Providing e-Transaction Guarantees in Asynchronous Systems with Inaccurate Failure Detection

Paolo Romano and Francesco Quaglia
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"

Abstract

In this paper we address reliability issues in Web-based transactional systems. We are interested in the category of systems characterized by stateless application servers. For these systems, a framework called e-Transaction has been recently proposed, which specifies a set of desirable end-to-end reliability guarantees. Within this framework we propose an innovative distributed protocol providing those reliability guarantees in the general case of multiple, autonomous back-end databases (typical of scenarios with multiple parties involved within a same business process). Compared to existing proposals coping with the e-Transaction framework, our protocol adopts a weaker approach to failure detection, i.e. it does not rely on any assumption on the accuracy of failure detection. Hence it reveals suited for a wider class of distributed systems, including those systems where the level of asynchrony makes stronger approaches to failure detection not feasible in practice. To achieve such a target, our protocol exploits an innovative scheme for distributed transaction management (based on ad-hoc demarcation and concurrency control mechanisms), which we introduce in this paper. We also provide hints on the protocol integration with conventional systems (e.g. database systems).

1 Introduction

Web-based transactional applications (e.g. e-Commerce applications) are typically supported by multi-tier systems, where middle-tier Web/application servers have the responsibility to interact with back-end databases on behalf of the client (e.g. an applet running in a browser). In this type of systems, reliability issues need to be addressed in different ways depending on whether middle-tier servers supporting the application are statefull or stateless entities. In fact, as recently discussed in [2], it is not clear how to adapt solutions tailored for one case to the other one.

Our focus is on systems with stateless middle-tier servers, for which the e-Transaction (exactly-once Transaction) framework has been recently proposed [10] as a set of desirable reliability guarantees belonging to the following three categories: *Termination*, *Agreement* and *Validity*. Termination guarantees ensure the liveness of the client-initiated interaction from a twofold prospective: not only it is guaranteed that a client does not remain indefinitely waiting for a response, but also that, in case of distributed transactions spanning multiple back-end sites, no database server maintains pre-committed data locked for an arbitrarily long time interval. Agreement embodies the safety properties of the system, ensuring both atomicity of the (distributed) transaction, and at-most once semantic for the processing of client requests. 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 present an innovative distributed protocol providing the e-Transaction guarantees in the general context of multiple, autonomous database servers in the back-end tier (i.e. the typical context of multiple parties involved within the same business process). Compared to existing solutions addressing the e-Transaction framework, our proposal has the distinguishing features of being suited for asynchronous systems and of not relying on any assumption on the accuracy of failure detection [5] ([1]). This allows our proposal to be suited for a wider class of distributed systems. These encompass general Web infrastructures layered on public networks over the Internet, possibly belonging to (and controlled by) providers offering different guarantees, or even no guarantee at all, on, e.g., the message transmission delay.

We note that the lack of accuracy in the failure detection may lead to the pathological situation in which false failure suspicions are issued indefinitely while handling the end-to-end interaction. In such a scenario, an extermination based approach before re-issuing requests might yield to an indefinite sequence of aborts of on-going work carried out on behalf of a given client by falsely suspected servers. On the other hand, if no extermination is performed, re-issuing requests might lead to blocking situations (due to pre-commit locks) involving both newly activated and previously activated work carried out by falsely suspected servers. In both

[1] The accuracy property of a failure detector embodies its ability not to (indefinitely) falsely suspect correct processes to have crashed.

cases, liveness can get compromised.

To overcome these problems, our protocol exploits an innovative scheme for distributed transaction management, based on ad-hoc demarcation and concurrency control mechanisms, which we refer to as Multi Instance Pre-commit (MIP). With this scheme, we allow a falsely suspected server to proceed with transaction processing and pre-commit (i.e. no attempt to force the abort of its work is performed). Also, any server performing fail-over of a client request is granted access to the pre-image of any uncommitted data item updated by (falsely) suspected servers previously processing that same client request. In this way newly activated work in case of fail-over does not need to force the abort of previously activated one, and the two works do not block each other, which provides liveness guarantees on the end-to-end interaction. At the same time, the different (pre-committed) work instances are reconciled at commit time to maintain application safety (e.g. at-most-once semantic for request processing).

The remainder of this paper is structured as follows. In Section 2, we present the protocol. Related work is discussed in Section 3. System integration issues are addressed in Section 4.

2 The Protocol

2.1 System Model

The e-Transaction framework [8, 10] has been proposed for three-tier systems with stateless application servers. We focus on these type of systems, and on the general case of multiple, autonomous database servers in the back-end tier.

Communication among processes (clients, application servers and database servers) is abstracted via message passing. Clients do not directly interact with database servers, they submit their requests to a set of application servers $\{AS_1, \ldots, AS_n\}$, which provide access to the application business logic. Once the client has sent a request to an application server, the request can be processed without further input from the client. The processing of the client request at the application server consists in the execution of a non-idempotent distributed transaction against a set of back-end database servers $\{DB_1, \ldots, DB_m\}$ ([2]), and in the computation of a response message to be delivered to the client. The response message destined to the client carries the result of the execution of the transactional business logic, which we assume to be non-deterministic since it depends on the state of the databases and, possibly, of other sources of non-determinism (such as the state of some hardware device). Application servers have no affinity for clients and 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. To simplify the presentation, we do not model chained invocation of middle-tier application servers, since, as already pointed out in [8], from a reliability standpoint chained invocation does not represent additional challenges in case of stateless application servers.

We model transactional interactions between an application server and the back-end database servers by means of two phases:

Compute phase: during this phase the application server performs (transient) manipulations of application data maintained by back-end database servers, e.g. by issuing SQL statements, which remain uncommitted as long as the database servers are not explicitly asked to commit the distributed transaction.

ACP phase: during this phase the application server coordinates the back-end database servers within an Atomic Commit Protocol (ACP) with the purpose of enforcing transaction atomicity (i.e. to ensure that all the database servers commit the distributed transaction, or none of them does). As it will be clear in the sequel, the ACP employed by our e-Transaction protocol is a version of the industrial standard two-phase commit (2PC) (i.e. it exhibits the same message pattern), which has been properly enhanced in order to support the MIP scheme.

Finally, processes are assumed to fail according to the crash failure model. Also, failures are soft so that database servers are allowed to correctly recovery after a crash.

2.2 Database Server Behavior

In our e-Transaction protocol, back-end database servers support distributed transactions according to the MIP scheme, whose features are described below:

Transaction Demarcation. Database servers associate with each transaction an identifier, namely a XID, which is composed by (1) a request identifier, namely req_id, univocally associated with a given client request, and (2) a transaction instance identifier, namely $inst\_id$, composed of a tuple $< category, instance\_number >$, where: $category$ identifies a given process (client or database server), and $instance\_number$ is a numerical value greater than or equal to zero. We assume that category values are ordered according to a lexicographic relation such that $category < category'$ if $category$ identifies a client process and $category'$ identifies a database server. Also, in case both the values identify two database servers, we say that $category < category'$ in case $category$ identifies a database server which precedes the one identified by $category'$ when ordering the set $\{DB_1, \ldots, DB_m\}$ according to some predetermined scheme (i.e. the set is ordered on the ba-

[2] For simplicity of protocol presentation, we assume that every transaction is executed over the same set of database servers, and that this set includes all the back-end databases. Extensions for handling the case of dynamic determination of the subset of involved back-end databases are not treated for space constraints.

sis of database servers indexes). Exploiting the previous ordering relation on category values, we assume that $inst\_id$ values are ordered according to the following relation: $inst\_id = < category, instance\_number >$ is less than $inst\_id' = < category', instance\_number' >$ if (i) $category < category'$ or (ii) $category = category'$ and $instance\_number < instance\_number'$. In the following, transactions sharing the same request identifier (req_id) but having different transaction instance identifiers ($inst\_id$) will also be referred to as *sibling transactions*.

ACP Supports. We model with the primitives `prepare` and `decide`, the database server interface for supporting the ACP. The primitive `prepare` takes as input a XID (i.e. a request identifier and a transaction instance identifier) and returns a value in the domain $\{prepared, abort\}$ reflecting whether the database server is able to commit the transaction or not. The primitive `decide` takes as input a XID and a decision in the domain $\{commit, abort\}$, and commits or aborts that transaction (i.e. determines the final outcome for that transaction). This primitive commits a transaction only if it was already pre-committed and the input decision is $commit$. We assume that, if invoked with the $commit$ indication for a prepared (i.e. pre-committed) transaction $XID = < req\_id, inst\_id >$, `decide` also aborts any pre-committed transaction with the same req_id having a transaction instance identifier $inst\_id'$ different from $inst\_id$. With no loss of generality we assume both `prepare` and `decide` to be non-blocking.

Concurrency Control. In case a transaction T requires (read/write) access to some data item d previously accessed (written/read) by a not yet committed (e.g. pre-committed) transaction T', T is granted access to the pre-image of d with respect to the execution of T' if T and T' share the same req_id (i.e. they are sibling transactions). Hence any update performed by a not yet committed transaction T' is not visible to any sibling transaction T. On the other hand, no assumption is made on how concurrency control regulates data accesses of non-sibling transactions.

Multi Instance Pre-commit Tables. A Multi Instance Pre-commit Table (MIPT) is persistently maintained by a database server for each set of transactions having the same req_id (i.e. sibling transactions associated with the same client request). In the following, we will denote with MIPT$_x$ the table keeping track of transactions with $req\_id = x$. The y-th entry of MIPT$_x$, namely MIPT$_x[y]$, stores the following information related to the transaction with $req\_id = x$ and transaction instance identifier $inst\_id = y$: (1) $state$: a value, in the domain $\{null, prepared, abort\}$, reflecting the transaction current state at that database (we assume that $null$ is the default initialization value); (2) $result\_set$: the (non-deterministic) output produced by the execution of the transaction at that database (also in this case we assume that $null$ is the default initialization value). Each MIPT$_x$ also keeps a special field, namely MIPT$_x.req$ which records the client request content that gave rise to the transactions with $req\_id = x$.

We are now able to provide the pseudo-code for the behavior of the database server within our e-Transaction protocol. This pseudo-code is shown in Figure 1. Actually, we do not describe the database server behavior during the transaction execution phase since, as stated in Section 2.1, this phase simply encompasses, e.g., a set of conventional SQL statements. The database server executes three tasks triggered by the receipt of different types of messages, and an additional background task.

Task 1: Upon the arrival of the Prepare$[req, < req_id, inst_id >, result_set]$ message from an application server, the `vote` method is invoked, which atomically performs the following operations. If MIPT$_{req_id}$ does not exist (i.e. the database server is attempting to prepare a transaction associated with a given req_id for the first time), the database server creates it and stores the request content within it. In case the entry of MIPT$_{req_id}$ with index $inst\_id$ has a $null$ state value (this always holds in case MIPT$_{req_id}$ did not exist and has been just created), the database attempts to prepare the transaction with $XID = < req\_id, inst\_id >$ by invoking `prepare`. In case the transaction is successfully prepared, the entry MIPT$_{req_id}[inst_id]$ is updated to store the $prepared$ state value and the $result\_set$ specified by the Prepare message. Otherwise, MIPT$_{req_id}[inst_id]$ is updated with the $abort$ value. Finally, when the `vote` method returns, MIPT$_{req_id}$ is sent back to the application server via a Vote message.

Task 2: Upon the arrival of the message Decide$[< req_id, inst_id >, decision]$ from an application server, the `decide` primitive is invoked to determine the requested outcome ($commit$ or $abort$) for the transaction. If the requested outcome is $commit$, `decide` (according to its specification) also enforces the abort of any other pre-committed transaction having the same req_id. Finally, a DecisionACK message is sent back to the application server.

Task 3: Upon the arrival of the message Resolve$[< req_id, inst_id >]$ from an application server, the method `resolve` is invoked, which atomically performs the following operations. For all the values of x less than $inst\_id$, it checks whether MIPT$_{req_id}[x]$ has a $null$ value. In the positive case, that value is set to $abort$. Finally, when the `resolve` method returns, MIPT$_{req_id}$ is sent back to the application server via a Vote message.

```
Class DatabaseServer{
 List ASlist = {AS_1, ..., AS_n}; ApplicationServer AS; TypeMIPT MIPT; Request req; State state; Outcome outcome;
 on stable storage Counter counter = InitialValue;

  void main(){
    while(true){
      cobegin
        || wait receive Prepare[req, < req_id, inst_id >, result_set] from application server AS_i // Task 1
          MIPT=vote(req, < req_id, inst_id >, result_set);
          send Vote[req_id, MIPT] to application server AS_i;
        || wait receive Decide[< req_id, inst_id >, decision] from application server AS_i // Task 2
          decide(< req_id, inst_id >,decision);
          send DecideACK[req_id] to application server AS_i;
        || wait receive Resolve[< req_id, inst_id >] from application server AS_i // Task 3
          MIPT=resolve(< req_id, inst_id >);
          send Vote[req_id, MIPT] to application server AS_i;
        || background: // Task 4
          for every transaction < req_id, - > maintained pre-committed for more than a TIMEOUT period {
            req = MIPT_{req_id}.req; AS = ASlist.next(); inst_id =< GetMyCategory(), ++counter >;
            send Request[req, < req_id, inst_id >] to AS;
            reset TIMEOUT period for the transaction identifier j;
          } // end for every
    } // end while
  } // end main

  TypeMIPT vote(Request req, XID < req_id, inst_id >, ResultSet result_set){
    atomically do{
      if (MIPT_{req_id} does not exist) {create MIPT_{req_id}; MIPT_{req_id}.req = req;}
      if (MIPT_{req_id}[inst_id].state == null) {
            state = prepare(req_id, inst_id);
            if (state == prepared) MIPT_{req_id}[inst_id].(state, result_set) = (prepared, result_set);
            else MIPT_{req_id}[inst_id].(state, result_set) = (abort, null);
      }// end if
    } // end of atomic statement
    return MIPT_{req_id};
  } // end vote

  TypeMIPT resolve(XID < req_id, inst_id >){
  InstanceIdentifier x;
    atomically do{
      ∀x < inst_id do {if ( MIPT_{req_id}[x].state == null) MIPT_{req_id}[x].state = abort;}
    }//end of atomic statement
    return MIPT_{req_id};
  }//end resolve
} // end class
```

Figure 1. Database Server Behavior.

Task 4: This is a background task used to avoid maintaining any pre-committed transaction blocked indefinitely. Within this task, the database server periodically checks whether there are transactions that are maintained in the pre-commit state longer than a timeout period. For each of these transactions, the original request content req is retrieved from the corresponding MIPT. Then, that same request is re-sent by the database server to whichever application server via a Request message. This message is also tagged with the original request identifier (i.e. req_id in the pseudo-code) and with a transaction instance identifier $inst\_id$ obtained by using (i) the database server identity as the category (i.e. the value $GetMyCategory()$) and (ii) an incremented counter value. Note that the used counter is assumed to be maintained on stable storage, which allows the database server to ensure monotonic increase of the counter even in case of recovery after a crash.

Observation 1: By the database server pseudo-code, an update on whichever $\text{MIPT}_{req\_id}[inst\_id]$ entry can only occur once, i.e. when that entry has the $null$ initialization value.

Observation 2: By the database server pseudo-code, whichever transaction $XID =< req\_id, inst\_id >$ can ever be prepared only if the corresponding entry $\text{MIPT}_{req\_id}[inst\_id]$ still keeps the $null$ initialization value.

2.3 Client and Application Server Behaviors

Figure 2 shows the pseudo-code defining the client behavior. Within the method `issue`, the client selects an application server and sends the request to this server, together with the request identifier req_id (we abstract over the details for the determination of the request identifier via `SetId()`) and the transaction instance identifier $inst\_id$ formed by a category value identifying the client process and a counter value (i.e. the instance number) maintained by the client application. It then waits for the reply, namely for an Outcome message for the transaction ([3]). In case the Outcome message arrives, carrying the $commit$ indication, `issue` simply returns the result of the transaction.

[3] The notation $< req\_id, - >$ means that the instance identifier is a don't care value. Hence, the client actually waits for an Outcome message associated with the specified req_id and with whichever $inst\_id$ value.

```
Class Client{
  List ASlist = {AS_1, ..., AS_n}; ApplicationServer AS; Result result;
  Outcome outcome; RequestIdentifier req_id; InstanceIdentifier inst_id;
  Counter counter = InitialValue;

  Result issue(Request req){
   outcome = abort;
   req_id = SetId(req);
   while(outcome == abort){
    AS = ASlist.next();
    set TIMEOUT;
    inst_id =< GetMyCategory(), ++counter >;
    send Request[req, < req_id, inst_id >] to AS;
    wait (receive Outcome[< req_id, - >, outcome, result] from
          any application server in ASlist) or (TIMEOUT);
   } // end while
   return result;
  } // end issue
} // end class
```

Figure 2. Client Behavior.

On the other hand, if the outcome is $abort$, the client re-transmits the same request after having incremented by one the counter used to define the transaction instance identifier. Otherwise, in case the contacted application server does not respond within a timeout period, the client selects a different application server and re-transmits its request to this server, also in this case after having incremented by one the counter used to define the transaction instance identifier. Then it waits again for an Outcome message from an application server or for a timeout expiration.

The pseudo-code defining the behavior of an application server is shown in Figure 3. The application server waits for a Request message from either a client or a database server. In case the Request message comes from a client, it is associated with either the original request, or a request re-transmission performed by the client. In case the request comes from a database server, say DB_i, it means that there is at least one pre-committed transaction instance associated with that same request, which has remained in the pre-commit state at DB_i for more than a timeout period (see Task 4 of the database server pseudo-code in Figure 1). In both cases, the Request message carries the request identifier (req_id) univocally associated with that client request, and the transaction instance identifier ($inst\_id$) defined by the identity of the sending process (client or database server) and by a monotonically increasing counter value. This simple scheme is sufficient to ensure that each Request message is univocally associated with a globally unique XID.

After the receipt of the Request message, the application server performs the compute phase for the corresponding transaction and determines the result set on each database. Then, it activates the first phase of the ACP protocol, during which it re-transmits Prepare messages to all the database servers on a timeout basis, until a Vote message is received from all of them. In our protocol, a Vote message from DB_i carries the MIPT$^i_{req_id}$ maintained by DB_i for transactions associated with that req_id value. Therefore, at the end of the Vote collection phase, an application server is informed not only about the state of the transaction it is currently handling (i.e. the one whose XID was specified in the Prepare message), but also about the state of any sibling transaction at all the database servers.

Once collected MIPT$^i_{req_id}$ from each database server DB_i, the application server verifies whether it is currently possible to take a positive (i.e. $commit$) decision for one of those sibling transactions. Specifically, the application server checks whether there is a transaction instance identifier j associated with req_id for which the following **Commit Condition** (**CC**) holds:

Sub-Commit-Condition-1 (SCC1): The transaction identified by $XID =< req\_id, j >$ has been prepared at all the database servers (i.e. the condition $\forall i \in [1, m], MIPT^i_{req\_id}[j].state == prepared$ is verified); and

Sub-Commit-Condition-2 (SCC2): No sibling transaction having $XID =< req\_id, j' >$, with $j' < j$, may ever become prepared at all the database servers (i.e. the condition $\forall j' < j, \exists i \in [1, m] : MIPT^i_{req\_id}[j'].state == abort$ is verified - by **Observation 2**, if the database server stores the $abort$ value on a MIPT entry, the corresponding transaction instance cannot be ever prepared at that database).

If no transaction instance associated with req_id has been found prepared at all databases (i.e. **SCC1** does not hold for any instance identifier, hence $PreparedInstance$ is not defined), then the application server makes sure that the transaction instance it is currently managing (i.e. the one associated with the received $inst\_id$) gets aborted at all the back-end databases. This is done by setting $InstanceToDecide$ to the value $inst\_id$, $outcome$ to the value $abort$, and then sending Decide messages with the negative (i.e. $abort$) indication for the transaction $XID =< req\_id, InstanceToDecide >$. These messages are re-sent on a timeout basis until acknowledgments are received from all the database servers. If both **SCC1** and **SCC2** are verified (i.e. $PreparedInstance$ is defined and no other transaction associated with req_id, having instance identifier less than $PreparedInstance$, can eventually become prepared), the application server sets $InstanceToDecide$ to the value $PreparedInstance$, $outcome$ to the value $commit$, and then sends Decide messages with the positive (i.e. $commit$) indication for the transaction $XID =< req\_id, InstanceToDecide >$ to the databases. Also in this case, these messages are re-sent on a timeout basis until the acknowledgements have arrived from all the database servers.

The only case left is when the application server has found some transaction instance associated with req_id prepared at all the databases (i.e. **SCC1** holds for some

```
Class ApplicationServer{
 List DBlist = {DB_1, ..., DB_m}; InstanceIdentifier PreparedInstance, CommittedInstance, InstanceToDecide; Outcome outcome; Result result;

 void main(){
  while(true){
      wait receive Request[req, < req_id, inst_id >] from client or from DB_i;
      compute the distributed transaction identified by < req_id, inst_id > and determine result_set^i for each DB_i; // compute phase
      repeat { // ACP vote phase
          send Prepare[req, < req_id, inst_id >, result_set^i] to each DB_i ∈ DBlist;
          wait (receive Vote[req_id,MIPT^i_{req_id}] from each DB_i ∈ DBlist) or TIMEOUT; // collect votes, i.e. MIPTs
      } until (received Vote[req_id,MIPT^i_{req_id}] from each DB_i ∈ DBlist);
      repeat{
          PreparedInstance = min(S), where S = {j : ∀i ∈ [1, m] MIPT^i_{req_id}[j].state == prepared};
          if (PreparedInstance is not defined){ InstanceToDecide = inst_id; outcome = abort; break;}
          if (PreparedInstance is defined) and (∀j < PreparedInstance, ∃i ∈ [1, m] : MIPT^i_{req_id}[j].state == abort){
             InstanceToDecide = PreparedInstance; outcome = commit; break;
          }
          repeat { // ACP resolve phase
             send Resolve[< req_id, PreparedInstance >] to each DB_i ∈ DBlist;
             wait (receive Vote[req_id,MIPT^i_{req_id}] from each DB_i ∈ DBlist) or TIMEOUT; // collect votes, i.e. MIPTs
          } until (received Vote[req_id,MIPT^i_{req_id}] from each DB_i ∈ DBlist);
      }until (TRUE);
      repeat{ // ACP decision phase
          send Decide[< req_id, InstanceToDecide >, outcome] to each DB_i ∈ DBlist;
          wait (receive DecideACK[req_id] from each DB_i ∈ DBlist) or TIMEOUT; // collect decision acknowledgments
      } until (received DecideACK[req_id] from each DB_i ∈ DBlist);
      if (Request was received from client){
          if (outcome == commit) assemble the result from all received MIPT^i_{req_id}[InstanceToDecide].result_set;
          else result = null;
          send Outcome[< req_id, − >, outcome, result] to client;
      }
  } // end while true
 } // end main
} // end class
```

Figure 3. Application Server Behavior.

transaction instance j, hence $PreparedInstance$ is defined), but it is still in doubt whether a transaction associated with the same req_id, and having instance identifier $j' < PreparedInstance$, can eventually become prepared at all the databases (i.e. **SCC2** does not currently hold). In this case, the application server sends Resolve messages (with the indication that we want to resolve doubts on instance identifiers up to $PreparedInstance$) and then recollects again Vote messages from each DB_i with the updated MIPT$^i_{req_id}$. (The Resolve message triggers some update operations on the corresponding MIPT at the recipient database, which mark the $null$ entries with index less than $PreparedInstance$ with the $abort$ value in order to prevent the corresponding transaction instances to be eventually prepared. Hence, the Vote messages will carry MIPTs comprising the updates triggered by the Resolve message, plus any update triggered by different types of messages, e.g. Prepare messages, sent to the databases by whichever application server.) Such a resolve phase gets over when the application server detects an instance identifier for which **CC** becomes satisfied (this might happen for a $PreparedInstance$ value different from the one for which **SCC1** was originally verified). At this point the final part of the ACP is executed for that instance via Decide messages, just as explained above.

As already hinted, the Request message triggering the activities at the application server might come from either the client or a database server. In case it comes from the client, the application server sends back to the client the final outcome right after the conclusion of the ACP. This also requires that the application server assembles the result message for the client by using all the result sets associated with the committed transaction instance (i.e. $InstanceToDecide$) on all the databases (each result set is retrieved from the MIPT$^i_{req_id}[InstanceToDecide]$ entry collected from each DB_i). On the other hand, in case the Request message comes from a database server, no reply needs to be sent back since database servers send Request messages with the only purpose to determine a final outcome for some transaction remained in the pre-commit state longer than a time-out period (see Task 4), and are not interested in receiving the corresponding result.

2.4 Correctness Assumptions

Due to space constraints, we cannot report the correctness proof of our protocol with respect to e-Transaction properties (as specified in [8, 9, 10]), for which we remind the interested readers to [15]. Anyway, in order to better frame our proposal, we list and discuss in the following the assumptions adopted in [15] to construct the correctness proof:

(A) No bound on message delay, clock drift or process relative speed is assumed, i.e. a classical asynchronous distributed system is considered [6]. Note that, assuming no level of synchrony and supporting failure detection via timeouts (as our protocol does) means making no assumptions on the accuracy of failure detection among processes. This is a relevant feature since, as it is well-known [5], no failure detector supporting accuracy properties can be implemented in a purely asynchronous distributed system.

(B) At least one application server is assumed to be correct, i.e. it is assumed not to crash. This assumption has been used to keep the protocol proof simple since it allows avoiding to explicitly handle application server recovery. In practical settings, our protocol can still guarantee the e-Transaction properties even in case of 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.

(C) Like in [8, 10], all database servers are assumed to be good, which means: (1) they always recover after a crash, and eventually stop crashing (i.e. eventually they become correct), and (2) if the application servers keep re-trying transactions, these are eventually prepared. Concerning point (1), assuming that 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. Concerning point (2), the ability to eventually commit transactions in case we keep retrying them does not contrast with the structure of our protocol, even if it allows multiple sibling transactions, associated with the same client request (i.e. tagged with the same req_id), to be concurrently active at the back-end databases (since they are activated on a timeout basis). In fact, these transactions do not block each other even in case of access to the same data, thanks to the assumed concurrency control scheme at the database side (see Section 2.2). Hence, the progress of none of these transactions is indefinitely prevented due to mutual dependencies.

3 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 [11]. However, this approach does not deal with the problem of loss of the outcome/result due, for example, to middle-tier server crash. The work in [12] tackles the latter issue by encapsulating within the same transaction both processing and the storage of the outcome at the client. This solution requires the client to be part of the transactional system since it is viewed as a recoverable resource participating in the 2PC protocol. Differently from this approach, our protocol does not require any recovery guarantee at the client side (as in the spirit of the e-Transaction framework). Also, exclusion of the client from the transaction boundaries allows the ACP latency to be independent of client participation timeliness. This provides advantages especially in case of clients connected via slower or less reliable channels (e.g. wireless channels), or in case of malicious clients intentionally delaying their replies while executing the ACP.

Solutions based on the use of persistent queues have also been proposed in literature [4], which are commonly deployed in industrial mission critical applications and supported by standard Message-Oriented-Middleware (MOM) technology. 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. Compared to this approach, our protocol does not require any additional log operation to be executed before/after the processing phase of the distributed transaction (in fact our protocol records the request content and the result on stable storage, i.e. within the proper MIPT, during the same log operation associated with the prepare phase of the distributed transaction). Also, in the case of large scale, geographical replication of the application servers, persistent queues are typically not replicated at all the application servers (because of the excessive overhead for maintaining their consistency). Hence, the log of the request/result on the persistent queue implies an additional interaction between remote systems, which might penalize the end-user perceived responsiveness.

There is some prior work addressing failure masking and/or recovery in transactional systems, e.g. [1, 7, 17]. These results are related to our work in that a form of logging of recovery information takes place at the database side to support specific reliability guarantees. However, the main difference with these approaches is that they focus only on client-server environments and do not address three-tier systems supporting, e.g., Internet applications.

The work in [2] addresses reliability in general multi-tier applications by employing interaction contracts between any two components, which specify permanent guarantees about state transitions, hence well fitting requirements of statefull middle-tier applications. Interaction contracts are implemented by logging sources of non-determinism, e.g., exchanged messages, so to allow state reconstruction via a replay phase in case of failures. Differently from these proposals, our solution is oriented to stateless middle-tier servers, not requiring to be involved in bilateral contracts suited for the interaction among statefull parties.

The work in [13] presents a protocol dealing with supports for the avoidance of duplicate transactions in three-tier systems when timeout based re-transmission logics are employed. Compared to that work, this paper presents a protocol addressing the whole set of reliability properties as defined by the e-Transaction framework, hence exhibiting the ability to cope with issues not tackled by that solution (e.g. data availability thanks to Termination guarantees).

The e-Transaction protocols in [8, 10, 14] require, respectively, *eventually perfect*, *perfect* and *eventually strong* failure detection capabilities, which provide the processes with the ability not to falsely suspect a correct process indefinitely. Hence, they require the failure detection system to be supported by an infrastructure providing a bounded level of asynchrony (see [5]). Instead, our proposal works also in completely asynchronous systems, with no guarantees at all on the level of failure detection accuracy (e.g. when timeout based failure detection is used in these kind of systems). Also, differently from our proposal, the solutions in [8, 10] require explicit coordination among the application server replicas (i.e. an application server receiving the client request needs to notify the request to the replicas before performing any further operation), which imposes additional overhead and reduces system scalability. Similar considerations can be made for what concerns the proposal in [18], where a primary server notifies to the backup replicas all the changes in its state before sending out any reply to the client. This solution also uses an agreement protocol to guarantee the consistency between the state of all the application server replicas and the database.

The e-Transaction protocols in [9, 16] 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 properties when multiple, heterogeneous transactional resources are involved in the end-to-end interaction.

4 System Integration Issues

For space constraints, we cannot present a detailed discussion on the issues concerning the integration of our proposal with commercial off-the-shelf systems. Anyway, we find worthwhile to provide at least some hints on the practical aspects an implementor would face when adopting our solution. Concerning the implementation of client and application server logic, this does not pose any practical difficulty. The former could be straightforwardly supported by means of, e.g., a Java applet running in a Web browser, whereas the implementation of the application server logic (and in particular of its activities as coordinator of the distributed transaction) could be delegated to middle-tier middleware frameworks (e.g. J2EE or .NET). Finally, for what concerns the integration of the MIP scheme with COTS database systems, in order to support, e.g., the ad-hoc transaction demarcation and concurrency control mechanisms (see Section 2.2), it would be necessary to alter the inner logic of the underlying DBMS. The hurdles an implementor should face while developing a MIP implementation are strictly related to the design choices of the specific DBMS. Anyway, MIP implementations could be relatively easily developed in case of integration with DBMS relying on data-item versioning for concurrency control purposes. Specifically, multiversion databases have the ability to maintain multiple versions of a same data-item, so that concurrency control selects which version must be supplied for a given read operation by a certain transaction [3]. Although this approach is orthogonal to our proposal (since multiversion concurrency control aims at increasing the concurrency level among independent transactions by letting them access different versions of the same data items), it could be anyway used as the basis for the concurrency control scheme required to support the MIP semantic (which aims at increasing the concurrency level among sibling transactions associated with the same client request).

References

[1] R. Barga, D. Lomet, S. Agrawal, and T. Baby. Persistent client-server database sessions. In *Proc. of the 7th International Conference on Extending Database Technology (EDBT)*, pages 462–477, 2000.

[2] R. Barga, D. Lomet, G. Shegalov, and G. Weikum. Recovery guarantees for Internet applications. *ACM Trans. Inter. Tech.*, 4(3):289–328, 2004.

[3] P. Bernstein and N. Goodman. Multiversion concurrency control theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[4] 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.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267, 1996.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), 1985.

[7] J. Freytag, F. Cristian, and B. Kähler. Masking system crashes in database application programs. In *Proc. of the 13th International Conference on Very Large Data Bases (VLDB)*, pages 407–416, 1987.

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

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

[10] 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.

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

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

[13] F. Quaglia and P. Romano. Reliability in three-tier systems without application server coordination and persistent message queues. In *Proc. of the 20th ACM Symposium on Applied Computing (SAC)*, pages 718–723, 2005.

[14] F. Quaglia and P. Romano. Ensuring e-transaction with asynchronous and uncoordinated application server replicas. In *Tech. Rep. TR-WEBMINDS-67*, 2006.

[15] P. Romano and F. Quaglia. Providing e-Transaction guarantees in asynchronous systems with inaccurate failure detection. In *Tech. Rep. TR-WEBMINDS-68*, 2006.

[16] 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*, 17(11):1578–1583, 2005.

[17] J. Stamos and F. Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1(4):383–408, 1993.

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