This article appeared in IEEE Transactions on Parallel and Distributed Systems
vol.18, no.3, pp.364-378, 2007

Ensuring e-Transaction with Asynchronous and Uncoordinated Application Server Replicas*

Francesco Quaglia and Paolo Romano
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza", Italy
Via Salaria 113, 00198 Roma, Italy
{quaglia,romanop}@dis.uniroma1.it

Abstract

A recently proposed abstraction, called e-Transaction (*exactly-once Transaction*), specifies a set of properties capturing end-to-end reliability aspects for three-tier Web-based systems. In this paper we propose a distributed protocol ensuring the e-Transaction properties for the general case of multiple, autonomous back-end databases. The key idea underlying our proposal consists in distributing, across the back-end tier, some recovery information reflecting the transaction processing state. This information is manipulated at low cost via local operations at the database side, with no need for any form of coordination among asynchronous replicas of the application server within the middle-tier. Compared to existing solutions, our protocol has therefore the distinguishing features of being both very light and highly scalable. The latter aspect makes our proposal particularly attractive for the case of very high degree of replication of the application access point, with distribution of the replicas within infrastructures geographically spread on public networks over the Internet (e.g. Application Delivery Networks), namely a configuration that also provides the additional advantages of reduced user perceived latency and increased system availability.

Index-Terms: Web-based Transactional Systems, Reliability, Three-tier Systems, Asynchronous Replication.

1 Introduction

Since the past decade, browsers have become the access point to complex transactional applications. Following the approach widely adopted in the industrial world, these applications are typically supported by multi-tier systems, e.g. three-tier systems, where middle-tier Web/application servers have

*An earlier version of this paper [25] appeared in *Proc. of the 20th ACM Symposium on Applied Computing*, 2005.

the responsibility to interact with back-end databases on behalf of the client (e.g. an applet running in a browser). This partitioning provides high modularity and flexibility. However, the multiplicity of the employed components, and their interdependencies, make it not trivial to achieve any meaningful form of reliability [12].

Actually, reliability issues need to be addressed in different ways depending on whether middle-tier servers supporting the application are stateful or stateless entities (as recently discussed in [2] it is in fact not clear how to adapt solutions tailored for one case to the other one). In the former case, we need to deal with consistency of the application server state and also mutual consistency of the application and the back-end database servers' states in the presence of failures. In the latter one the attention is instead devoted only to the consistency of back-end databases, in terms of atomicity of the distributed transaction and avoidance of transaction duplication in case of failures.

For three-tier systems with stateless application servers, Frolund and Guerraoui have proposed the *e-Transaction* (exactly-once Transaction) abstraction as a desirable set of reliability properties [13]. The e-Transaction is specified through seven properties belonging to three categories: *Termination*, *Agreement* and *Validity*. Termination properties ensure the liveness of the client-initiated interaction from a twofold prospective: not only it is guaranteed that a client does not remain indefinitely blocked waiting for a response, but also that no database server maintains pre-committed data locked for an arbitrarily long time interval. The latter requirement is clearly very important because the availability of data (and consequently of the whole application) may be compromised if a database server that has endorsed the responsibility to commit a transaction (i.e. the transaction is in the pre-commit state), would not timely abort or commit it. Agreement embodies the safety properties of the system, ensuring both atomicity and consistency of the distributed transaction, and at-most once semantic for the processing of client requests. The latter aspect is very frequently poorly tackled by popular Web sites which simply warn users not to hit the check-out/commit button twice when a long delay occurs, so to prevent that multiple updates are performed on the back-end databases. These expedients are clearly inadequate, especially for the case of commercial or financial applications where these issues may lead to a loss in revenue and possibly to legal or ethical implications. Finally, Validity restricts the space of possible results to exclude meaningless ones, e.g. where results are invented or transactions are committed even though some database is unable to pre-commit them.

In this paper we are interested in the case of stateless application servers, and we propose an e-Transaction protocol suited for emerging Web infrastructures referred to as Application Delivery Networks (ADNs), like those provided by, e.g., Akamai or Edgix ([1]). These infrastructures are char-

[1] ADNs represent the 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.

acterized by both very high degree of replication and widespread diffusion of the application access point (i.e. of the application servers hosting the transactional logic) on a geographical scale. For this type of organization, it is desirable to have different access points offering fail-over capabilities to each other. However, this should be achieved in a scalable manner so to provide minimal overhead and low user perceived latency. This is exactly what our protocol does since it provides reliability guarantees without the use of any form of coordination among application server replicas (during both normal behavior and fail-over). Also, the protocol is designed for the general case of transactions spanning multiple, autonomous back-end databases, as in the case of multiple parties involved within a same business process.

The key idea of our proposal is to store recovery information concerning the transaction processing state (ITP for short - Information on Transaction Processing) across the back-end databases participating in the transaction. The ITP includes both classical information logged in standard two-phase commit (2PC) (such as the transaction state), plus additional information proper of our protocol (such as the identity of the application server processing the transaction), and allows our protocol to effectively deal with both transaction atomicity and idempotence, while preserving liveness. The ITP is manipulated through *local transactions*, which are independent of the global distributed transaction associated with the client request. These local transactions are executed by the database server in a transparent way for the application server, thus not requiring any additional interaction between application and database servers. This is just what allows our protocol to exhibit minimal overhead compared to a baseline approach that does not provide end-to-end reliability guarantees, as it will be shown by the results of a quantitative analysis we have carried out through parameterized performance models and an industry standard benchmark for On-Line-Transaction-Processing (OLTP) systems.

The remainder of this paper is structured as follows. In Section 2 we present the distributed system model we consider. Section 3 and Section 4 are devoted, respectively, to the description of our protocol and to the proof of its correctness with respect to e-Transaction properties. In Section 5, garbage collection of recovery information and other practical issues are discussed. Related work is addressed in Section 6. Finally, Section 7 presents a comparative performance study of our protocol with other solutions.

2 System Model

We consider a classical distributed, asynchronous system model, in which there is no bound on message delay, clock drift or process relative speed [9]. Process communication takes place exclusively through message exchange. Processes can fail according to the crash-failure model [15]. Communication channels are assumed to be reliable, therefore each message is eventually delivered unless either the sender or the receiver crashes during the transmission. In the following paragraphs we describe the main features of every class of processes in the system, i.e. clients, application servers and database servers.

2.1 Clients

Client processes do not directly communicate with database servers, they only interact with application servers. This takes place by invoking the method `issue`, which is used to activate the transactional logic on the application server. This method takes the client request content as the parameter and returns only upon receipt of a positive outcome ($commit$) for the corresponding transaction. The method returns the result of the transaction execution.

2.2 Application Servers

We consider a set of n application server processes $\{AS_1, \ldots, AS_n\}$. 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 [4]. For presentation simplicity we assume that every transaction is executed over the same set of database servers, and that this set includes all the system back-end databases. However, in Section 5 we discuss approaches for relaxing such an assumption in order to cope with applications performing transactional access to a subset of the back-end databases.

Application servers have no affinity for clients, and do not know each other existence. Moreover, they are stateless, in the sense that they do not maintain states across requests from clients, i.e. a request from a client can only determine changes in the state of the databases.

Application servers have a primitive `compute`, which embeds the non-idempotent 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` represents the output of the execution of the transactional logic at the databases, which must be communicated to the client. `compute` is non-deterministic as its result depends on the state of the databases, and (possibly) on other non-deterministic factors (such as the current state of some device). In other words, multiple invocations of this primitive may return different results. As in [11, 13], `compute` is assumed to be non-blocking, which means it eventually returns unless the application server crashes.

For simplicity, we do not consider chained invocation of the application servers. However, for what concerns reliability aspects, this does not result in any loss of generality since application servers are stateless. As a consequence, the crash of the single 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.

2.3 Database Servers

We assume a finite set of m autonomous database servers $\{DB_1, \ldots, DB_m\}$, each one possibly keeping a different data set. A database server is viewed as a stateful, autonomous resource that offers the XA interface [31] and recovers after a crash. In our model, we do not consider the whole set of functions of the XA API. Specifically, we are interested only in transaction commitment functionalities which we model through the `xa_prepare` and `xa_decide` primitives. `xa_prepare` takes a transaction identifier as 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 as 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 as input, i.e. the database server attempts to decide on an unknown transaction ([2]). `xa_decide` returns $commit$ if the database server voted yes for a transaction and $commit$ is passed as 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) the identifier of the transaction, (ii) the identifier of the application server that executes the transaction, (iii) the transaction result (i.e. the output of the `compute` primitive executed by the application server), and (iv) a value that allows the identification of one of the following states for the transaction:

1. **Prepared.** This value indicates that the transaction is pre-committed at that database.
2. **Commit.** This value indicates that the transaction has already been committed at that database.

[2] We recall that according to the XA specification a database server, i.e. a resource manager in the XA terminology, is allowed to forget about a transaction, namely about a transaction identifier, once the transaction is either committed or aborted.

3. **Abort.** This value indicates that the transaction was aborted, or needs to be aborted, at that database.

The ITP is accessed and manipulated within ACID transactions by means of `insert`, `overwrite` and `lookup` primitives, implemented, e.g., through SQL statements on a conventional database system ([3]). `insert` takes four input parameters, namely the identifiers of the transaction and of the application server executing it, a value in the domain $\{prepared, abort\}$ and a result, and records them (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 raising the `DuplicatePrimaryKeyException`. `overwrite` takes two parameters, namely the 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 identifier. More precisely, the transaction state is set to the value passed as second input parameter to the primitive. Both `insert` and `overwrite` encompass real transactional data manipulation, thus requiring eager disk access (the corresponding transactions are executed as independent top-level actions, i.e. not within the context of the transaction on whose behalf they are executing). The `lookup` primitive is used to retrieve from the ITP the state and the result associated with a certain transaction, together with the identity of the application server that executed the transaction. This primitive takes as input a single parameter, namely the transaction identifier. It returns the special value nil in case that transaction identifier has no corresponding ITP logged.

All the primitives available at the database server are assumed to be non-blocking, i.e. they eventually return unless the database server crashes after the invocation.

3 The protocol

3.1 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, selects an application server and sends the request

[3]An ACID transaction manipulating the ITP can be also supported efficiently through an ad-hoc implementation exploiting capabilities of file system API, thus possibly providing performance advantages over a transaction executed on top of the database system. Actually, we have decided to describe the protocol by relying on SQL for the manipulation of the ITP exclusively for simplicity of presentation. However, as we shall discuss in Section 7, we have also developed an ad-hoc prototype implementation just based on file system API.

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

  Result issue(RequestContent req) {
    Outcome outcome=abort; Identifier id; Result res;
    AS=aslist.next();
    while (outcome==abort) {
        set a new value for id;
        send [Request,req,id] to AS;
        wait receive [Outcome,res,outcome,id] or suspect(AS);
        if suspect(AS) (res,outcome)=terminate(id);
    }
    return res;
  } /* end issue */
  (Result,Outcome) terminate(Identifier id) {
    while (true) {
        AS=aslist.next();
        send [Terminate,id] to AS;
        wait receive [Outcome,res,outcome,id] or suspect(AS);
        if (received [Outcome,res,outcome,id]) return(res,outcome);
    }
  } /* end terminate */
}
```

Figure 1: Client Behavior.

to this server, together with the identifier. It then waits for the reply, namely for an Outcome message for the transaction ([4]). If the outcome is $commit$, `issue` simply returns the result of the transaction. If the outcome is $abort$, the client chooses a new identifier and re-transmits the request. Otherwise, in case the contacted application server is suspected to have crashed ([5]), the client invokes the `terminate` method. Within this method, the client keeps on re-transmitting Terminate messages to the application servers (application server crash suspicion is the trigger for the re-transmission), 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 re-transmits the request.

3.2 Application Server Behavior

The application server behavior is shown in Figure 2. It makes use of the `DTManager` class in Figure 3. Two execution paths are possible at the application server depending on the type of the message

[4]We indicate with **receive** a blocking statement returning upon message delivery at the application level. Also, the **received** statement immediately returns with boolean indication on whether a given message is already available at the application level.

[5]We use the classical **suspect** statement to abstract over details of the underlying failure detection mechanism.

```
Class Application Server {
  List dblist={DB_1, DB_2, ..., DB_m}; DTManager dt;

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

Figure 2: Application Server Behavior.

received from the client.

If a Request message is received, then `compute` is invoked to execute the distributed transaction. Next the application server starts the distributed commit protocol by invoking the `prepare` method of the `DTManager` class. While executing this method, the application server periodically re-transmits the Prepare message on a timeout basis to all the database servers until a Vote message is received from every database server. Then, the commit protocol goes on through the `decide` method of that same class. If an unanimous positive vote was collected (i.e. every database server voted yes), Decide messages with the $commit$ decision are sent to all the database servers. If any no vote was received, the whole transaction has to be aborted. In this case, Decide messages with the $abort$ decision are sent to the database servers. Decide messages are re-transmitted, again on the basis of a timeout mechanism, until an Outcome message is received from every database server. Once the interaction with database servers is concluded, an Outcome message is sent back to the client, carrying the outcome of the transaction ($commit$ or $abort$), together with the result.

A different behavior is triggered by the arrival of a Terminate message. In this case, the application server invokes the `resolve` method of the `DTManager` class 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 re-transmission logic if the database servers do not respond within a pre-specified

```
Class DTManager {
  List dblist={DB_1, DB_2, ..., DB_m};

  (Result,Outcome) resolve (Identifier id) {
      repeat {
          send [Resolve,id] to dblist;
          set TIMEOUT;
          wait until ( ( for every DB_k ∈ dblist: receive [Status,id,status,result] ) or TIMEOUT );
      } until ( received [Status,id,status,result] from every DB_k ∈ dblist );
      if (received [Status,id,abort,nil] from some DB_k ∈ dblist) {this.decide(id,abort); return (nil,abort); }
      else {this.decide(id,commit); return (result,commit);}
  } /* end resolve */
  Status prepare(Identifier id, Result result) {
      repeat {
          send [Prepare, id, result] to dblist;
          set TIMEOUT;
          wait until ( (for every DB_k ∈ dblist: receive [Vote,id,vote]) or TIMEOUT );
      } until ( received [Vote,id,vote] from every DB_k ∈ dblist );
      if ( received [Vote,id,yes] from every DB_k ∈ dblist ) return prepared;
      else return abort;
  } /* end prepare */
  void decide(Identifier id, Outcome decision){
      repeat {
          send [Decide, id, decision] to dblist;
          set TIMEOUT;
          wait until ( (for every DB_k ∈ dblist: receive [Outcome,id,outcome]) or TIMEOUT );
      } until (received [Outcome,id,outcome]) from every DB_k ∈ dblist );
  } /* end decide */
}
```

Figure 3: `DTManager` (Distributed Transaction Manager) Class.

time period. If at least one database server reports an $abort$ status for that transaction, the application server makes sure that the transaction is aborted everywhere. This is done by invoking the `decide` method with adequate values for the parameters. On the other hand, if every database server responds with a Status message carrying either a $prepared$ or a $commit$ value retrieved by the ITP associated with the transaction, the application server exploits again the `decide` method of the `DTManager` class to commit the transaction at those sites where it is prepared, but still uncommitted. Finally, the transaction outcome and the result reported by the databases are sent to the client.

3.3 Database Server Behavior

Figure 4 shows the pseudo-code defining the database server behavior. This server exploits the `DTManager` class in Figure 3 and also the `ITPLogger` class in Figure 5. The database server executes three tasks triggered by the receipt of different types of messages, and an additional background

```
Class Database Server {
  List dblist={DB_1, DB_2, ..., DB_n}; DTManager dt; ITPLogger itp; Result result; Status status; Outcome outcome;

  void main(){
   on recovery do {
     for every pre-committed transaction j: if ((itp.retrieve(j)==abort) or (itp.retrieve(j)==nil)) xa_decide(id,abort);
   }
   while (true) {
     cobegin
       || wait receive [Prepare,id,result] from AS_i; // Task 1
         if (xa_prepare(id)==yes) vote=itp.prepare(id,result,AS_i);
         else vote=no;
         send [Vote,id,vote] to AS_i;
       || wait receive [Decide,id,decision] from AS_i or DB_i; // Task 2
         outcome=xa_decide(id,decision);
         if (outcome==unknown_tid) {send [Outcome,id,decision] to AS_i or DB_i; itp.set(id,decision); }
         else {send [Outcome,id,outcome] to AS_i or DB_i; itp.set(id,outcome);}
       || wait receive [Resolve,id] from AS_i or DB_i; // Task 3
         (status,result)=itp.try_abort(id);
         send [Status,id,status,result] to AS_i or DB_i;
       || background: for every pre-committed transaction j such that itp.retrieve(j)==prepared: // Task 4
                   if ( suspect(itp.getAS(j)) ) dt.resolve(j);
   }
  } /* end main */
}
```

Figure 4: Database Server Behavior.

task.

Task 1: This task is activated upon receipt of a Prepare message from an application server. The database server invokes the `xa_prepare` primitive in the attempt to pre-commit the transaction. If this operation fails, a negative Vote is sent back to the application server. Conversely, if `xa_prepare` returns a yes vote, the `prepare` method of the `ITPLogger` class is invoked to insert the ITP associated with the transaction with the $prepared$ state value. If the insertion of the ITP fails, it means that the database already stores the ITP associated with the transaction ([6]). In this case, the transaction state maintained by the ITP is retrieved through the `lookup` primitive. If the state value is $abort$, a negative vote is sent back to the application server via a Vote message. Otherwise (i.e. the transaction is prepared and the ITP insertion with $prepared$ state value succeeds) a Vote message with yes is sent back to the application server.

[6] 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.

```
Class ITPLogger {
  Status status; Result res; ApplicationServer AS;

  Vote prepare(Identifier id, Result result, ApplicationServer AS) {
    try {insert(id,prepared,result,AS); return yes;}
    catch (DuplicatePrimaryKeyException ex) { if (lookup(id).status == abort) return no; else return yes; }
  } /* end prepare */
  (Status,Result) try_abort(Identifier id){
    try {insert(id,abort,nil,nil); xa_decide(id,abort); return (abort,nil);}
    catch (DuplicatePrimaryKeyException ex) {(status,res,AS)=lookup(id); return (status,res);}
  } /* end try_abort */
  void set(Identifier id, Outcome outcome) { overwrite(id,outcome); }
  Status retrieve(Identifier id) { return lookup(id).status; }
  ApplicationServer getAS(Identifier id) { return lookup(id).AS;}
}
```

Figure 5: `ITPLogger` Class.

Task 2: This task is activated upon receipt of a Decide message. The database server invokes the `xa_decide` primitive to take a final decision for the transaction. As it will be shown by Lemma 4 in Section 4.2, if `xa_decide` returns $unknown\_tid$, i.e. the database server is asked to decide for an unknown transaction ([7]), this implies that the database server must have already taken the same decision it is currently asked to take. Therefore, the database server simply sends back an Outcome message with an outcome equal to the requested decision, and accordingly updates the corresponding ITP. On the other hand, if `xa_decide` returns either $commit$ or $abort$, the database server sends back an Outcome message carrying the outcome returned by `xa_decide` and accordingly updates the corresponding ITP.

Task 3: This task is activated upon receipt of a Resolve message. The resolve phase is used in the attempt to abort a transaction possibly left pending due to some failure (e.g. crash of the application server originally taking care of it). In this case, the database server invokes the `try_abort` method of the `ITPLogger` class in the attempt to insert the ITP with the $abort$ value for the transaction state. If the latter operation succeeds, `xa_decide` is invoked during the execution of `try_abort`, with the $abort$ value as input parameter, and a Status message is sent back indicating the $abort$ value for the transaction state. Conversely, if the ITP insertion fails, the transaction state maintained by the ITP is retrieved through the `retrieve` method of `ITPLogger` and is sent back to the application server via a Status message.

[7] As pointed out in Section 2, by the XA specifications the database does not keep track of identifiers of already committed/aborted transactions.

Task 4: This is a background task used to avoid maintaining any pre-committed transaction blocked by a crash of the application server taking care of it. Actually, fail-over of a crashed application server might be performed by other application server replicas, if the client contacts them through Terminate messages. However, also the client might crash, e.g. after the issue of its request, thus originating a situation in which application server replicas are not notified that fail-over of the application server originally taking care of the transaction needs to be performed. This background task executed by the database server copes with this exact type of situation. Within this task, the database server checks whether there are transactions having an ITP with *prepared* state value, and for which the application server originally taking care of them is suspected to have crashed (recall the identity of such application server is retrieved through the ITP). For all of those transactions, the `resolve` method of the `DTManager` class is invoked to determine their final outcome.

Finally, the database server executes the following actions on recovery after a crash. Every pre-committed transaction either having an ITP with the *abort* value or having no ITP logged, is aborted through the `xa_decide` primitive. Actually, the presence of pre-committed transactions with the *abort* value within the corresponding ITP may occur due to non-atomicity in the execution of **Task 3**. Specifically, it could happen that, while executing the `try_abort` method within this task, after the successful insertion of the ITP with an *abort* indication for the transaction state, the database server is unable to execute the `xa_decide` primitive, e.g. due to a crash failure. If this problem were not tackled, we might have pre-committed transactions, which possibly hold locks on data, with an ITP indicating an *abort* state (i.e. the transactions need to be aborted), whose abort procedure will not eventually be handled by the database server. Similarly, the presence of pre-committed transactions with no corresponding ITP logged may occur due to non-atomicity in the execution of **Task 1**, i.e. the database server might crash after preparing the transaction through `xa_prepare` but before recording the ITP through the `prepare` method of the `ITPLogger` class. These transactions can be aborted by the database server since no *yes* vote has been sent out for them (recall the *yes* vote is sent out by the database server only after successful insertion of the ITP with *prepared* value).

3.4 Observations

Observation 1: By the protocol structure, a Decide message with the *commit* indication is ever sent to a database server, only if (1) positive Vote messages are collected by an application server from all the database servers or (2) Status messages with *prepared*/*commit* are collected by an application/database server from all the database servers. Note that this implies all the database servers have performed successful insertions of the ITP with the *prepared* value for the transaction state.

Observation 2: By the protocol structure, a Decide message with the $abort$ indication is ever sent to a database server, only if (1) at least one negative Vote message is collected by an application server from some database server or (2) at least one Status message with $abort$ is collected by an application/database server from some 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.

4 Protocol Correctness

In this section we provide a proof of the protocol correctness wrt the e-Transaction properties, which were formally specified in [11, 12, 13]. Since the protocol proposed in this work represents a solution to the very same problem, we inherit its specification, which is recalled below for the sake of the readers' convenience. As hinted, there are three categories of properties that define the e-Transaction problem: Termination, Agreement, and Validity. These properties are specified as follows:

Termination:

T.1 If the client issues a request, then, unless it crashes, the client eventually delivers a result.

T.2 If any database server votes for a result, then the database server eventually commits or aborts the result.

Agreement:

A.1 No result is delivered by the client unless the result is committed by all database servers.

A.2 No database server commits two different results.

A.3 No two database servers decide differently on the same result.

Validity:

V.1 If the client delivers a result, then the result must have been computed by an application server with, as a parameter, a request issued by the client.

V.2 No database server commits a result unless all database servers have voted yes for that result.

The intuitive meaning of the previous properties has been already pointed out in the Introduction. As the only additional note, the above properties express guarantees on data integrity (e.g. distributed transaction atomicity - see **A.3**) and data availability (e.g. the ability of a database server not to maintain pre-committed data blocked forever - see **T.2**) independently of what happens to the client. This well fits the "pure" crash model for the client (i.e. the client is not required to recover after a failure), which allows the e-Transaction framework to deal with very thin clients not providing access to stable storage ([8]).

[8] Access to stable storage might be precluded not only because of hardware constraints, but also due to security and

We proceed by first introducing the assumptions required to prove the correctness of our protocol. Next, according to the style in [11, 13], we prove the seven e-Transaction properties individually, by also relying on lemmas we introduce in order to simplify the structure of the proof.

4.1 Correctness Assumptions

We assume that at least one application server in the set $\{AS_1, \ldots, AS_n\}$ is correct, i.e. it does not crash. This assumption is required to ensure protocol termination since, in compliance with the e-Transaction framework, application servers adhere to a pure crash model. On the other hand, in practical settings, our protocol can still guarantee the e-Transaction properties even in the case of the simultaneous crash of all the application servers, as long as at least one of them eventually recovers and remains up long enough to complete the whole end-to-end interaction (application server recovery would not require any particular handling since application servers are assumed to be stateless processes).

Like in [11, 13], we assume that all database servers are good, which means: (1) they always recover after crashes, and eventually stop crashing (i.e. eventually they become correct), and (2) if the application servers keep re-trying transactions, these are eventually committed. Note that assuming the databases recover and eventually stop crashing means in practice assuming that application data are eventually available long enough to allow the end-user to successfully complete its interaction with the system. On the other hand, admitting the possibility for a database not to recover and remain up would lead to the extreme, not very realistic case in which the whole application remains indefinitely unavailable.

We assume that failure detection of clients and database servers against application servers is supported by a $\Diamond S$ (eventually strong) failure detector. Actually, this is a very weak failure detector class, whose properties can be expressed, according to its specification in [6] (and to the case of failure detection against application servers), as follows:

- *Strong Completeness.* Eventually every application server that crashes is permanently suspected by every correct process.
- *Eventual Weak Accuracy.* There is a time after which some correct application server is never suspected by any correct process.

Before proceeding we note that weaker assumptions than those introduced in this section can only lead to violations of Termination properties (i.e. liveness), but have no impact on Agreement

privacy issues.

and Validity properties (i.e. safety). As an example, the $\diamond S$ failure detector is required in order to ensure that at least one application server will be eventually given enough time to complete transaction processing, thanks to the avoidance of premature activation of the fail-over phase through Terminate messages. This would not be guaranteed by a failure detector which does not ensure the Eventual Weak Accuracy property.

4.2 Correctness Proof

Lemma 1: *If a correct application server receives either a* Request *or a* Terminate *message from a client, it eventually sends the corresponding* Outcome *message to the client.*

Proof (Sketch). Suppose a correct application server receives either a Request or a Terminate message from a client. In this case, given that all the primitives available at the application server are non-blocking, the correct application server keeps on re-transmitting, on a timeout basis, Prepare/Decide messages (case of Request from the client), or Resolve messages (case of Terminate from the client) to the database servers until Vote/Outcome or Status replies are received from all of them. Let t be the time after which all the database servers stop crashing and remain up. After t, by channel reliability, all the database servers eventually receive the above messages, triggering the activation of the corresponding database server tasks. Since these tasks execute only non-blocking primitives, all the database servers eventually send Vote/Outcome or Status messages to the correct application server. Given that channels are reliable, these messages are eventually received by this server. (A Status message might trigger a new round of interaction with the database servers through Decide messages which also eventually lead to a reply from the databases through Outcome messages, given that we are at time after t.) Hence the correct application server is able to eventually send back to the client the Outcome message.

Lemma 2: *If the client issues a request, then unless it crashes, it eventually receives a corresponding* Outcome *message.*

Proof (Sketch). Consider a client that sends a Request message and that does not crash. There are two cases: (1) The message is sent to a correct application server. By reliability of communication channels, this message is eventually received by this server that, by Lemma 1, eventually sends the Outcome message back. Again by reliability of communication channels, this message is eventually received by the client, since it does not crash. Hence the claim follows. (2) The message is sent to a non-correct application server. Suppose by contradiction that the client does not get the Outcome message. In this case, by the completeness property of the failure detector, the client eventually suspects this server and sends a Terminate message to another application server. At this point we have two cases: (2.1) The Terminate message is destined to a correct application server. In this case (by reliability of communication channels and Lemma 1) the client gets the Outcome message, hence the assumption is contradicted and the claim follows. (2.2) The Terminate message is destined to a non-correct application server. In this case, since we have assumed that the client does not get the Outcome message, by the completeness property of the failure detector, the client eventually suspects this server and sends a Terminate message to another application server. We note however that case 2.2 cannot occur indefinitely, since a Terminate message is eventually sent to a correct application server (recall we have assumed that there is at least one correct application server). Hence we eventually fall in case 2.1 and the claim follows.

Lemma 3: *If a database server suspects an application server, the database server eventually decides for every transaction already pre-committed by that application server.*
Proof (Sketch). If a database server, say DB_i, suspects an application server, say AS_i, this means that DB_i has pre-committed a transaction T initiated by AS_i, and has succeeded in inserting the ITP with $prepared$ value. In this case DB_i starts sending Resolve messages for the transaction to all the database servers on a timeout basis (this occurs even if DB_i crashes and then recovers). Let t be the time after which all the database servers stop crashing and remain up. After t, by channel reliability, all the database servers eventually receive the above Resolve messages, triggering the activation of the corresponding database server tasks. Since these tasks execute only non-blocking primitives, all the database servers eventually send back to DB_i Status messages, which are eventually received. Next, DB_i broadcasts to all the database servers (including itself) a Decide message carrying either the $abort$ or $commit$ decision. By the same previous arguments (i.e. reliability of channels and non-blocking primitives) these messages are eventually received by all the database servers and processed since we are at time after t, which leads DB_i to eventually take a decision for the transaction T. Hence the claim follows.

Lemma 4: *If a database is asked to decide $abort$ or $commit$ for an unknown transaction (this is the case of the $unknown\_tid$ in Task 2), the database has already taken the same decision it is currently asked to take.*
Proof (Sketch). By Observation 1 and Observation 2 in Section 3.4, no two application/database servers can take a different decision on the outcome of any 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, then it cannot have taken a different decision. Hence the claim follows.

Termination T.1: *If the client issues a request, then, unless it crashes, the client eventually delivers a result ([9]).*

Proof (Sketch). By Lemma 2, if the client issues a request and does not crash, it eventually receives an Outcome message. There are two cases: (1) If the Outcome message carries the $commit$ indication then the client delivers the result and the claim follows. (2) If the Outcome message carries the $abort$ indication, the client re-submits a new request and continues to do so until an Outcome message carrying the $commit$ indication is received. However, case 2 cannot occur indefinitely since there will be a time t after which all the database servers stop crashing and remain up, and the re-transmitted requests arrive to a correct application server (recall we have assumed there is at least one of such correct processes), which, by the accuracy property of failure detection, is never suspected by the client or by database servers. Hence neither does the client send Terminate messages nor do the databases send Resolve messages to attempt the abort of the transaction. Given that the database servers are good, the correct application server is able to eventually commit the transaction. Therefore, the Outcome message sent to the client eventually carries the $commit$ indication. Hence, the claim follows.

Termination T.2: *If any database server votes for a result, then the database server eventually commits or aborts the result.*

Proof (Sketch). Suppose a database votes for a result (i.e. pre-commits the transaction) but fails to insert the corresponding ITP with $prepared$ value. This is either because the database server crashes before the ITP insertion (in this case the transaction is aborted upon recovery) or because the `try_abort` method successfully inserts the ITP with $abort$ (in this case the transaction is aborted by this method or upon recovery). In either case the claim follows.

On the other hand, if the ITP with the $prepared$ state value is successfully inserted after the vote, we have two additional cases. (1) The application server that pre-committed the transaction crashes. Then, by the completeness property of failure detection, the database server eventually suspects the application server and by Lemma 3 decides on the transaction. Hence the claim follows. (2) The application server that pre-committed the transaction is correct. Then, by Lemma 2, this application server eventually sends an Outcome message to the client. This implies that the application server has ensured that all the database servers have decided for that transaction. Hence the claim follows.

[9]In the rest of this section, we use "vote/decide for a result" as synonymous with "vote/decide for a transaction". Similarly, "commit/abort a result" is used as synonymous with "commit/abort a transaction". This is done in order to maintain the same terminology used in [11, 12, 13] for the presentation of e-Transaction properties while ensuring, within the discussions, compatibility with the presentation of our protocol. For this same reason, delivery of the result at the client side expresses that the `issue` method returns a result associated with a committed transaction.

Agreement A.1: *No result is delivered by the client unless the result is committed by all database servers.*

Proof (Sketch). The client delivers the result only when an Outcome message with the *commit* indication is received from an application server. On the other hand, the application server sends the Outcome message to the client only after an Outcome message indicating *commit* is received by all database servers, which means that the result has been committed by all database servers.

Agreement A.2: *No database server commits two different results.*

Proof (Sketch). For a database server to commit two different results, we need the client to send at least two requests to the application servers. The client re-submits a new request only after it has received the Outcome message from an application server carrying the *abort* indication for the result associated with the last issued request. On the other hand, the application server returns to the client the Outcome message with the *abort* indication only after each database server either has already recorded an ITP with the *abort* state or has voted *no* for that result. As a consequence each time a new request is issued by the client, no previous request can eventually be committed since after the *abort* state is logged within the ITP, the database server rejects voting *yes* for that result. The same happens in case the database server already voted *no* since the `xa_prepare` primitive does not recognize the identifier associated with the result. As a consequence, no two different results can eventually be committed by a database server.

Agreement A.3: *No two database servers decide differently on the same result.*

Proof (Sketch). A database can decide *commit* only if it receives a Decide message with the *commit* indication, whereas it can decide *abort* if either (1) it receives a Decide message with the *abort* indication, or (2) it receives a Resolve message that causes a successful insertion of the ITP with the *abort* value for that result. By Observation 1 and Observation 2 in Section 3.4, no two Decide messages with different indications (*commit* or *abort*) can ever be sent, since the conditions enabling the send of Decide messages with different indications are mutually exclusive. Hence no two database servers can ever decide differently due to case (1). On the other hand, after the insertion of the ITP with the *prepared* value in **Task 1**, 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 of **Task 3**. Therefore, when a database server receives a Decide message to commit a transaction (by Observation 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. Hence no two database servers can ever decide differently due to case (2).

Validity V.1: *If the client delivers a result, then the result must have been computed by an application server with, as a parameter, a request issued by the client.*
Proof (Sketch). The client delivers the result after it receives an Outcome message with the $commit$ indication from an application server for a given identifier. Such a message is sent to the client if, and only if, the application server has received from each database server an Outcome message with the $commit$ indication for that identifier. This happens only if the result has already been committed. Given that a transaction associated with an identifier is committed only after an application server has computed and prepared it on all the databases, and given that the application server computes and prepares the transaction only after it has received the Request message with that identifier from the client, then it is not possible that the client delivers the result unless it has issued a request that has been computed by the application server.

Validity V.2: *No database server commits a result unless all database servers have voted yes for that result.*
Proof (Sketch). A database server commits a result after it receives the Decide message for that result with a $commit$ indication. On the other hand, by Observation 1 in Section 3.4, such a message can be sent only after every database server has logged an ITP for that result storing a $prepared$ state value. By the database server pseudo-code, an ITP with $prepared$ state value is logged only after the database server voted yes for that result. Thus, if a database server commits a result, then all database servers must have voted yes for that result.

5 Garbage Collection and Other Practical Issues

The following mechanism could be coupled with the protocol to deal with garbage collection of unneeded recovery information (i.e. unneeded ITPs) from the databases.

If the client experiences a nice run (i.e. a run with no suspect of failure), then an acknowledgement message can be sent to the application server right after the Outcome message has been received at the client side. Upon receipt of this message, the application server simply issues a request for discarding the corresponding ITPs to the back-end databases.

On the other hand, if the client receives the Outcome message for a given request identifier (possibly with the $abort$ outcome) only after having suspected the application server and after having sent Terminate messages for that request, the ITP (possibly inserted at each back-end database with the $abort$ value during the resolve phase handled by the application server) needs to be maintained. This is done to avoid that the transaction associated with that request identifier, which is asynchronously processed by the originally contacted application server, gets eventually committed, hence violating, e.g., the **A.2** e-Transaction property due to a re-transmission of a different request instance by the client after the receipt of the $abort$ outcome.

We note however that, in practical life, nice runs represent the most common cases. Hence, discarding the ITP in nice runs, while maintaining such a recovery information only in case of unlikely failure (or suspect of failure) situations, means in practice very limited growth of storage usage for recovery information over time.

Garbage collection without acknowledgments from the client could be further addressed if it were possible to determine a known period of time after which the client does not re-transmit requests and the application/database servers do not attempt to perform further processing/resolve actions for a given request. As already discussed in [11], these mechanisms are feasible if the underlying system matches assumptions proper of a timed model, e.g. [7]. Anyway, in practical settings one could still rely on approaches based on the association of an adequately selected Time-To-Leave (TTL) with each ITP, after which deletion of the ITP itself from the database can be performed.

As an additional note, we have presented the protocol under the assumption that all the back-end databases are accessed by the distributed transaction. However, for real life applications on, e.g., large scale systems, it could be possible that a request from a client needs transactional access only to a subset of those databases. Dealing with such a case would require a mechanism for persistently associating the client request with the identities of the subset of back-end databases really involved in the distributed transaction. In case the association can be deterministically defined by the application server as a function of the client request content, persistence of the association could be supported by including within the ITP the identities of the set of involved database servers, so that they could be able to execute the resolve phase associated with pending pre-committed transactions at the back-end tier (see **Task 4** in Section 3.3). The request content should also be piggybacked on Terminate messages from the client so that each application server can determine the set of involved database servers if the resolve phase is handled by the middle-tier.

On the other hand, if the association between the request content and the accessed back-end databases cannot be deterministically defined, one could rely on the following mechanism. The client request identifier can be used as the input of, e.g., a hash function determining the identity of a given database, say DB_x, on which the insertion of the ITP is forced during the prepare/resolve phase, even though that database would not be required to be accessed by the application logic. At the same time, the identities of the databases really involved in the transaction, plus the identity of DB_x, can be logged within the ITP. In this way, similarly to the above case, a database server keeping a pre-committed transaction is able to activate the resolve phase since the list of the involved databases is available within the ITP kept for that transaction. Also, an application server receiving the Terminate message from the client should send a Resolve message to DB_x (identified via the hash function) for either (i) retrieving the list of the involved databases, if the ITP with the $prepared$ state value has been inserted at DB_x, in order to execute the resolve phase on all of them, or (ii) allowing eventual abort of that transaction by performing the insertion of the ITP with the $abort$ value on DB_x ([10]).

Overall, in the case of deterministically defined association between the client request content and the set of involved back-end databases, we would only require slightly larger storage space for the ITP, but no additional log operations. On the other hand, in the case of non-deterministically defined association between the client request content and the set of involved back-end databases, we might require an additional round of messages in between the application server and DB_x during the resolve phase, in order to retrieve the list of the other database servers involved in the transaction. However, this additional communication cost does not need to be paid in nice runs. At the same time, a properly defined hash function allowing even distribution of additional forced ITP insertions (i.e. additional load) across the whole back-end tier would also prevent bottlenecks.

Finally, our protocol has been presented for the case of clients not recovering after a crash. This has been done in compliance with the e-Transaction specification, which, as already hinted, assumes a pure crash model for the clients. However, if stable storage capabilities were allowed at the client side, our protocol could be easily extended to deal with clients recovering after a crash. This could be done by logging Request/Outcome messages, and by having the client send out, upon recovery, Terminate messages for any Request message not having the corresponding Outcome in the log.

6 Related Work

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 [14]. However, this approach does not deal with the problem of loss of the outcome due, for example, to middle-tier server crash. The work in [18] tackles the latter issue by encapsulating within the same transaction both processing and the storage of the outcome at the client. In this solution, differently from our protocol, the client is required to be part of the transactional system since it is viewed as a recoverable resource participating in the 2PC protocol. This also means that, differently from our protocol, the solution in [18] necessarily requires stable storage capability at the client side.

Solutions based on the use of persistent queues have also been proposed in literature [3], which are commonly deployed in industrial mission critical applications and supported by standard middle-

[10] In case the transaction has been prepared at some database other than DB_x, this database will eventually abort the transaction during a resolve phase since this phase will find out the $abort$ indication within the ITP maintained by DB_x.

ware technology (e.g. JMS in the J2EE architecture, Microsoft MQ and IBM MQ series). With this approach, the application server receiving the client request needs to insert it 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. This solution is not suited for the case of high degree of replication and distribution of the application access point on a geographical scale. Specifically, since for such a large system organization queues are typically not replicated at all the application servers (because of the excessive overhead for maintaining their consistency), any interaction between the application server and the queuing system implies an interaction between remote systems. Hence, the additional interactions with the queuing system (for queueing requests and responses before and after the access to the application data), penalize the user perceived latency as compared to our protocol. Our solution is therefore more suited for those types of large systems, as we will show through the experimental study in Section 7.

The works in [1, 10] leverage database server logging to mask DBMS failures to client applications (e.g. by virtualizing ODBC sessions and materializing their state as persistent database tables). These solutions, despite being originally designed for client-server applications, could be also exploited in three-tier systems, e.g. to optimize server side failure handling by masking back-end database crash and recovery to the middle-tier application server executing the transactional logic. Compared to these approaches, our solution addresses reliability issues along the whole end-to-end interaction, allowing a client request to be re-submitted to a different middle-tier server replica, which is not required to recover any previously activated transactional session (it can simply start a new session). In this way, the transfer of the state of the original transaction coordinator to the fail-over replica is avoided, which makes our proposal particularly attractive for large scale replication of middle-tier servers.

Several optimizations were proposed in literature, aiming at reducing the messaging and logging overhead of standard 2PC, e.g. Presumed Commit/Abort [21], Early Prepare [29], Coordinator Log [28]. Our proposal differs from these solutions in that we exploit distributed logging activities performed by 2PC participants not only to achieve transaction atomicity, but also to ensure exactly-once execution semantic (i.e. idempotence and termination) of end-to-end interactions in a three-tier system. This is achieved by including additional information within the log performed by the 2PC participants, so to allow transaction outcome testability and retrieval of the result produced by the execution of the non-deterministic transactional logic.

The works in [2, 26], which extend and generalize the client-server tailored solutions in [19], 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 stateful middle-tier applications. Interaction contracts are implemented by logging sources of non-determinism, e.g. exchanged messages, to allow state reconstruction via a replay phase in the case of failures. Differently from these proposals, our solution is oriented to stateless middle-tier servers, not requiring to be involved in bilateral contracts suited for the interaction among stateful parties.

Another approach to address reliability in three-tier systems is the use of group communication [8, 20, 22]. 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 back-end databases, which is instead the case tackled by our protocol.

The solutions proposed by Frolund and Guerraoui in [11, 13] are based on primary-backup replication or asynchronous replication of the application servers. With these approaches, an application server receiving the client request needs to notify the replicas about the request, before performing any further operation (in the solution in [11] notification to the replicas takes place through a so called write-once register, providing the abstraction of a consensus object). The main problem with these solutions is that, differently from our proposal, they require explicit coordination among the replicas of the application server, which imposes an additional overhead and reduces system scalability. 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 cost of coordination can be kept low thanks to low delivery latency of messages among the replicas. This is not the case in large Web infrastructures like, e.g., ADNs, for which our proposal is more attractive, as we will show through the experimental study in Section 7. Similar considerations can be made for what concerns the proposal in [35], 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 application server replicas and the database.

There are two protocols [12, 24] that ensure the e-Transaction properties without the need for any coordination scheme over the middle-tier. However, the present work fundamentally differs from these proposals since they are restricted to the simpler case of a single back-end database server. Instead we address the more complex and general case of transactions that are striped across multiple, autonomous, distributed database servers, for which we need to face the additional problem of enforcing the agreement among multiple, heterogeneous transactional resources. Hence, compared to [12, 24], this proposal presents an innovative logging and manipulation scheme of recovery information at the back-end tier, providing support for atomicity of distributed transactions when 2PC is employed with stateless coordinators in a three-tier system. As a consequence, differently from the solutions in [12, 24], our proposal can cope with the case of, e.g., multiple parties involved in the same business process supported by the transactional application, like in challenging e-Commerce scenarios.

This paper extends our own prior work in [25], which presents a protocol dealing with effective support for the avoidance of duplicate transactions in three-tier systems when a timeout based re-transmission logic is employed. Compared to that work, this paper presents a protocol addressing the whole set of reliability properties as defined by the e-Transaction framework (i.e. Termination, Agreement and Validity properties), whose correctness under common assumptions has also been shown, and discusses other relevant practical issues.

7 Performance Measures and Comparison

In this section, we aim at quantitatively comparing the performance of our protocol against existing solutions. We carry out the comparison by presenting 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 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 databases due to contention).

We are interested in studying the case of normal execution (nice runs), i.e. when no process crashes or is suspected to have crashed. This is because, as already hinted in Section 5, those 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. Anyway, it is worth remarking that our protocol supports in practice the transfer of the transaction coordinator role over the middle-tier without explicit coordination among different application servers (transfer is triggered by the client re-transmission logic). Hence, as soon as there is at least one available (i.e. up and working) application server instance, system availability when our protocol is employed only depends on the availability of back-end database servers. This can be typically guaranteed via a set of solutions (see, e.g., [27, 33, 34]).

For simplicity of presentation, but with no loss of generality and without penalizing any of the compared protocols, we will focus on response time as seen by the application server, thus omitting the round-trip time between client and application server. We compare our protocol with the following alternatives:

1. A baseline protocol that coordinates distributed transaction processing via 2PC without logs on the coordinator (see Figure 6.a). This protocol tolerates crashes, with recovery of the back-end databases only.

2. The persistent queue (PQ) approach [3], whose behavior is schematized in Figure 6.b. This approach performs the enqueue of the client request as the first action. Next, it requires START

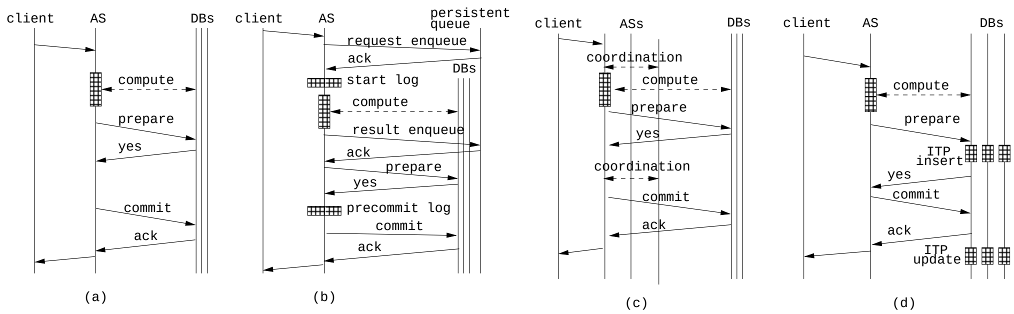

Figure 6: Schematization of the Behavior of the Compared Protocols.

and PRECOMMIT logs at the application server, i.e. classical 2PC, to guarantee atomicity of the distributed transaction. As already mentioned, this transaction also includes the enqueuing of the result of the data manipulation performed during the compute phase.

3. The primary-backup replication scheme (PBR) presented in [13] and the asynchronous replication scheme (AR) presented in [11]. The behavior of both these protocols can be schematized as shown in Figure 6.c, where the COORDINATION phase represents either the activity of propagating recovery information (i.e. the client request 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 for AR).

For completeness, we also show (see Figure 6.d) the schematized behavior of our protocol. Compared to the baseline we simply add (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. However, the latter action is performed after the database server sends out the Outcome message to the application server. Therefore the cost of this operation does not contribute to the response time perceived by the application server.

Supposing with no loss of generality that (i) the back-end databases have the same computational capacity and that (ii) the round-trip time between the application server and each back-end database $RTT_{as/db}$ is equal for all the databases, the response time of the baseline protocol $T_{baseline}$ in the case of normal execution, seen by the application server, can be expressed as ([11]):

$$T_{baseline} = (T_{SQL} + T_{xa\_prepare} + T_{xa\_commit}) + 3 \times RTT_{as/db} \tag{1}$$

where the term $RTT_{as/db}$ takes into account the fact that activation of functions in the XA interface and of the SQL associated with the transaction needs a message exchange from the application server

[11] In the 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.

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	2.22 or 20.30	0.46 or 0.81

to the database server and the corresponding acknowledgment ([12]). All the other protocols execute the same actions of the baseline plus additional actions (see Figure 6). Hence, the response time of each protocol can be expressed as follows:

$$T_{PQ} = T_{baseline} + T_{start} + T_{precommit} + 2 \times RTT_{as/qs} + T_{req-enqueue} + T_{res-enqueue} \quad (2)$$

$$T_{PBR} = T_{AR} = T_{baseline} + 2 \times T_{coordination} \quad (3)$$

$$T_{our\_protocol} = T_{baseline} + T_{ITP\_insert} \quad (4)$$

where $RTT_{as/qs}$ is the round-trip time between an application server and the queuing system used by PQ, and $T_{req-enqueue}$ (resp. $T_{res-enqueue}$) represents the time to enqueue the client request (resp. the transaction result) within that system.

Some parameters appearing in the latency models are left as independent variables in the performance study. They are (i) $RTT_{as/db}$, typically dependent on the relative locations of application servers and database servers, (ii) $RTT_{as/qs}$, typically dependent on the relative locations of application servers and the persistent queuing system, and (iii) $T_{coordination}$, which depends 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 speed of communication among application server replicas.

Other parameters have been measured through prototype implementations of PQ and of our protocol, relying on DB2 V8.1 and LINUX (kernel version 2.4.21). Our prototype of PQ performs the START and PRECOMMIT logs via operations on the file system, as in the approach commonly used by transaction monitors [4]. Also, as typically found in industrial environment (e.g. [30]), persistent message storing is supported through a database system, namely DB2 in our case. For what concerns our protocol, we have developed two different implementations. The first one manipulates the ITP via local transactions on DB2. In this case the ITP is maintained inside a user level database table. The second implementation is instead based on an optimized approach relying on the LINUX file system. In this case the ITP is maintained on files and ACID properties, as well as support for primary key constraint semantic, are ensured via standard file locking and synchronous operations on the disk. In order to use a representative value for T_{SQL} in the comparative study, we have also implemented the

[12] 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 for the management of the transactional logic.

TPC BENCHMARK^{TM} C (New-Order-Transaction) [32] (this benchmark portrays the activity of a wholesale supplier), 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. The two different values for T_{ITP_insert} and T_{ITP_update} refer to the cases of ITP managed through either the file system or local transactions on DB2 ([13]). Each reported value, expressed in msec, is the average over a number of samples that ensures a 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.)

We provide now quantitative comparisons among the protocols based on the proposed response time models and the obtained measurements. We analyze two different, representative scenarios. In the first one, application servers and database servers are assumed to be located on the same LAN; the same happens for the persistent queuing system used by PQ. In this case we have set $T_{coordination} = RTT_{as/qs} = RTT_{as/db}$ ([14]) and have varied the value of these parameters between 50 μsec and 10 msec so as to figure out different settings for what concerns the speed of the network and the communication layer among application servers, or among application servers and the queuing system in PQ. The case of 50 μsec is representative of settings in which the hosts on the LAN are connected, e.g., by a high speed switch, coupled with an ad-hoc message passing layer [23]. Instead, the case of 10 msec might be representative of a slower LAN with less performance effective message passing (e.g. based on RPC stacks such as RMI). In the second scenario we analyze, database servers are assumed to be spread on the Internet, thus $RTT_{as/db}$ has been set to the reasonable value of 250 msec [16]. Instead, $T_{coordination}$ and $RTT_{as/qs}$ have been varied between 1 and 250 msec so as to capture different organizations with respect to the spatial location of application server replicas and the queuing system in PQ. Lower values capture the cases in which those replicas (and the queuing system) are distributed, e.g., over the same LAN. The extreme value of 250 msec captures, instead, the opposite case in which they are distributed, e.g., over the Internet. Figure 7 shows the percentage of overhead of the

[13]As already pointed out, T_{ITP_update} does not contribute to the response time of our protocol as seen by the application server side since the ITP update is executed after the database server has sent the outcome to the application server. However we report the obtained measure for this parameter in order to provide the reader with indications on the update cost at the database side.

[14]In the absence of faults, a round of messages is the lower bound on message complexity required for transmission and acknowledgement of recovery information between the primary and the backups in the PBR solution. Also, as shown in [17], in the absence of faults a round of messages is the lower bound on message complexity for achieving consensus, i.e. for the management of the consensus object in AR. Therefore, the lower bound on the latency of coordination can be modeled as a round trip among two application servers, which is equal to $RTT_{as/db}$ if all the processes are hosted by machines on the same LAN.

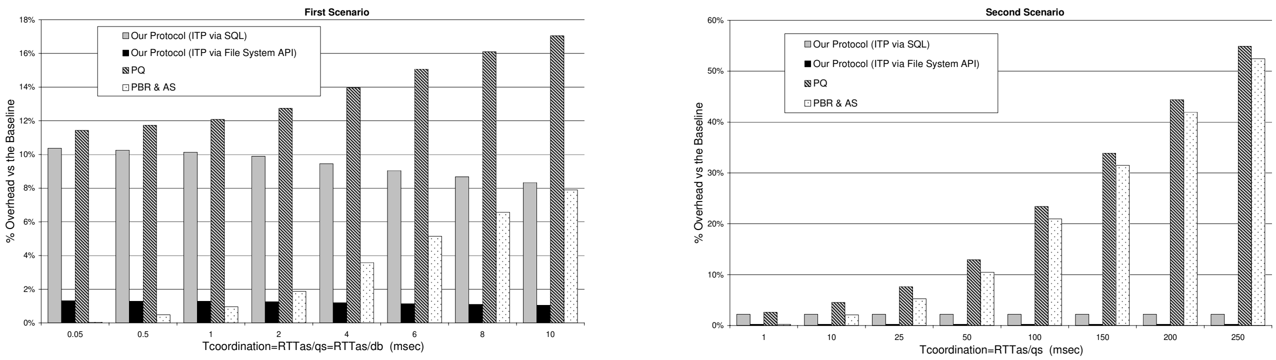

Figure 7: Overhead Percentage vs the Baseline Protocol.

protocols compared to the baseline. For the first scenario, we observe the following tendencies. The optimized implementation of our protocol based on file system API exhibits a negligible overhead percentage. For minimal values of $T_{coordination}$ and $RTT_{as/db}$, i.e. up to 0.5 msec, PBR and AR show the lowest overhead. On the other hand, as soon as the round trip time on the LAN hosting the processes gets larger, these protocols tend to perform similarly to the less efficient implementation of our protocol based on SQL. As an extreme, for round trip time of 10 msec, they show an increase in the overhead of up to 9 times as compared to the optimized implementation of our protocol. On the other hand, PQ performs worse than our protocol independently of the selected values for $RTT_{as/qs}$ and $RTT_{as/db}$, with gain from our protocol that increases while the values of $RTT_{as/qs}$ and $RTT_{as/db}$ increase. We note however that, independently of the relative values of their overhead percentages, all the protocols, except PQ, actually exhibit additional overhead compared to the baseline which is at most of 10%. With respect to the latter point, the optimized implementation of our protocol shows overhead percentage constantly under 1.5%. In the second scenario our protocol outperforms PBR, AR and PQ, for almost any considered value of $T_{coordination}$ and $RTT_{as/qs}$. Specifically, these protocols show overhead percentage comparable to our protocol only if $T_{coordination}$ (resp. $RTT_{as/qs}$) is maintained up to 10 msec (resp. 1 msec). On the other hand, as soon as these parameters assume a larger value, the overhead percentage of these protocols definitely grows. As an extreme, with the value of 250 msec, such an overhead percentage gets up to 27 times larger than the overhead percentage of the SQL based implementation of our protocol. Also, differently from the first scenario, this time PBR, AR and PQ exhibit a remarkable additional overhead compared to the baseline, i.e. up to 55%. Instead, our protocol keeps the additional overhead over the baseline constantly under 2% for the SQL based implementation, and under 0.3% for the optimized implementation relying on file system API. Overall, our protocol keeps the percentage of overhead over the baseline under 10% (this is drastically reduced to 2% in case of the optimized implementation) for whichever considered settings of both the investigated scenarios. This points out how, differently from other proposals, it exhibits good performance independently of the particular system organization, thus being attractive for usage in both the cases of local and wide area distribution of the application/database servers.

8 Summary

In this paper we have presented a distributed protocol ensuring reliability guarantees in the context of three-tier systems with stateless application servers, i.e. the so called e-Transaction guarantees. The protocol copes with the general case of multiple, autonomous databases in the system back-end. Hence it addresses the case of, e.g., multiple parties involved in a same business process supported by the transactional application. Our proposal completely avoids coordination among the replicas of the application server, which offer fail-over capabilities to each other. Hence it is well suited for contexts of large scale replication and/or geographic distribution of the application servers hosting the transactional logic for the manipulation of application data. We have also reported the results of a quantitative study (based on both standard benchmarks and analytical models), which demonstrate the performance benefits from our protocol compared to state of the art solutions.

References

[1] R. Barga, D. Lomet, S. Agrawal and T. Baby, " Persistent Client-Server Database Sessions", *Proc. of the 7th International Conference on Extending Database Technology (EDBT)*, pp.462-477, 2000.

[2] R. Barga, D. Lomet, G. Shegalov and G. Weikum, "Recovery Guarantees for Internet Applications", *ACM Transactions on Internet Technology*, vol.4, no.3, pp.289–328, 2004.

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

[4] P. Bernstein and E. Newcomer, "Principles of Transaction Processing", *Morgan Kaufmann*, 1997.

[5] 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. of the 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp.42–53, 1995.

[6] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the ACM*, vol.43, no.2, pp.225–267, 1996.

[7] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model", *IEEE Transactions on Parallel and Distributed Systems*, vol.10, no.6, pp.642–657, 1999.

[8] P. Felber, B. Gabrinato and R. Guerraoui, "The Design of a CORBA Group Communication Service", *Proc. of the 15th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pp.150–159, 1996.

[9] M.J. Fischer, N.A. Lynch and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process", *Journal of the ACM*, vol.32, no.2, pp.374–382, 1985.

[10] J.C. Freytag, F. Cristian and B. Käehler, "Masking System Crashes in Database Application Programs", *Proc. of the 13th International Conference on Very Large Data Bases (VLDB)*, pp.407–416, 1987.

[11] S. Frølund and R. Guerraoui, "Implementing e-Transactions with Asynchronous Replication", *IEEE Transactions on Parallel and Distributed Systems*, vol.12, no.2, pp.133–146, 2001.

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

[13] S. Frølund 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.

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

[15] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal and A. Schiper, "Consensus in Asynchronous Distributed Systems: A Concise Guided Tour", *Advances in Distributed Systems*, LNCS 1752, pp.33–47, 2000.

[16] "Internet Traffic Report", `http://www.internettrafficreport.com`

[17] I. Keidar and S. Rajsbaum, "On the Cost of Fault-Tolerant Consensus When There Are No Faults", *Proc. of the 1st Latin-American Symposium on Dependable Computing (LADC)*, pp.366–368, 2003.

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

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

[20] S. Maffeis, "Adding Group Communication and Fault-Tolerance to CORBA", *Proc. of the 1st USENIX Conference on Object Oriented Technologies (COOTS)*, 1995.

[21] C. Mohan, B. Lindsay and R. Obermarck, "Transaction Management in the R* Distributed Database Management System", *ACM Transactions on Database Systems*, vol.11, no.4, pp.378–396, 1986.

[22] P. Narashimhan. L.E. Moser and P.M. Melliar-Smith, "Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance", *Proc. of the 3rd USENIX Conference on Object Oriented Technologies (COOTS)*, pp.81–90, 1997.

[23] S. Pakin, M. Lauria and A. Chen, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet", *Proc. of the 9th ACM Int. Conference on Supercomputing (ICS)*, 1995.

[24] P. Romano, F. Quaglia and B. Ciciani, "A Lightweight and Scalable e-Transaction Protocol for Three-Tier Systems with Centralized Back-End Database", *IEEE Transactions on Knowledge and Data Engineering*, vol.17, no.11, pp.1578–1583, 2005.

[25] F. Quaglia and P. Romano, "Reliability in Three-tier Systems Without Application Server Coordination and Persistent Message Queues", *Proc. of the 20th ACM Symposium on Applied Computing (SAC)*, pp.718–723, 2005.

[26] G. Shegalov, G. Weikum, R. Barga and D. Lomet, "EOS: Exactly-Once E-Service Middleware", *Proc. of the 28th Conference on Very Large Databases (VLDB)*, pp.1043–1046, 2002.

[27] L. Shu, J.A. Stankovic and S. Son, "Achieving Bounded and Predictable Recovery using Real-Time Logging", *The Computer Journal*, vol.47, no.3, pp.373–394, 2004.

[28] J.W. Stamos and F. Cristian, "Coordinator Log Transaction Execution Protocol", *Distributed and Parallel Databases*, vol.1, no.4, pp.383–408, 1993.

[29] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES.", *IEEE Transactions on Software Engineering*, vol.5, no.3, pp.188–194, 1979.

[30] Sun Microsystems, "Sun Java System Message Queue v3.5 SP1 Administration Guide", `http://docs.sun.com/source/817-6024/imq_sys.html#wp23407`

[31] The Open Group, "Distributed TP: The XA Specification", `http://www.opengroup.org/pubs/catalog/c193.htm`

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

[33] M. Xiong, K. Ramamritham, J. Haritsa and J.A. Stankovic, "MIRROR: A State-Conscious Concurrency Control Protocol for Replicated Real-Time Databases", *Information Systems*, vol.27, no.4, pp. 277–297, 2002.

[34] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera and J. Gray, "The Design and Architecture of the Microsoft Cluster Service - A Practical Approach to High Availability and Scalability", *Proc. of the 28th Int. Symp. on Fault-Tolerant Computing Systems (FTCS)*, pp.422–431, 1998.

[35] H. Wu, B. Kemme and V. Maverick, "Eager Replication for Stateful J2EE Servers", *Proc. of the CoopIS, DOA, and ODBASE, OTM Confederated International Conferences*, pp.1376-1394, 2004.

Authors' Biographies

Francesco Quaglia received the Laurea degree (MS level) in Electronic Engineering in 1995 and the PhD degree in Computer Engineering in 1999 from the University of Rome "La Sapienza". From summer 1999 to summer 2000 he held an appointment as a Researcher at the Italian National Research Council (CNR). Since January 2005 he works as an Associate Professor at the School of Engineering of the University of Rome "La Sapienza", where he has previously worked as an Assistant Professor since September 2000 to December 2004. His research interests include distributed systems, parallel computing, parallel/distributed simulation, fault-tolerant programming and performance evaluation of software/hardware systems. He regularly serves as a referee for several international conferences and journals. He serves on the program committees of major conferences (like PADS, DS-RT, ANSS and NCA), and is a member of the editorial board of the *International Journal of Simulation and Process Modelling*. He has also served as Program Co-Chair of the 16th edition of PADS.

Paolo Romano received the Laurea degree (MS level) in Computer Engineering from the University of Rome "Tor Vergata". He is currently a PhD student at the University of Rome "La Sapienza". His research interests are in dependable distributed systems and his current working area is fault tolerant and highly performing wide-scale transactional systems. He serves as a referee for several international conferences in the distributed systems and networking areas.