Ensuring e-Transaction Through a Lightweight Protocol for Centralized Back-end Database

Paolo Romano, Francesco Quaglia and Bruno Ciciani

DIS, Università "La Sapienza", Roma, Italy

Abstract. A reasonable end-to-end reliability guarantee for three-tier systems, called e-Transaction (exactly-once Transaction), has been recently proposed. This work presents a lightweight e-Transaction protocol for centralized back-end database. Our protocol does not require coordination among the replicas of the application server and does not rely on any assumption for what concerns the processing order of messages exchanged among processes, as instead required by some existing solution.

1 Introduction

The concept of "e-Transaction" (exactly-once Transaction) has been recently introduced in [5] as a desirable, yet realistic, form of end-to-end reliability guarantee for three-tier systems. In this paper we present an e-Transaction protocol for three-tier systems in which the application servers interact, as in the case of most e-Commerce applications, with a centralized back-end database. Our e-Transaction protocol handles failures (or suspect of failures due to reduced system responsiveness possibly caused by host/network overload) by simply letting the client perform a timeout based retransmission logic of its request to different replicas of the application server. On the other hand, we use some recovery information, locally manipulated at the database side, to guarantee that the corresponding transaction is committed exactly one time. Manipulation of the recovery information does not require coordination among the application server replicas, which do not even need to know each other existence. Our proposal is therefore inherently scalable, and well suited for both local and geographic distribution of the replicas themselves.

Beyond providing the description of the protocol, together with its correctness proof and experimental measures demonstrating its minimal overhead, we present an extended comparative discussion with existing proposals in support of reliability, pointing out the advantages from our solution. The discussion will also outline that the proposal closest to our protocol (both in terms of structure and overhead compared to a baseline approach that does not provide reliability guarantee), namely the one in [4], relies on specific assumptions for what concerns the order of message processing to avoid duplication of transactions at the back-end database. Our solution does not require any of those assumptions, thus being suitable for a wider range of system settings.

2 System Model

The target three-tier system we consider, consists of a set of processes, which communicate through message exchange. Processes can fail according to the crash-failure model. If a process does not fail, we say that the process is correct, and we assume there is at least a correct application server process at any time. Communication channels between processes are assumed to be reliable, therefore each message is eventually delivered unless either the sender or the receiver crashes during the transmission. In what follows we present features of each class of processes in the system, i.e. client, application server and database server, together with basics about the recovery information maintained at the back-end database.

Client. A client process does not communicate with the database server, it only interacts with the application servers in the middle-tier. It sends a request to an application server in order to invoke the transactional logic on this server (i.e. the application server is requested to perform some updates on the back-end database) and then waits for the outcome. The client sends the request by invoking the function `issue`, which takes the request content as a parameter. `issue` returns only upon the receipt of a positive outcome (commit) for the transaction ([1]).

Application Server. Application server processes have no affinity for clients. Moreover, they are stateless, in the sense that they do not maintain states across requests from clients. A request from a client can only determine changes in the state of the back-end database. 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. We assume 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. `compute` is assumed to be non-blocking, which means it eventually returns unless the application server crashes.

[1] For simplicity of presentation, we model with positive outcome also transactions for which the application logic cannot admit update on the database, e.g. like in a bank transfer operation with not enough money on the corresponding account, but that are correctly handled by the database, e.g. with no rollback caused by the concurrency control mechanism.

Database Server. The system back-end consists of a database server which is assumed to eventually recover after a crash. Also, there is a time after which the database stops crashing and remains up, allowing any legal transaction to be eventually committed ([2]). In practice, this means assuming the database can experience a period of instability during which it can crash and recover, and then experiences a period during which it does not crash, which is long enough to allow a legal transaction to be eventually committed.

The database server has a primitive `decide` which can be used to invoke the commitment of a pending transaction. This primitive is called by the database server (with a transaction identifier as the unique parameter) upon the receipt of a message from the application server that asks for a final decision (commit/rollback) for a pending transaction. `decide` returns commit/rollback depending on the final decision the database takes for the transaction, together with any exception possibly raised during the decision phase. We assume that the `decide` primitive is non-blocking, i.e. it eventually returns unless the database server crashes after the invocation. 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, then the return value of this primitive is rollback.

Recovery Information. The database offers an 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 a unique client request identifier, which is stored within the database as a part of the transaction execution itself, together with the result of the transaction. If the identifier is stored within the database, then this means that the corresponding client request originated a transaction that has already been committed. Note that the testable transaction abstraction can be implemented in a transparent way to the client by simply modifying the application server transactional logic. Specifically, as in [4], we assume application servers have an additional primitive `insert` allowing them to ask the database server to write the identifier of a client request within the database together with the result obtained by the execution of the `compute` primitive. Additionally, just like `compute`, the primitive `insert` is non-blocking, i.e. it eventually returns unless the application server crashes.

3 The Protocol

The protocol we present ensures the following two properties synthesizing the e-Transaction problem as introduced in [4, 5]:

[2] We use the term "legal" to refer to a transaction that does not violate any integrity constraint on the database. As an example, the attempt to duplicate a primary key makes a transaction illegal.

Safety. The back-end database does not commit more than one transaction for each client request.

Liveness. 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.

It is important to note that, 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).

We present the protocol describing its behavior separately for every class of processes in the system, i.e. client, application server and database server.

Client Behavior. The pseudo-code defining the behavior of the function `issue` used by the client is shown in Figure 1. 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 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` reselects an application server (possibly different from the last selected one) and retransmits the Request message to that application server, with the already selected request identifier. Upon successive timeout expirations, the client keeps on retransmitting the Request message (with that same identifier) until it receives the commit outcome.

Application and Database Server Behaviors. The application server behavior is shown in Figure 2. Upon the receipt of a Request message, this server invokes the primitive `compute` to start a transaction on the back-end database. The transaction identifier assigned by the database server is returned to the application server and maintained by *tid*. The application server then invokes `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 with that *tid* to the database server and waits for the outcome. This same message is periodically retransmitted 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,

```
result 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] to AS;
10.      } /* end if */
11.   } /* end while */
12.   return res;
13. }
```

Fig. 1. Client Behavior.

```
Application Server:
1.    result res;
2.    transaction_identifier tid;
3.    while(true){
4.          await receive Request[req,id] from client;
5.          [res,tid]=compute(req);
6.          outcome=TestableTransaction(res,id);
7.          send Outcome[outcome,res,id] to client;
8.    } /* end while */

outcome TestableTransaction(result res, request_identifier id){
9.    insert(res,id); /* where id is a primary key */
10.   repeat{
11.      send Decide[tid] to the database server;
12.      await receive Outcome[outcome,exception,tid] or TIMEOUT;
13.   }until(received Outcome[outcome,exception,tid]);
14.   if(exception.type = duplicated_primary_key_exception){
15.      set res=exception.result;
16.      return COMMIT;
17.   } /* end if */
18.   return outcome;
19. }
```

Fig. 2. Application Server Behavior.

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 retransmitted multiple times to different application servers. 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.

Upon the receipt of the Outcome message in reply from the database server, 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.

```
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.    }
```

Fig. 3. Database Server Behavior.

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.

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.

3.1 Proof of Correctness

Theorem 1. - Safety
The back-end database does not commit more than one transaction for each client request.

Proof. **(By Contradiction).** Given the structure of the protocol, it is possible that multiple transactions associated with the same client request are activated by the application servers. Assume, by contradiction, that a generic number $N > 1$ of them are eventually committed. In this case, the database server must have received multiple Decide messages from the application servers for transactions associated with the same client request. By the application server pseudo-code, this server sends the Decide message to the database server (see line 11 in Figure 2) only after it has executed a whole transaction that encompasses both the data manipulation proper of the application business logic through the `compute` primitive (see line 5 in Figure 2), and the storing of the request unique identifier together with the result of the data manipulation through the `insert` primitive (see line 9 in Figure 2). As a consequence, the $N > 1$ transactions associated with the same client request, which are eventually committed, must perform a successful insertion of the unique request identifier within the database. However, this is impossible since the database maintains a primary key constraint on the request identifier, hence no more than one of those N transactions can perform that insertion successfully. Therefore the assumption is contradicted and the claim follows.

Lemma 1.
If a correct application server sends a Decide *message to the database server*

asking for a decision on a transaction, the application server eventually receives an Outcome *message for that transaction from the database server.*

Proof. **(By Contradiction).** Assume by contradiction that a correct application server sends a `Decide` message to the database server and that no Outcome message from the database server is ever received for the corresponding transaction. In this case, the correct application server keeps on retransmitting the Decide message to the database server indefinitely (see lines 10-13 in Figure 2). Hence, a Decide message will be sent by the application server to the database server at time $t' > t$, where t be the time after which the database server stops crashing and remains up. Given that after time t both the correct application server and the database server are always up, for the assumption on the reliability of the communication channels we can claim that the database server will eventually receive the Decide message. Also, the database server will eventually take a decision through the `decide` primitive (since it does not crash anymore) and will send an Outcome message to the application server. Again, since communication channels are assumed to be reliable, the correct application server will eventually receive that Outcome message. Therefore the assumption is contradicted and the claim follows.

Theorem 2. - Liveness
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.

Proof. **(By Contradiction).** Assume by contradiction that the client issues a request, does not crash and does not eventually receive a commit outcome. In this case, the client keeps on retransmitting the Request message to the application servers indefinitely (see lines 5-11 in Figure 1). As we have assumed that channels are reliable and that at least an application server is correct (i.e. it does not crash), an unbounded amount of Request messages will eventually be delivered to a correct application server. Moreover, if an Outcome message is sent back by a correct application server to the client, this message will eventually be received, since the client does not crash. Then, to show that the previous assumption is wrong, we only need to show that a correct application server receiving an unbounded amount of Request messages will eventually send to the client an Outcome message with a commit indication.

When a correct application server receives a Request message from the client, it calls the primitives `compute` and `insert` (see lines 5 and 9 in Figure 2). These primitives, being non-blocking, eventually return, therefore the application server eventually sends to the database server the Decide message (see line 11 in Figure 2). By Lemma 1, an Outcome message for the transaction is eventually sent back by the database server and is eventually received by the correct application server (recall this message also carries the value of the *exception* flag). There are two possible cases:

A.1 If the Outcome message received from the database server carries a commit indication or the *exception* flag notifies the attempt to duplicate a primary key, then an Outcome message with commit is sent back to the client together with the transaction result. Therefore, the assumption is contradicted.
A.2 If the Outcome message received from the database server carries a rollback indication, with the *exception* flag notifying no attempt to duplicate a primary key, then the transaction was legal but such a reply from the database server implies that

the database was unable to commit the required operations (e.g. due to decisions of the concurrency control mechanism). In this case, an Outcome message with a rollback indication is sent to the client.

We note anyway that case A.2 (i.e. the only one that does not contradict the assumption) can't occur indefinitely as we have assumed that there is a time after which the database server remains up and is able to commit any legal transaction. We can therefore assert that we eventually fall in case A.1, which contradicts the assumption. Hence, the claim follows.

3.2 Protocol Overhead

Our protocol is essentially based on logging recovery information (i.e. the client request identifier and the result of the transaction) at the back-end database while processing the transaction associated with the client request. The cost of logging this recovery information is actually the unique overhead we pay as compared to a baseline protocol for the three-tier organization, which is not able to provide any end-to-end reliability guarantee. We argue that this overhead is negligible in practice since it only consists of the cost for a single SQL INSERT statement. To support this claim, we have performed some measurements related to the New-Order and the Payment Transactions specified by the TPC BENCHMARK^{TM} C [10], both reflecting on-line database activity, as typically found in production environments, but exhibiting different profiles for what concerns read/write operations. The measurements have been taken by running the Solid FlowEngine 4.0 DBMS on top of a multi-processor system, equipped with 4 Xeon 2.2 GHz, 4 GB of RAM and 2 SCSI disks in RAID-0 configuration, running Windows 2003 Server. The application logic was implemented in JAVA2 with stored procedure technology. The below table reports the cost of database activities for both the baseline protocol and our proposal. 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. These experimental data clearly show that the overhead exhibited by our protocol for logging the recovery information is minimal, never exceeding 2%, even for the lighter transaction profile, namely the Payment Transaction.

	Baseline	Our protocol	Overhead
New-Order Transaction	72.2	73.1	+1.21%
Payment Transaction	46.4	47.3	+2.06%

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. This solution imposes the use of a distributed commit protocol, such as two-phase commit (2PC), since the client is required to be part of the transactional system. Therefore, it exhibits higher communication/processing overhead as compared to our protocol.

Several solutions based on the use of persistent queues have also been proposed in literature [1, 2], which are commonly deployed 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 where the application data are accessed (i.e. the request message must be dequeued within the same distributed transaction that manipulates application data and enqueues the response to the client). 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. Additionally, as discussed in [3, 5], the use of persistent queues, in combination with classical 2PC as the distributed commit protocol, imposes explicit coordination among the application servers to support fail-over of an application server (i.e. the coordinator of the distributed transaction) suspected to have crashed. This originates additional overhead and reduces scalability. Since our protocol does not use any coordination scheme among the application server replicas, it provides better system performance and scalability, thus being attractive especially in the case of high degree of replication of the application access point, with the replicas possibly distributed on a geographic scale, e.g. like in Application Delivery Networks (ADNs) such as those provided by Sandpiper, Akamai or Edgix ([3]).

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 stateful client/middle-tier applications, e.g. like CAD or work-flow systems.

Frolund and Guerraoui have presented three different e-Transaction protocols [3–5]. 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, e.g., a cluster environment, where the

[3] These infrastructures result as 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, and to increase the application availability.

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).

Like our solution, the third protocol by Frolund and Guerraoui [4] relies on the testable transaction abstraction ([4]), and has the advantages of not requiring explicit coordination across the middle-tier and of not using any distributed commit scheme. However, differently from our proposal, it handles failure suspicions through a "termination" phase executed upon timeout expiration at the client side. During this phase, the client sends, on a timeout basis, terminate messages to the application servers in the attempt to discover whether the transaction associated with the last issued request was actually committed. An application server that receives a terminate message from the client tries to rollback the corresponding transaction, in case it were still uncommitted (possibly due to crash of the application server originally taking care of it). At this point the application server determines whether the transaction was already committed by exploiting the testable transaction abstraction. In the positive case, the application server retrieves the transaction result to be sent to the client. Otherwise, a rollback indication is returned to the client in order to allow it to safely send a new request message (with a different identifier) to whichever application server.

Our protocol avoids the termination phase since it makes retransmissions of a same request idempotent operations thanks to the use of a primary key constraint imposed on the recovery information. From the point of view of performance, the avoidance of the termination phase reduces the fail-over latency as compared to [4]. More importantly, avoiding the termination phase makes our protocol a more general solution. In fact, by admission of the same authors, the employment of such a phase limits the usability of their protocol to environments where it can be ensured that a request message is always processed before the corresponding terminate messages. This is due to the fact that, according to the specifications of the standard interface for transaction demarcation, namely XA, when a rollback operation is performed for a transaction with a given identifier, the database system can reuse that identifier for a successive transaction activation (see [9] - state table on page 109). Hence, if a terminate message was 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 a reply to a terminate message, the client might activate a new transaction, with a different identifier, which could eventually get committed, thus leading to multiple updates at the database and violating safety. In order to achieve the required processing order constraint for request and terminate messages, the authors suggest to delay the processing of the terminate messages at the application servers. This expedient might reveal adequate in case the application is deployed over an infrastruc-

[4] Also this protocol logs some recovery information at the database while processing the transaction through a similar `insert` primitive.

ture with controlled message delivery latency and relative process speed, e.g. a (virtual) private network or an Intranet. However, if the system can experience periods during which the message delivery latency gets unpredictably long and/or the process speeds diverge, e.g. like in an asynchronous system, simply delaying the processing of a terminate message would not suffice to ensure such an ordering constraint. The latter constraint could be enforced through additional mechanisms (e.g. explicit coordination among the servers), but these would negatively affect both performance and scalability of this protocol. By all means, delaying the processing of terminate messages, even if adequate for specific environments, would further penalize the user perceived system responsiveness during the fail-over phase as compared to our solution. Conversely, our protocol does not rely on any constraint on the processing order of messages exchanged among processes, thus it requires no additional mechanism to enforce such an order and can be straightforwardly adopted in an asynchronous system, e.g. an infrastructure layered on top of public networks over the Internet.

References

1. P. Bernstein, M. Hsu and B. Mann, "Implementing Recoverable Requests Using Queues", *Proc. 19th ACM Conference on the Management of Data*, 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*, 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*, 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 Workshop on Enterprice Distributed Object Computing*, pp.194-205, 1998.
8. D. Lomet and G. Weikum, "Efficient Transparent Application Recovery in Client-Server Information Systems", *Proc. 27th ACM Conference on the Management of Data*, pp.460-471, 1998.
9. The Open Group, "Distributed Transaction Processing: The XA+ Specification Version 2", 1994.
10. Transaction Processing Performance Council (TPC), "TPC Benchmark^{TM} C, Standard Specification, Revision 5.1", 2002.