)$$

$$T^{our\_prot} = (1 - P_{TO})T_{req} + P_{TO}(TO + T_{retry} + \frac{P_{TO}TO}{1 - P_{TO}}) \quad (9)$$

$T_{compute} + T_{ins}$	T_{lookup}	$T_{rollback}$
47.30	1.42	0.55

	$RTT_{CL/AS}$	$RTT_{AS/DB}$
Scenario A	150	150
Scenario B	150	5

Figure 5. Parameter Values (msecs).

Parameter Treatment. In order to use realistic values for $T_{compute}$, T_{ins}, $T_{rollback}$ and T_{lookup}, we have developed prototype implementations of (i) basic modules supporting the actions required by the protocols at the database side, and of (ii) the Payment Transaction profile, specified by the well known TPC BENCHMARK^{TM} C [13]. The top table in Figure 5 lists the costs of the activity on the back-end database, which have been measured by running the Solid FlowEngine 4.0 DBMS [10] on top of a multi-processor server equipped with 4 Xeon 2.2 GHz, 4 GB of RAM and 2 SCSI disks in RAID-0 configuration. The application logic was implemented in JAVA2 with stored procedure technology. Each reported value, expressed in msec, is the average over a number of samples that ensures confidence interval of 10% around the mean at the 95% confidence level.

For what concerns the parameters $RTT_{CL/AS}$ and $RTT_{AS/DB}$, we note that they are typically dependent on the relative locations of clients, application servers and database server. In the analysis we consider the following two classical scenarios for Web based transactional systems:

Scenario-A: Clients, application and database servers are all geographically distributed and communicate with each other through the Internet.

Scenario-B: Geographically spread clients, connected to the application servers through the Internet. Application and database servers residing either on the same

LAN or on a geographically distributed infrastructure with low/controlled message delivery latency, e.g. a (private) dedicated WAN.

The bottom table in Figure 5 shows the considered values for $RTT_{CL/AS}$ and $RTT_{AS/DB}$ in each scenario.

P_{TO} and P_{commit} have been left as independent parameters in the performance study. This choice has been taken mostly because the real value of these parameters might depend on a large set of unpredictable environmental factors like, for example, (i) the ratio between the selected timeout value and the current system speed (if for some reason, e.g. host/network overload, the system speed gets reduced then we might get an increase in the likelihood of timeout) and (ii) the likelihood for a failure to occur before the transaction is committed at the database (in this case we need to re-execute the transaction during the fail-over) or after the commit, i.e. just while reporting the result to the client (in this case the fail-over will simply retrieve the transaction result to be communicated to the client). Therefore, the treatment of P_{TO} and P_{commit} as independent parameters has the advantage of allowing us not to exclude any possibility for what concerns those environmental factors.

Finally, for both **Scenario-A** and **Scenario-B**, the timeout value TO has been set to 30 seconds.

Results. We use the end-to-end latency models previously presented to plot the Additional Overhead Percentage (AOP), defined as follows:

$$\text{AOP} = \frac{T^{fg} - T^{our\_prot}}{T_{req}} \tag{10}$$

AOP represents the expected additional latency perceived by the end user when employing the protocol in [4] with respect to the one provided by our proposal, normalized to the expected latency T_{req} for a request transmission that nicely ends with no timeout expiration at the client side.

In Figures 6 and 7 we report the AOP values respectively for the two considered scenarios. While drawing, we let P_{TO} vary from 0 to 0.1, selected as a reasonable interval. The second independent parameter, namely P_{commit}, has instead been varied in an interval centered around 0.5. This choice follows from that the likelihood for the lookup phase to return an already established result, or a null value, respectively, depends on whether the lookup primitive is executed before, or after, the commit of the transaction associated with the previously issued request. Given that failures or overloads might occur at any time instant, it is reasonable that the lookup phase equally likely returns with either an already established result for the transaction or a null result indicating that the transaction was not already committed, which it the reason why the observation interval for P_{commit} is centered around the value 0.5. The dotted lines on the surfaces are level curves and are traced at the distance of 10% of AOP.

The plotted surfaces show similar shapes, indicating that the protocol in [4] exhibits significant additional overhead (i.e. of at least of 25%), as compared to our proposal, as soon as we have minimal likelihood of timeout expiration at the client side. On the other hand, with an additional

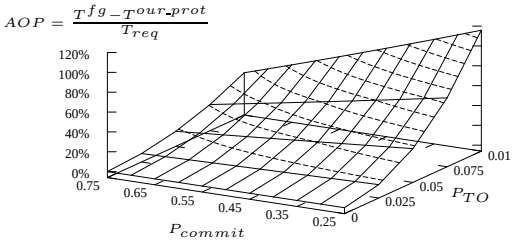

Figure 6. AOP for Scenario-A.

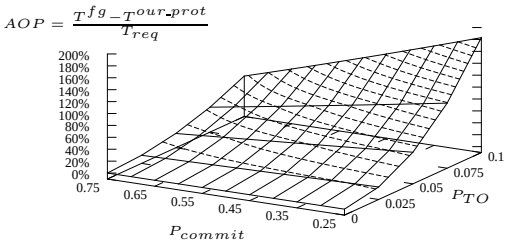

Figure 7. AOP for Scenario-B.

slight growth of P_{TO}, our protocol shows an overhead percentage of two order of magnitudes lower that the one in [4]. Such a gain appears independently of the considered scenario. However, fixed some specific values for P_{TO} and P_{commit}, the maximum reduction of the overhead percentage thanks to our protocol is achieved for **Scenario-B**. This phenomenon is due to the fact that, when moving from Internet distribution of all the involved processes (i.e. **Scenario-A**) to the case of an infrastructure with lower communication latency among application servers and database server (i.e. **Scenario-B**), the value of T_{req}, expressing the end-to-end latency in case of normal behavior (i.e. no timeout expiration at the client) gets reduced. At the same time, the contribution of the Internet latency between client and application servers still plays a relevant role for the protocol in [4] due to its impact on the termination phase required by the protocol itself in case of timeout expiration. Hence, normalization of the difference of the latencies provided by the two protocols by T_{req} exalts the performance of our protocol thanks to its avoidance of the termination phase.

References

[1] P. Bernstein, M. Hsu and B. Mann, "Implementing Recoverable Requests Using Queues", *Proc. 19th ACM Int. Conference on the Management of Data (SIGMOD)*, pp.112-122, 1990.

[2] E.A. Brewer, F.T. Chong, L.T. Liu, S.D. Sharma and J.D. Kubiatowicz, "Remote Queues: Exposing Message Queues for Optimization and Atomicity." *Proc. 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Santa Barbara, CA, pp.42-53, 1995.

[3] S. Frolund and R. Guerraoui, "Implementing e-Transactions with Asynchronous Replication", *IEEE Transactions on Parallel and Distributed Systems*, vol.12, no.133-146, pp.2001.

[4] S. Frolund and R. Guerraoui, "A Pragmatic Implementation of e-Transactions", *Proc. 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pp.186-195. 2000.

[5] S. Frolund and R Guerraoui, "e-Transactions: End-to-End Reliability for Three-Tier Architectures", *IEEE Transactions on Software Engineering*, vol.28, no.4, pp. 378-398, 2002.

[6] J. Gray and A Reuter, "Transaction Processing: Concepts and Techniques", Morgan Kaufmann, 1993.

[7] M.C. Little and S.K. Shrivastava, "Integrating the Object Transaction Service with the Web", *Proc. 2nd IEEE Int. Workshop on Enterprice Distributed Object Computing (EDOC)*, pp.194-205, 1998.

[8] D. Lomet and G. Weikum, "Efficient Transparent Application Recovery in Client-Server Information Systems", *Proc. 27th ACM Int. Conference on the Management of Data (SIGMOD)*, pp.460-471, 1998.

[9] P. Romano, F. Quaglia and B. Ciciani, "An Efficient e-Transaction Protocol Exploiting Optimistic Concurrency Control", Tech. Rep. 17-03, Dipartimento di Informatica e Sistemistica, Universita' di Roma "La Sapienza". June 2003. `http://ftp.dis.uniroma1.it/pub/quaglia/tr17-03.ps`

[10] Solid Information Technology Ltd, "Solid FlowEngine", `www.solidtech.com`

[11] The Jakarta Project, "Jakarta Commons: HTTP Client", `http://jakarta.apache.org`

[12] The Open Group, "Distributed Tansaction Processing: the XA+ Specification Version 2", `http://www.opengroup.org/onlinepubs/008057699/toc.pdf`

[13] Transaction Processing Performance Council (TPC), "TPC Benchmark ^{TM} C, Standard Specification, Revision 5.1", 2002.

[14] P.S. Yu, D.M. Dias and S.S. Lavenberg, "On the Analytical Modeling of Database Concurrency Control", *Journal of the ACM*, vol.40, no.4, pp.831-872, 1993.