Reliability in Three-Tier Systems without Application Server Coordination and Persistent Message Queues

Francesco Quaglia
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza"
quaglia@dis.uniroma1.it

Paolo Romano
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza"
paolo.romano@dis.uniroma1.it

ABSTRACT

When dealing with fault tolerance in three-tier systems, two major problems need to be addressed, that is how to prevent duplicate transaction executions when classical timeout based retransmission logics are employed, and how to ensure the agreement among the back-end databases despite failures (a transaction needs to be aborted or committed at all the involved databases independently of the failure scenario). In this paper we address these problems by proposing a fault tolerant protocol that, unlike previous solutions, (i) avoids the additional phase of storing the client request into a persistent message queue and (ii) avoids explicit coordination of middle tier application servers (during both normal behavior and fail-over). Our protocol reduces therefore the overhead imposed on the end-to-end interaction, thus improving user perceived responsiveness, and provides better scalability.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*fault tolerance*; H.2.4 [**Information Systems**]: Database Management—*distributed databases*

General Terms

Performance, reliability

Keywords

Distributed protocols, transaction processing

1. INTRODUCTION

When dealing with fault tolerance in three-tier systems, two major problems need to be addressed. The first one is how to prevent duplicate transaction executions when classical timeout based retransmission logics are employed as simple and pragmatical approach to perform failure detection and activate fail-over. The second one is how to ensure the agreement among the back-end databases despite failures, which means that a transaction needs to be aborted or committed at all the involved databases independently of the failure scenario.

A classical solution to address these problems [1] consists in inserting the client request into a persistent message queue before performing any other operation. The request is then dequeued within the same distributed transaction that manipulates application data and inserts the result of the manipulation into the persistent message queue. Actually, the persistent queue is used as a filter to avoid multiple updates of the back-end databases in case of client request retransmissions ([1]). Also, agreement assurance among the back-end databases relies on the additional mechanism of making the transaction coordinator decision (commit/abort) available at all the application server replicas before notifying the decision itself to the databases (this allows application servers to safely perform fail-over of each other, e.g., by aborting a distributed transaction and then reprocessing the corresponding request instance from the queue). The main problem with this approach is that the additional interactions between the application server and the queuing system plus the need for having all the replicas aware of decisions on distributed transactions, impose overhead that negatively affects user perceived latency, especially in the case of geographical scale distribution of the application servers, like in recent Web infrastructures offered by Akamai or Edgix referred to as Application Delivery Networks (ADNs) ([2]).

More recent solutions proposed in [4, 5] are based on primary-backup or asynchronous replication of the application servers. With these approaches, an application server needs to notify to the replicas (i) the client request identifier, before performing any further operation, and (ii) the decision on the distributed transaction, before sending it to

[1]This is because the attempt to insert the client request multiple times into the persistent message queue fails either because (i) a previously enqueued request with the same identifier is found or because (ii) the transaction result associated with that same identifier is found.

[2]In this type of infrastructures, the queuing system is typically not replicated at the different application (edge) servers to avoid the excessive overhead required to ensure the replicas coherency. Hence interactions between the application server and the queuing system, or among different application servers to mutually notify commit/abort decisions, being remote, are costly.

the databases (in the solution in [4], notification to the replicas takes place through a so called write-once register, i.e. a consensus-like abstraction). This is done in order to make all the other application servers aware of the existence of the request, or of the already established decision, so that: (a) Any other request instance with the same identifier (possibly retransmitted by the client to whichever application server), can be discarded. (b) It can be ensured that any already established decision is not subverted, thus preserving the agreement among the databases. The main problem with these solutions is that they require explicit coordination among the replicas of the application server, which imposes overhead and reduces system scalability.

In this paper we propose an innovative fault tolerant protocol that: (i) Does not make use of persistent message queues, thus avoiding the additional interactions between the application servers and the queuing system, which helps reducing the user perceived latency. (ii) Does not prescribe any form of explicit coordination among middle tier application servers, which provides scalability and, hence, the possibility to efficiently cope with highly replicated application servers distributed on a geographical scale (e.g. like in ADNs).

The key idea of our proposal is to store information concerning the transaction processing state (ITP for short - Information on Transaction Processing) across the distributed database servers participating in the transaction. The ITP is exploited by the replicated application servers to guarantee that (a) no more than one update is performed on the databases for each client request, despite possible retransmissions, and that (b) no two back-end database servers disagree on the final outcome of a distributed transaction. Actually, the idea of storing recovery information at the data layer for reliability purposes has been already exploited in [3, 8]. However, the main difference with our approach is that the proposals in [3, 8] consider the case of a single back-end database server. Instead we address the more complex and general case of transactions that are striped across multiple database servers. Therefore we need to face the additional problem of enforcing the agreement among multiple transactional resources.

The remainder of the paper is structured as follows. In Section 2 we present the three-tier system model we consider. Section 3 is devoted to the description of our protocol. Finally, Section 4 presents a comparative performance study of our protocol with other solutions.

2. SYSTEM MODEL

We consider a classical asynchronous distributed system, in which processes can fail according to the crash-failure model. Like in [3, 4, 5], we assume the system architecture to adhere to the three-tier paradigm.

Application servers are stateless in the sense that they do not maintain states across request invocations ([3]). Application servers collect request messages from the clients and drive updates over a set of distributed database servers within the boundaries of a global transaction [2]. For presentation simplicity, but without loss of generality, we assume that every transaction is executed over the same set of database servers. Application servers have a primitive `compute`, which embeds the transactional logic for the interaction with the databases. 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 databases inside a distributed transaction that is left uncommitted, therefore the changes applied to data are not made permanent as long as the databases do not decide positively on the outcome of the transaction. The result value returned by the primitive `compute`, which is assumed to be non-deterministic, represents the output of the execution of the transactional logic, which must be communicated to the client.

Database servers in the back-end tier eventually recover after a crash. A database server is viewed as a stateful, autonomous resource that offers the XA interface [10]. This interface provides transaction commitment functionalities that we model through the `xa_prepare` and `xa_decide` primitives. `xa_prepare` takes a transaction identifier in input and returns a value in the domain $Vote = \{yes, no\}$. A *yes* vote implies that the database server is able to commit the transaction (i.e. the transaction is pre-committed at that database), whereas a *no* vote is returned when the database server is unable to commit the transaction (i.e. it is aborted at that database). The `xa_decide` primitive takes in input a transaction identifier and a decision in the domain *Decision*=\{*commit*, *abort*\} and returns a value in the domain *Outcome*=\{*commit*, *abort*, *unknown_tid*\}. The *unknown_tid* value is an error code reported when an unknown transaction identifier is passed in input, i.e. the database server attempts to decide on an unknown transaction. (We recall that, according to the XA specification, a database server is allowed to forget about a transaction identifier once the transaction is either committed or aborted.) `xa_decide` returns *commit* if the database server voted *yes* for that transaction and *commit* is passed in input. Otherwise the transaction is aborted and `xa_decide` returns the value *abort*.

Each database server stores some recovery information, namely the ITP (Information on Transaction Processing), which is used to determine the processing state of a given transaction. The ITP consists of (i) an identifier associated with the transaction, (ii) the transaction result (i.e. the output of the `compute` primitive executed by the application server), and (iii) a value identifying one of the following states for the transaction: **Prepared** - the transaction has been pre-committed at that database; **Commit** - the transaction has been committed at that database; **Abort** - the transaction has been aborted, or needs to be aborted, at that database.

The ITP is recorded and accessed through the primitives `insert`, `overwrite` and `lookup`. `insert` takes three input parameters, an identifier for the transaction, a value in the domain $\{prepared, abort\}$ and a result, and records them

[3]There exist some solutions addressing reliability for the case of stateful application servers, e.g. [7, 9]. These solutions are not treated or discussed in detail in this paper simply because they deal with a different kind of system organization. However, as pointed out in [4], having stateless application servers is an important aspect of three-tier applications for the following main reasons: (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 (which means that we can freely migrate them), and do not have client affinity (which means that a client can be easily redirected to a different replica).

```
Class Client {
  CircularList ASList={$AS_1$,$AS_2$,..,$AS_n$}; ApplicationServer AS=$AS_1$;

  Result issue(Request req) {
    Outcome outcome=$abort$; Identifier id; Result res;
    while (outcome==$abort$) {
            set a new value for id;
            send[Request,req,id] to AS;
            set TIMEOUT;
            wait receive[Outcome,res,outcome,id] or TIMEOUT;
            if (TIMEOUT) (res,outcome)=this.terminate(id);
    } /* end while */
    return res;
  } /* end issue */

  (Result,Outcome) terminate(Identifier id) {
    while ($true$) {
            AS=ASList.next();
            send[Terminate,id] to AS;
            set TIMEOUT;
            wait receive[Outcome,res,outcome,id] or TIMEOUT;
            if (received [Outcome,res,outcome,id]) return(res,outcome);
    } /* end while */
  } /* end terminate */
}
```

Figure 1: Client Behavior.

(i.e. inserts the corresponding tuple) within a database table. This primitive is used to mark the state of the transaction within the ITP as *prepared* or *abort*. We assume the transaction identifier to be a primary key for that database table, therefore, any attempt to insert the previous tuple within the database multiple times is rejected by the database itself, which is able to notify the rejection event by rising an exception. `overwrite` takes in input two parameters, namely a transaction identifier, and a value in the domain $\{commit, abort\}$, and is used to set the state maintained by the ITP associated with that transaction to *commit* or *abort*. Finally, the `lookup` primitive takes in input a single parameter, namely a transaction identifier, and is used to retrieve from the ITP the state and the result associated with that transaction.

3. THE PROTOCOL

Client Behavior. Figure 1 shows the pseudo-code defining the client behavior. Within the method `issue`, the client generates an identifier associated with the request and sends the request to an application server, together with the identifier. It then waits for the reply, namely for an Outcome message indicating the outcome of the transaction. In case the outcome is *commit*, `issue` simply returns the result of the transaction.

In case of timeout expiration after the transmission of the request, the client invokes the `terminate` method, which attempts to force the abort of the transaction associated with that request. Within this method, the client keeps on retransmitting Terminate messages to the application servers on the basis of a timeout mechanism, until an outcome is returned via an Outcome message indicating that the transaction was either aborted or committed. If the outcome is *abort*, the client chooses a new identifier and simply retransmits the request.

Application Server Behavior. The application server behavior is shown in Figure 2. If a Request message arrives, then `compute` is invoked to start the distributed transaction. Next, the application server invokes the `prepare` method. While executing this method, the application server sends Prepare messages to every database server enlisted in the transaction. These messages are periodically resent on the basis of a timeout mechanism to all the database servers that did not respond with a Vote message before the timeout expiration (given that database servers eventually recover after

```
Class ApplicationServer {
  List dbList={$DB_1$, $DB_2$, . . . , $DB_m$};

  void main() {
   Result res=$nil$; Outcome outcome;
   while ($true$) {
    cobegin
      || wait receive[Request,req,id] from client;
         res=compute(req,id);
         if ( this.prepare(id,res)==$abort$ )
            {res=$nil$; outcome=$abort$; this.decide(id,$abort$);}
         else {outcome=$commit$; this.decide(id,$commit$);}
         send [Outcome,res,outcome,id] to client;
      || wait receive[Terminate,id] from client;
         (res,outcome)=this.resolve(id);
         send[Outcome,res,outcome,id] to client;
   } /* end while */
  } /* end main */

  (Result,Outcome) resolve(Identifier id) {
   Boolean abortFlag=$false$; List receiveList=$nil$;
   while (receiveList $\neq$ dbList){
    send[Resolve,id] to (dbList - receiveList);
    set TIMEOUT;
    repeat {
     wait until ((receive [Status,id,status,result] from any $DB_i$) or TIMEOUT);
     if (received[Status,id,status,result] from $DB_i$) {
       receiveList = receiveList + {$DB_i$};
       if (status==$abort$) abortFlag = $true$;
       }
     } until (TIMEOUT or (receiveList == dbList));
   } /* end while */
   if (abortFlag == $true$) {this.decide(id,$abort$); return ($nil$,$abort$);}
   this.decide(id,$commit$);
   return (result,$commit$);
  } /* end resolve */

  Status prepare(Identifier id, Result result){
   List receiveList=$nil$;
   while (receiveList $\neq$ dbList){
    send [Prepare,id,result] to (dbList - receiveList);
    set TIMEOUT;
    repeat {
     wait until ((receive [Vote,id,vote] from any $DB_i$) or TIMEOUT);
     if (received [Vote,id,vote] from $DB_i$) receiveList=receiveList+{$DB_i$};
    } until (TIMEOUT or (receiveList == dbList));
   } /* end while */
   if received ([Vote,id,yes] from every $DB_i$ in dbList ) return $prepared$;
   else return $abort$;
  } /* end prepare */

  void decide(Identifier id, Outcome decision) {
   List receiveList=$nil$;
   while (receiveList $\neq$ dbList){
    send [Decide,id,decision] to (dbList - receiveList);
    set TIMEOUT;
    repeat {
     wait until ((receive [Outcome,id,outcome] from any $DB_i$) or TIMEOUT);
     if (received[Outcome,id,outcome] from $DB_i$) receiveList=receiveList+{$DB_i$};
    } until (TIMEOUT or (receiveList == dbList));
   }/* end while */
  } /* end decide */
}
```

Figure 2: Application Server Behavior.

a crash, they eventually reply to these messages). In case an unanimous positive vote is collected (i.e. every database server voted *yes*), a Decide message carrying a *commit* decision is sent to all the database servers through the `decide` method. If any *no* vote is received, the whole transaction has to be aborted. In this case, Decide messages indicating the intention to abort the transaction are sent to the database servers. To ensure the termination of the commit protocol, Decide messages are retransmitted, again on the basis of a timeout mechanism, until an Outcome message is received from every database server (database servers eventually reply to Decide messages since, as recalled above, they eventually recover after a crash). Once the interaction with database servers is concluded, an Outcome message is sent back to the client with the transaction outcome (*commit* or *abort*) and the result.

Upon the receipt of a Terminate message possibly sent by the client during fail-over, the application server invokes the `resolve` method to determine the final outcome of a given transaction possibly activated by a different application server due to a Request message received from the client. Within the `resolve` method, the application server collects the state of the transaction logged by the ITP maintained by every database server. Specifically, it sends Resolve messages to the database servers and waits for Status messages from all of them. Also in this case we use a timeout based retransmission logic. If every database server responds with a Status message carrying either a *prepared* or a *commit* value, the application server exploits the `decide` method to commit the transaction on those sites where it is prepared, but still uncommitted (e.g. due to crash of the application server originally taking care of it). Otherwise, that same method is used to abort the transaction at all the databases. Finally, the transaction outcome and the result are sent to the client.

Database Server Behavior. Figure 3 shows the behavior of the database server. To help the reader we present the pseudo-code first by introducing the main execution paths, associated with the receipt of different type of messages. Then we provide additional explanations.

```
Class DatabaseServer {
  Result result; Status status; Outcome outcome;

  void main(){
   while (true) {
    cobegin
     || wait receive [Prepare,id,result] from application server;
        if (xa_prepare(id)==yes) vote=this.prepare(id,result); else vote=no;
        send [Vote,id,vote] to application server;
     || wait receive [Decide,id,decision] from application server;
        outcome=xa_decide(id,decision);
        if(outcome==commit)
          {send [Outcome,id,commit] to application server; overwrite(id,commit);}
        if(outcome==abort)
          {send [Outcome,id,abort] to application server; overwrite(id,abort);}
        if(outcome==unknown_tid)
          {send [Outcome,id,decision] to application server; overwrite(id,decision);}
     || wait receive [Resolve,id] from application server;
        (status,result)=this.try_abort(id);
        send[Status,id,status,result] to application server;
   } /* end while */
  } /* end main */

  Vote prepare(Identifier id, Result result) {
   try {insert(id,prepared,result); return yes;}
   catch (DuplicatePrimaryKeyException ex) {
    if (lookup(id).status == abort) {xa_decide(id,abort); return no;}
    else return yes;
   }
  } /* end prepare */

  (Status,Result) try_abort(Identifier id){
   try {insert(id,abort,nil); xa_decide(id,abort); return (abort,nil);}
   catch (DuplicatePrimaryKeyException ex) {return lookup(id);}
  } /* end try_abort */
}
```

Figure 3: Database Server Behavior.

Prepare **message in input (A).** In this case, the database server invokes the `xa_prepare` primitive in the attempt to pre-commit the transaction. If this primitive returns *yes*, the database server invokes the `prepare` method to insert the ITP associated with the transaction, with *prepared* as the state value. Then the vote is sent back to the application server (through a Vote message).

Decide **message in input (B).** In this case, the database server invokes the `xa_decide` primitive to take a final decision for the transaction. Then, through the `overwrite` primitive, the state of the transaction maintained by the corresponding ITP is overwritten with the value *commit*/*abort*. Afterwards, the Outcome message is sent back to the application server.

Resolve **message in input (C).** In this case, the database server invokes the `try_abort` method attempting to insert the ITP associated with the transaction, with *abort* as the value for the transaction state. If the latter operation succeeds, the transaction is aborted through `xa_decide`, and a Status message is sent back to the application server with the *abort* value for the state of the transaction. If the insertion of the ITP fails (we recall that we have assumed the transaction identifier to be a primary key - see Section 2 - therefore at most one insertion of the ITP associated with a given transaction can occur), it means that the database already keeps the ITP associated with the transaction. In this case, the transaction state maintained by the ITP is retrieved through the `lookup` primitive, and is sent back to the application server via a Status message.

There is a main point we have left pending while describing the execution path (A). Specifically, the attempt to insert the ITP, with the *prepared* value for the transaction state, might fail by raising a primary key exception. (As said, the same might happen for the execution path (C) when attempting to insert the ITP with an *abort* value for the transaction state.) This is the mechanism we employ for ensuring the agreement on the outcome of the distributed transaction despite failures or suspect of failures. Specifically, given that:

1. A database server decides to commit a transaction, i.e. receives a Decide message with the *commit* indication, only if one application server has collected Vote messages with *yes* or Status messages with *prepared* (or *commit*) from all the database servers. Note that this implies that all the database servers have performed a successful insertion of the ITP with the *prepared* value for the transaction state.
2. A database server decides to abort a transaction, i.e. receives a Decide message with the *abort* indication, only if one application server has collected a Vote message with *no* or a Status message with *abort* from at least one database server. Note that this implies that either (i) the insertion of the ITP with an *abort* value is successful on at least one database, or (ii) the insertion of the ITP with the *prepared* value for the transaction state fails on at least one database, or is not attempted at all due to the fact that `xa_prepare` returns *no* at that database.

then no two application servers (possibly performing fail-over of each other), can take a different decision on the outcome of a distributed transaction since the conditions enabling the send of a Decide message with the *commit* or *abort* indication are mutually exclusive. As a direct consequence, if a database server is asked to decide *commit* or *abort* for an unknown transaction (this is the case of the *unknown_tid* in path (B)), then it must have already taken the same decision it is currently asked to take ([4]). Therefore, it can simply send back to the application server a message specifying, for that transaction, the same outcome the application server has requested for, namely *commit* or *abort*. Then, the database server possibly updates the corresponding ITP to reflect that final outcome for the transaction.

Additionally, agreement on the outcome of a transaction among the database servers cannot be violated due to Resolve messages employed during fail-over. Specifically, after the insertion of the ITP with the *prepared* value in path (A), the database server will reject any successive insertion of the ITP, thus avoiding the abort of the transaction due to the receipt of Resolve messages activating the execution path (C).

[4]As pointed out in Section 2, the XA layer does not keep track of identifiers of already committed/aborted transactions.

Therefore, when a database server receives a Decide message to commit a transaction (by point 1, this implies that all the database servers have performed successful insertion of the ITP with the *prepared* value for the transaction state) it is sure that no database server will ever accept aborting that transaction due to Resolve messages possibly sent by any application server.

Our protocol avoids duplicate transactions since the client resubmits a new request only after it has received the Outcome message from an application server, carrying the *abort* indication for the transaction associated with the last issued request (otherwise it keeps on performing retransmission of Terminate messages). On the other hand, the application server returns to the client the Outcome message with the *abort* indication only after each database server either already recorded an ITP with the *abort* state or voted *no* for that transaction. As a consequence each time a new request instance is sent by the client, no previous request instance can give rise to a transaction that gets eventually committed since after the *abort* state is logged within the ITP, the database server rejects voting *yes* for that transaction. The same happens in case the database server already voted *no* since the `xa_prepare()` primitive does not recognize the transaction identifier.

Finally, to tackle failure situations in which pre-committed transactions remain pending indefinitely (e.g. simultaneous crash of both the client and the application server processing the client request), an additional type of process can be envisaged which periodically checks the presence of pre-committed transactions and resolves them through a logic similar to the one of the application server `resolve` method. This would allow ensuring data availability by unlocking data pre-committed by those transactions.

4. EVALUATION

In this section, we aim at quantitatively comparing the performance of our protocol against existing solutions. We carry out the comparison by providing simple, yet realistic, models for the response time of each protocol, and studying the output provided by the models while varying some system parameters. We are interested in the case of no data contention and light system load. This allows us to evaluate the impact of each protocol on the response time more accurately, since we avoid any interference due to overhead factors not directly related to the distributed management of transaction processing performed by the protocols themselves (e.g. the overhead caused by delay in the access to data within the database due to contention). Finally, we are interested in studying the case of normal behavior, i.e. when no process crashes. This is because failure free runs are the most likely to occur in practice, thus representing an adequate test-bed for the evaluation of the real performance effectiveness of any solution.

We compare our protocol with the following alternatives: **(1)** The persistent queue (PQ) approach [1], whose behavior is schematized in Figure 4.a. This approach performs the enqueuing of the client request as the first action. Next, it uses START and PRECOMMIT logs at the application server to guarantee the agreement on the distributed transaction outcome, despite failures. As already mentioned, this transaction also includes the enqueuing of the result of the data manipulation performed during the compute phase. **(2)** The primary-backup replication scheme (PBR) presented in [5]

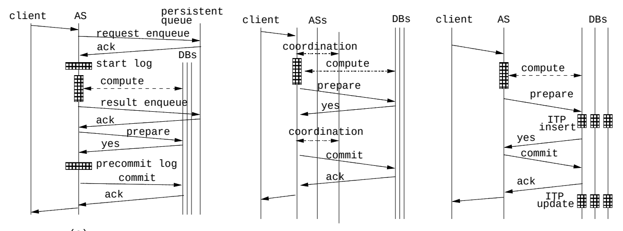

Figure 4: Behavior of the Compared Protocols.

and the asynchronous replication scheme (AR) presented in [4]. The behavior of both these protocols can be schematized as shown in Figure 4.b, where the COORDINATION phase represents either the activity of propagating recovery information (i.e. the client request identifier and the transaction result) from the primary application server to the backups, this holds for PBR, or the activity of updating the consensus object (i.e. the write-once register), this holds in case of AR.

For completeness, we also show (see Figure 4.c) the schematized behavior of our protocol, which performs (i) the insertion of the ITP with *prepared* state value (this is done before the database server sends out the Vote message) and (ii) the update of the state maintained by the ITP to the *commit* value (this is done after the database server sends out the Outcome message to the application server, hence not directly contributing to the end-to-end latency).

Response Time Models. Response times for the considered protocols, evaluated with no loss of generality at the application server side, can be expressed as:

$$\begin{aligned} T_{PQ} = T_{base} &+& T_{start} + T_{precommit} + 2 \times RTT_{as/qs} + \\ && + \, T_{req-enqueue} + T_{res-enqueue} \quad (1) \\ T_{PBR} = T_{AR} &=& T_{base} + 2 \times T_{coordination} \quad (2) \\ T_{our\_protocol} &=& T_{base} + T_{ITP\_insert} \quad (3) \end{aligned}$$

where (a) $RTT_{as/qs}$ is the round trip time between an application server and the queuing system used by PQ, (b) $T_{req-enqueue}$ (resp. $T_{res-enqueue}$) represents the time to enqueue the client request (resp. the transaction result) within that system and (c) T_{base} is a common term for all the protocols expressed as:

$$T_{base} = (T_{SQL} + T_{xa\_prepare} + T_{xa\_commit}) + 3 \times RTT_{as/db} \quad (4)$$

under the hypothesis that (i) the the back-end databases have the same computational capacity, that (ii) the round trip time between the application server and each back-end database $RTT_{as/db}$ is equal for all the databases ([5]) and that (iii) activation of functions in the XA interface and of the SQL associated with the transaction needs a single message exchange from the application server to each database server and the corresponding acknowledgment ([6]).

Parameter Treatment. Some parameters appearing in the latency models are left as independent variables in the performance study. They are: (i) $RTT_{as/db}$, dependent on the

[5]In case of heterogeneous databases and/or different round trip times with the application server, the expressions we propose are still representative when considering the maximum value, across all the databases, for the terms they contain.

[6]We model the case of transactional logic activated via a single message, e.g. like in stored procedures. This is done in order to avoid the introduction of an arbitrary delay in the response time models caused by an arbitrary number of message exchanges between application and database servers.

Table 1: Measured Parameter Values (Expressed in msec).

common parameters			PQ				our protocol	
T_{SQL}	$T_{xa\_prepare}$	T_{xa_decide}	T_{start}	$T_{precommit}$	$T_{req-enqueue}$	$T_{res-enqueue}$	T_{ITP_insert}	T_{ITP_update}
186.78	6.07	9.99	1.66	0.44	20.30	0.77	20.30	0.81

relative locations of application servers and database servers, (ii) $RTT_{as/qs}$, dependent on the relative locations of application servers and the persistent queuing system, and (iii) $T_{coordination}$, dependent on the specific algorithm selected for either the management of the update of backup application servers in PBR, or the management of the consensus object (i.e. the write-once register) in AR, and also on the communication speed among application server replicas.

Other parameters have been measured through prototype implementations of PQ and our protocol, relying on DB2 V8.1 and LINUX (kernel version 2.4.21). The prototype of PQ performs START and PRECOMMIT logs via operations on the file system, as in the approach commonly used by transaction monitors [2]. Also, as typically found in industrial environment, persistent message storing is supported through a database system, namely DB2 in our case. For what concerns our protocol, we have developed a prototype that maintains the ITP on a user level table managed via local transactions on DB2. In order to use a representative value for T_{SQL} in the comparative study, we have also implemented the New-Order-Transaction profile of the TPC BENCHMARK^{TM} C [11], and measured the latency for the related SQL operations. Table 1 lists obtained measures for the case of application server and database server both hosted by a Pentium IV 2.66GHz with 512GB RAM and a single UDMA100 disk. 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. As we are interested in the case of no data contention and light system load, all the measures have been taken for the case of requests submitted one at a time.

Comparison. While comparing the protocols we have set $T_{coordination} = RTT_{as/qs} = RTT_{as/db}$ ([7]) and have varied the value of these parameters between 10 and 250 msec. This allows capturing different system settings for what concerns the communication speed among processes, ranging from configurations such as LANs or WANs with low/controlled message delivery latency, to geographical infrastructures deployed on public networks over the Internet, exhibiting higher communication delay.

If the round trip time among processes remains below 25 msecs, the latency for the considered protocols is nearly the same. On the other hand, as soon as that round trip time increases, the performance gap between our protocol and the other solutions rapidly grows. As an extreme, when the round trip time reaches 250 msec, the solutions in [1, 4, 5] exhibit response times about 50% higher than our protocol. This outlines the effectiveness of our proposal for generic system settings.

[7]In absence of faults, a round of messages is the lower bound on the message complexity required for transmission and acknowledgement of recovery information between the primary and the backup in the PBR solution, and for achieving consensus [6], i.e. for the management of the consensus object in AR. Hence, in normal behavior, the coordination latency can be modeled as a round trip delay.

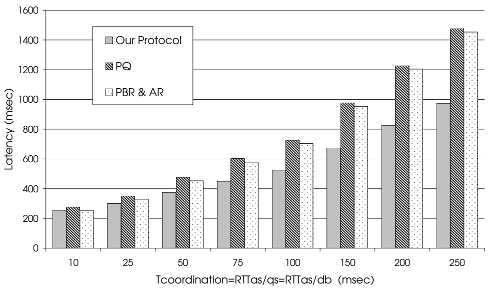

Figure 5: Protocol Latencies.

5. REFERENCES

[1] P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proc. of the 19th ACM Int. Conference on the Management of Data (SIGMOD)*, pages 101–112, May 1990.

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

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

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

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

[6] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults. *SIGACT News*, 32(2):45–63, 2001.

[7] 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 (EDOC)*, pages 194–205. IEEE Computer Society Press, 1998.

[8] P. Romano, F. Quaglia, and B. Ciciani. Ensuring e-Transaction through a lightweight protocol for centralized back-end database. In *Proc. of the 2nd Int. Symposium on Parallel and Distributed Processing and Applications (ISPA)*, pages 903–913. LNCS, Springer-Verlang, 2004.

[9] 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 (VLDB)*, pages 1043–1046. Morgan Kaufmann, 2002.

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

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