A Lightweight and Scalable e-Transaction Protocol for Three-Tier Systems with Centralized Back-End Database*

Paolo Romano, Francesco Quaglia and Bruno Ciciani
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy

Abstract

The e-Transaction abstraction is a recent formalization of end-to-end reliability properties for three-tier systems. In this work we present a protocol ensuring the e-Transaction guarantees in case the back-end tier consists of a centralized database. Our proposal addresses the case of stateless application servers, and is both simple and effective since (i) it does not employ any distributed commit protocol, and (ii) does not require coordination among the replicas of the application server.

Index-terms: Transaction Processing, Reliability, Efficient Fail-over, Three-Tier Systems.

1 Introduction

The concept of "e-Transaction" (exactly-once Transaction) has been recently introduced in [7] as a reasonable form of end-to-end reliability guarantee for a three-tier transactional system. Essentially, this abstraction guarantees that a transaction is executed exactly once at the back-end database, and that the client eventually receives the result of the server side computation, despite failures. In this paper we consider three-tier systems with a centralized back-end database, as in the case of most e-Commerce Web sites, and propose a simple e-Transaction protocol, suited for stateless application servers, with the following properties: (1) It does not rely on any distributed commit protocol, such as two-phase commit (2PC), thus avoiding the overhead of distributed commit schemes (in terms of increased message complexity and latency). (2) It does not impose any explicit coordination among the replicas of the application server, which makes our proposal inherently scalable and, hence,

*This article appeared in IEEE Transactions on Knowledge and Data Engineering, Vol.17, No.11, November 2005.

particularly attractive for emerging Web infrastructures like Application Delivery Networks (ADNs). These are characterized by a high degree of replication and geographic distribution of the access point to the application logic.

Beyond providing the description of the protocol, we also propose a quantitative evaluation of its performance vs other solutions. This evaluation has been carried out through the industry standard TPC-C benchmark [17] and parameterized analytical models allowing us to consider a wide spectrum of 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 description of the protocol. A comparative discussion with related work is provided in Section 4. Section 5 presents the quantitative analysis.

2 System Model

We consider a classical three-tier architecture, with reliable connection oriented communication (e.g., TCP like) between clients and application servers, and between application servers and the back-end database server. Processes can fail according to the crash-failure model [9]. Also, failures are soft so that logged information is not corrupted, hence allowing the database server to correctly recover. All messages are eventually delivered unless either the sender or the receiver crashes during the transmission, or the underlying connection breaks. For simplicity of presentation, but with no loss of generality, we model with a "connection lost exception" both the event of broken network connection and the event of timeout expiration at the application level while waiting for incoming data on an established connection ([1]).

We consider the case of stateless application servers, which do not maintain states across request invocations. As discussed in [6], stateless application servers provide the following benefits: (i) Fail-over is fast because we do not have to wait for a server to recover its state. (ii) Stateless servers do not have host affinity (hence we can freely migrate them), and do not have client affinity (hence a client can be easily redirected to a different replica during fail-over). For simplicity, we do not consider chained invocation of application servers (e.g., as in multi-tier organizations). However, for what concerns reliability aspects, this does not result in any loss of generality since we consider stateless application servers. As a consequence, the crash of the application server contacted by the client in our model is equivalent to the crash of any of the application servers in a chained invocation scheme. The business logic executed

[1]With respect to this point, 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 on incoming data on that connection [15].

by the application server within a transaction contains one or more database interactions (e.g., via SQL statements). Also, we assume that a transaction is left uncommitted as long as no commit statement is issued from the application server to the database server (e.g., by avoiding the auto-commit mode in ODBC or JDBC technology).

The back-end consists of a database server, offering ACID properties, which is accessible through standard connection oriented programming interfaces like, e.g., ODBC or JDBC. As in conventional DBMS technology, every uncommitted transaction is aborted in case of database server crash. The same happens to an uncommitted transaction in case the corresponding connection between the application server and the database server gets lost due, e.g., to application server crash. The abort of a transaction can also occur for the following set of reasons: (i) Autonomous decisions taken by the database server (e.g., concurrency control driven aborts). (ii) The corresponding client request has a malformed content with respect to the application semantic (e.g., it attempts to perform a bank transfer with not enough money on the corresponding account). (iii) The transaction is illegal since it attempts to violate any integrity constraint on the database (e.g., it attempts to duplicate a primary key).

3 The Protocol

Our protocol ensures the following two properties synthesizing the e-Transaction abstraction for the considered system model: (i) The back-end database does not commit more than one transaction for each client request (*Safety* - at most once). (ii) If a client sends a well formed 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). The combination of safety and liveness provides exactly-once semantic for the processing of a well formed request sent by a client that does not crash. (According to [5, 7], an e-Transaction protocol is not required to ensure liveness in the presence of client crash. This might be acceptable when thinking that the e-Transaction framework deals with very thin clients, typically having no ability to maintain recovery information, as in the case of applications accessible through cell-phones, or if it is impossible to access the client disk storage both for security and privacy issues, as in the case of Web-sites invasively delivering cookies. However, if stable storage capability at the client side is admitted, a simple extension of our protocol might be adopted to ensure liveness even in the presence of client crashes, as we shall discuss.) We present the protocol describing the client and the application server behaviors. (As stated in Section 2, the back-end database complies with conventional DBMS technology and is accessible through standard connection oriented interfaces. Hence, the database server behavior does not need any explicit description.)

```
(Outcome, Result) client_stub(request_content req){
1.  Identifier id; Boolean retransmitting=false;
2.  set a new value for id;
3.  select an application server AS;
4.  while (true){
5.      try {connect to AS;
6.              if ( !retransmitting ) send Request[req,id];
7.              else send Request[req,id,check];
8.              await receive Outcome[outcome,res,id] ;
9.              if ( (outcome==commit) or (outcome==malformed_request) ) return [outcome,res];
10.     } catch ( connection_lost_exception ) { select a different application server AS; retransmitting=true; }
11.} /* end while */
} /* end client_stub */
```

Figure 1: Client Behavior.

Client Behavior. The pseudo-code defining the client behavior (within a stub used to hide the lower level protocol logic from the application level) is shown in Figure 1. The stub takes the request content as input and returns an outcome (commit or abort due to a malformed request) and a result encoding any output message, e.g., a response HTML page. The client stub executes a very simple logic. It generates an identifier to be associated with the request, connects to an application server and sends a Request message to this server, tagged with the request identifier. It then waits for the reply. In case it receives *commit* or *malformed_request*, this outcome and the corresponding result are returned to the client application. In case the *abort* outcome is received, the stub retransmits the Request message to that same application server. On the other hand, in case a *connection_lost_exception* is raised, the client stub connects to a different application server replica and retransmits the Request to it ([2]). Each time the client stub retransmits the Request message, it includes in the message an additional parameter, namely *check*, which notifies to the application server that this request is a resubmission.

Application Server Behavior. The application server behavior is shown in Figure 2. After a connection from a client is established and a Request message is received, the application server connects in its turn to the back-end database. If *check* is not included in the Request message (i.e., the **if** statement in line 5 is skipped), then the application server

[2] With current software technology, a number of ways exist to make the client stub aware of the identities of a set of replicated application servers. For instance, in case the stub were embedded within an applet, we could make it aware of those identities at download time, or it could dynamically discover them, e.g., through UDDI based services.

executes the transactional business logic on the back-end database and computes the result to be delivered to the client application. Then, the application server executes an insert operation of the tuple (id,res) containing the client request identifier and the computed result within a database table. Afterwards, it issues the commit statement to the database server. Actually, the insertion of that tuple, performed by the application server as a part of the execution of the transaction encompassing the application business logic, implements the "testable transaction" abstraction [1, 5], according to which *recovery information* is used to determine whether a given transaction has already been committed, and also to retrieve the corresponding result. Our instance of testable transaction uses the client request identifier id as a primary key, which is the mechanism we adopt to guarantee the safety property. Hence, any attempt to commit multiple transactions associated with the same client request identifier is rejected by the database itself, which detects the attempt to duplicate the primary key and notifies the rejection event by rising a *duplicate_id_exception*. This makes the client request for updating data within the database an idempotent operation, so that the request can be safely retransmitted multiple times to different application servers without incurring the risk of duplicate transactions at the back-end database.

In case the $malformed\_request\_exception$ is raised when processing the business logic on line 9, the client stub is notified, via an Outcome message carrying the $malformed\_request$ indication, that the request content does not comply with the application semantic.

In case the *duplicate_id_exception* is raised by the insertion of the tuple (id,res) on line 10, the application server retrieves the result of the already committed transaction, associated with that same client request, through a `lookup` operation that takes as input the client request identifier. The result is then sent back to the client stub through an Outcome message, also providing the *commit* indication (the client request has been already processed successfully).

Upon the receipt of a Request message which includes the parameter *check* indicating a retransmission (i.e., the **if** statement in line 5 is executed), the application server calls `lookup` to verify whether the request identifier, and the corresponding result, have been already stored within the database. In the positive case, the application server sends an Outcome message with *commit* to the client stub together with the associated result. Hence, the application server avoids executing the transactional business logic prior to discovering, through the *duplicate_id_exception*, that an instance of the same transaction has been already committed. This might help saving time and resources, especially in case the execution of the business logic is compute intensive.

Other two types of exceptions might be raised while interacting with the back-end database,

```
void ApplicationServer(void){ /* accepts a client connection and processes the client request */
result_type res;
try {accept client connection;
        await Request message from client;
        try {connect to DB;
                if ( received Request[req,id,check] ) {
                        res = lookup(id);
                        if ( res != nil ) { send Outcome[commit,res,id] to client; return; }
                } /* end if */
                execute the business logic associated with req and compute res;
                insert the tuple (id,res) in a database table; /* primary key constraint maintained on id values */
                issue the commit statement to DB;
                send Outcome[commit,res,id] to client;
        } catch ( malformed_request_exception ) { send Outcome[malformed_request,res,id] to client; }
            catch ( duplicate_id_exception ) { res = lookup(id); send Outcome[commit,res,id] to client; }
            catch ( transaction_aborted_exception or DB_connection_lost_exception ) { send Outcome[abort,nil,id] to client; }
} catch ( client_connection_lost_exception ) { return; }
} /* end ApplicationServer */
```

Figure 2: Application Server Behavior.

namely *aborted_transaction_exception* and *DB_connection_lost_exception*. These exceptions lead the application server to send back to the client stub an Outcome message with *abort* and a *nil* value for the result. This indicates that the transaction could not be committed either because the database server autonomously aborted it or because the connection to the database has broken while processing the transaction ([3]). Furthermore, if the *client_connection_lost_exception* is raised (this potentially occurs for any statement acting on the client connection, e.g., the sending of a message over that connection), the application server has no possibility to notify any Outcome message to the client, hence it simply skips performing any operation.

As a last observation, the retransmission logic at the client side allows achieving at-least once semantic for a well formed request issued by a client that does not crash for the realistic case in which a retransmitted request instance is eventually processed without experiencing any exception.

Coping with Client Crash. If we admit stable storage at the client side, our protocol can be simply extended to provide liveness even in the presence of client crashes. Specifically,

[3]As an optimization, the application server could retry the processing of the transaction a few times, in case of *aborted_transaction_exception* or *DB_connection_lost_exception*, before sending back the Outcome message to the client stub. This might improve performance by saving additional interactions between the client and the application server.

just before sending the first Request message to whichever application server, the client stub might log the request content and its identifier on stable storage (e.g., the disk), so that, any new client instance, activated after a crash, could use the logged identifier within the request retransmission logic. On the other hand, the request content and the identifier can be removed from the log after the result is delivered to the application.

Garbage Collection. The following mechanism could be coupled with the protocol to deal with garbage collection of unneeded recovery information from the database table supporting the testable transaction abstraction.

In case the client stub does not experience connection loss, then an acknowledgement message can be sent to the application server right after the result has been delivered to the client application. Upon the receipt of this message, the server simply discards the corresponding tuple (id, res) from the database table.

On the other hand, in case the client stub receives the result message after having performed multiple transmissions of the request (triggered by connection lost exceptions), a different type of acknowledgment message could be sent to the application server. Upon the receipt of this type of acknowledgment message, the server discards the result from the tuple (id, res), but maintains the request identifier. This is done to avoid that any transaction originated by a previously transmitted request instance, which is asynchronously processed by some application server, gets eventually committed (hence violating safety). We note that the result (e.g., an HTML page) typically takes up the most part of the storage required for the tuple (id, res). Hence discarding it means in practice freeing almost all the storage allocated for the recovery information associated with a given client request. Anyway, one can still rely on associating with each entry in that table an adequately selected Time-To-Leave (TTL) after which deletion of the request identifier id from the database can be performed.

4 Related Work and Discussion

Transaction Monitors or Object Transaction Services, such as OTS in CORBA or JTS in J2EE, address reliability by encapsulating the processing of the client request within an atomic transaction to be performed by the middle tier (application) server [8]. This solution does not address the loss of the result due, for example, to middle tier server crash. The work in [11] tackles this issue by encapsulating the storage of the result at the client (via browser handled cookies) within the atomic transaction. Differently from our approach, this solution includes the client within the transaction boundaries, and thus requires (and pays the price of) a distributed commit protocol, such as two-phase commit (2PC).

Several solutions based on the use of persistent queues have also been proposed in the literature [2, 3], 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). With this approach, the request message must be dequeued by the application server within the same transaction that manipulates application data and enqueues the response to the client. Compared to our proposal, this approach incurs higher overhead given that it requires the additional phase of storing the client request within the queue prior to processing the corresponding transaction. Also, in the case of a queuing system external to the database, the additional cost of a distributed commit protocol (e.g., 2PC) must be payed.

The works in [1, 14] address reliability in general multi-tier applications by employing interaction contracts between any two components, which specify permanent guarantees about state transitions, hence well fitting requirements of statefull middle tier applications. Interaction contracts are implemented by logging sources of non-determinism, e.g., exchanged messages, so to allow state reconstruction via a replay phase in case of failures. Differently from these proposals, our solution is oriented to (and reveals very simple and practical for) stateless middle tier servers, not requiring to be involved in bilateral contracts suited for the interaction among statefull parties.

Group communication has been used as a solution for masking failures in a three-tier system [4, 12, 13]. However the target of this approach is to provide reliable delivery of client requests at the middle tier, not to provide end-to-end reliability in case the middle tier interacts with a back-end database, which is instead the case addressed by our protocol.

The e-Transaction protocols in [6, 7] by Frølund and Guerraoui, use explicit coordination among the replicas of the application server during both normal behavior and fail-over. Since coordination is not required in our protocol, we avoid the coordination overhead and achieve higher scalability. This allows our proposal to better cope with high degree of replication and/or geographic distribution of the application servers (e.g., as in ADNs). Similar considerations can be made for what concerns the proposal in [18], where a primary server notifies to the backup replicas all the changes in its state before sending out any reply to the client. This solution also uses an agreement protocol to guarantee the consistency between the state of all the replicas and the database.

A third protocol by Frølund and Guerraoui [5] relies on the testable transaction abstraction (it logs the tuple (id,res) while processing the transaction at the back-end database). However, differently from our proposal, it handles fail-over through a "termination" phase, which is required since the employed testable transaction instance imposes no primary key

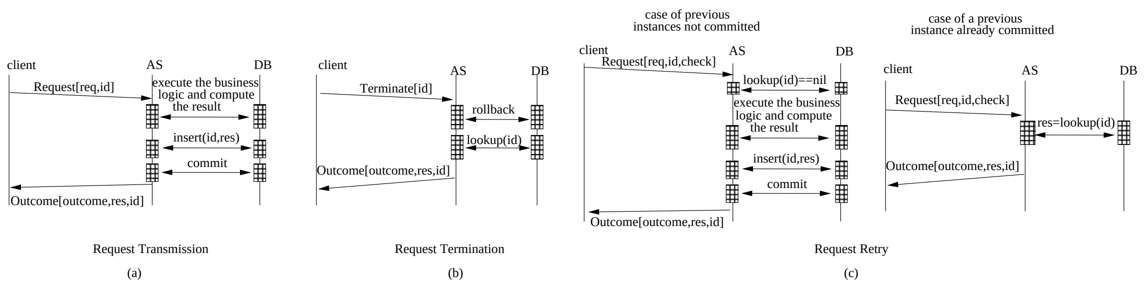

Figure 3: Basic Client-Initiated Interactions.

constraint on the request identifier. The termination phase discovers whether the transaction associated with the last issued client request was actually committed and exterminates it in the negative case. This is done by letting the client send terminate messages to the application server replicas which, in their turn, attempt a forced rollback operation of the corresponding transaction and then perform a lookup operation to retrieve the transaction result, if already established. In the negative case, an abort indication is returned to the client, which sends a new request message (with a different identifier) to whichever application server.

The avoidance of the termination phase allows our protocol to reduce the fail-over latency compared to [5], as it will be quantified in Section 5. Also, by admission of the same authors, the employment of such a phase requires the underlying infrastructure to adopt mechanisms for guaranteeing that a request message is always processed before the corresponding terminate messages (e.g., coordination among the application server replicas, or, as explicitly suggested by the authors, delayed processing of terminate messages at the application servers), with consequent negative effects on performance and/or scalability. This processing order constraint derives from that, according to the specifications of the XA standard interface for transaction demarcation, 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 [16] - state table on page 109). Hence, if a forced rollback request was processed before the corresponding transaction activation, this transaction could get eventually committed. On the other hand, upon the receipt of a reply indicating that the lookup phase returned negatively, the client might activate a new transaction, with a different identifier, which could also eventually get committed, thus leading to multiple updates at the database and violating safety.

5 Performance Analysis and Results

We concentrate on a quantitative comparison between our protocol and that by Frølund and Guerraoui in [5], namely the closest one to our proposal. We denote as $P_{positive\_lookup}$ the conditional probability that, in case fail-over is activated by the client stub due to a connection lost exception, the lookup phase handled by the application server during fail-over returns with an already established result for the transaction. Also, we denote as: (i) $T_{transaction}$ the expected time required to execute the whole transaction at the back-end database, including the insertion of the recovery information in the apposite database table. (ii) $T_{network}$ the average network delay for the round trip interactions between the client and the application server, and between the application server and the database server. (iii) $T_{trans\_rollback}$ the expected time required for handling a forced rollback request for a transaction. (iv) T_{lookup} the expected time for performing a lookup operation in the table maintaining the recovery information.

We can now derive the expressions for the expected latencies of (i) a request transmission interaction (see Figure 3.a), proper of the two protocols, (ii) a request termination interaction (see Figure 3.b), proper of the protocol in [5] and (iii) a request retry interaction including the *check* parameter (see Figure 3.c), proper of our protocol. For simplicity, we consider the case of transactional logic (including the insertion of the recovery information) executed through a single round trip interaction between application and database servers, e.g., as in stored procedures (this allows avoiding the introduction of an arbitrary delay in the latency models, caused by an arbitrary number of interactions for the management of the transactional logic). Also, with no loss of fairness, we avoid to model transaction aborts due to autonomous decisions of the database server. The expressions are:

$$T_{req} = T_{network} + T_{transaction} \tag{1}$$

$$T_{term} = T_{network} + T_{trans\_rollback} + T_{lookup} \tag{2}$$

$$T_{retry} = T_{network} + T_{lookup} + (1 - P_{positive\_lookup})T_{transaction} \tag{3}$$

Actually, no processing delay of terminate messages has been introduced within the request termination interaction, which, as suggested by the authors, is required by the protocol in [5] to ensure safety. This even favors the protocol in [5] in the comparative analysis.

To build complete end-to-end latency models for the two protocols we need to consider the probability P_{conn_lost} that a connection lost exception is raised at the client side during any of the basic client-initiated interaction. Given that connection lost exceptions are typically triggered on the basis of a timeout mechanism, we can model the client experienced delay associated with this type of exception through a timeout delay TO. Hence, we can express the expected end-to-end latency for the considered protocols as follows:

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

$$T^{our\_protocol} = (1 - P_{conn\_lost})T_{req} + P_{conn\_lost}(TO + T^{our\_protocol}_{failover}) \tag{5}$$

where $T^{fg}_{failover}$ and $T^{our\_protocol}_{failover}$ represent the expected fail-over latencies of the two protocols. The protocol in [5] lets the client activate successive request termination interactions (triggered by connection lost exceptions at the client side) 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 (also in this case triggered by connection lost exceptions at the client side) 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:

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

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

For what concerns the treatment of $T_{transaction}$, $T_{trans\_rollback}$ and T_{lookup} in the performance study, we have developed prototype implementations of modules for the manipulation of the recovery information, and of two transaction profiles specified by the TPC BENCHMARK^{TM} C [17], namely the New-Order Transaction (NOT) and the Payment Transaction (PT). These profiles portray the activities of a wholesale supplier and are representative, respectively, of a mid-weight and of a light-weight read-write transaction. In Table 1, we report for the two transaction profiles the costs measured by running DB2/UDB v8.1 on top of Windows 2003 Server on a multi-processor machine equipped with 4 Xeon 2.2 GHz, 4 GB of RAM and 2 SCSI disks in RAID-0 configuration. The application logic was implemented with stored procedure technology using JDBC. Each reported value is the average over a number of samples that ensures a confidence interval of 10% around the mean at the 95% confidence level. (For both transaction profiles, the cost to insert the recovery information is less than 5% of $T_{transaction}$, which demonstrates the minimal overhead imposed by our protocol to support reliability.) For what concerns P_{conn_lost}, we selected the reasonable value of 0.01, reflecting the fact that, in real life experience, a very limited percentage of connections get lost. Finally, we set $P_{positive\_lookup}$ to the value of 0.5, indicating that, during fail-over, we equally likely find the transaction already committed or uncommitted. For what concerns network delays, we consider the following two classical scenarios. **Scenario-A**: the application is supported by a geographical infrastructure possibly layered on public networks over the Internet. **Scenario-B**: the application is supported by a LAN, as in the case of Intranet applications, or by a geographical infrastructure with low delivery latency, e.g., a (private) dedicated WAN. Table 1 also shows corresponding representative values we have selected for $T_{network}$ [10] (recall this parameter accounts for two round trip latencies). Finally, for each scenario we leave the value of TO as the independent parameter of the performance study. This allows us to compare the two protocols when considering settings with different features for the variance of the end-to-end interaction latency. Specifically, lower values for TO are representative of settings with highly predictable performance (e.g., a closed system with a limited number of clients),

$T_{transaction}$ (PT)	$T_{transaction}$ (NOT)	T_{lookup}	$T_{trans\_rollback}$	$T_{network}$ **(Scenario-A)**	$T_{network}$ **(Scenario-B)**
28.93	125.44	1.45	0.10	300	10

Table 1: Parameter Values (in msecs).

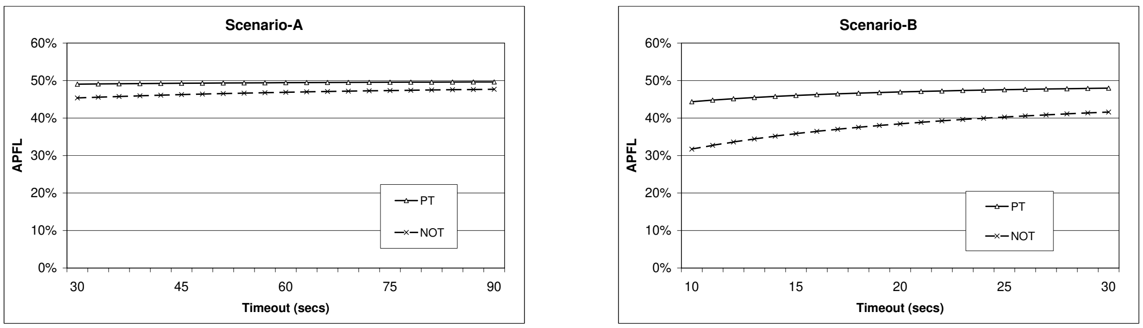

Figure 4: APFL Values for the Two Considered Scenarios.

for which suspects of failures can be reasonably triggered on the basis of relatively aggressive timeouts. On the other hand, longer timeout values are representative of situations with much larger fluctuations (and hence variance) for the end-to-end response time (as in systems layered on top of best effort public infrastructures), for which the timeout values need to be more conservative in order not to incur excessive false failure suspicions. We plot in Figure 4 the Additional Percentage of Fail-over Latency (APFL) of the protocol in [5], compared to our proposal, expressed as APFL $= \frac{T^{fg}_{failover} - T^{our\_protocol}_{failover}}{T^{our\_protocol}_{failover}}$. This parameter is representative of the different responsiveness of the two protocols while handling fail-over. Looking at the results, we note that our protocol provides fail-over latency which is between the 30% and the 50% lower than the one of the protocol in [5]. Also, for both the scenarios, the APFL curve is almost flat vs the selected timeout values. This points out how our proposal can be adopted to provide more responsive fail-over for the case of both aggressive timeout (suited for the case of predictable system performance, i.e., low response time variance) and conservative timeout (suited for systems with higher response time variance).

References

[1] R. Barga, D. Lomet, G. Shegalov, and G. Weikum. Recovery guarantees for internet applications. *ACM Transactions on Interernet Technology*, 4(3):289–328, 2004.

[2] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, San Francisco, 1997.

[3] 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. In *Proc. of the 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 42–53. ACM Press, 1995.

[4] P. A. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. In *Proc. of the 15th Symposium on Reliable Distributed Systems*, pages 150–161, Los Alamitos, Ca., USA, Oct. 1996. IEEE Computer Society Press.

[5] S. Frølund and R. Guerraoui. A pragmatic implementation of e-transactions. In *Proc. of the 19th Symposium on Reliable Distributed Systems*, pages 186–195. IEEE Computer Society Press, 2000.

[6] S. Frølund and R. Guerraoui. Implementing E-transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146, feb 2001.

[7] S. Frølund and R. Guerraoui. e-Transactions: End-to-end reliability for three-tier architectures. *IEEE Transactions on Software Engineering*, 28(4):378–395, 2002.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1991.

[9] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In S. S. S. Krakowiak, editor, *Advances in Distributed Systems*, number LNCS 1752, pages 33–47. Spinger Verlang, Berlin Heidelberg, 2000.

[10] Internet Traffic Report. http://www.internettrafficreport.com.

[11] M. Little and S. Shrivastava. Integrating the object transaction service with the Web. In *Proc. of the 2nd Int. Workshop on Enterprise Distributed Object Computing*, pages 194–205. IEEE Computer Society Press, 1998.

[12] S. Maffeis. Adding group communication and fault-tolerance to CORBA. In *USENIX Conference on Object-Oriented Technologies*, 1995.

[13] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol interface to provide CORBA with fault tolerance. In *Proc. of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*, pages 81–90. The USENIX Association, 1997.

[14] G. Shegalov, G. Weikum, R. Barga, and D. Lomet. EOS: Exactly-Once E-Service middleware. In *Proc. of the 28th Conference on Very Large Databases*, pages 1043–1046, Hong Kong SAR, China, 2002. VLDB Endowment, Morgan Kaufmann.

[15] The Jakarta Project. Jakarta commons: HTTP client. In *jakarta.apache.org*.

[16] The Open Group. *Distributed TP: The XA+ Specification Version 2*. 1994.

[17] Transaction Processing Performance Council. *TPC Benchmark C, Standard Specification, Revision 5.1*. 2002.

[18] H. Wu, B. Kemme, and V. Maverick. Eager replication for stateful J2EE servers. In *Proc. of the CoopIS, DOA, and ODBASE, OTM Confederated International Conferences*, pages 1376–1394. LNCS 3291, Springer Verlang, 2004.