Providing e-Transaction Guarantees in Asynchronous Systems with No Assumptions on the Accuracy of Failure Detection

Paolo Romano and Francesco Quaglia

Abstract—In this paper, we address reliability issues in three-tier systems with 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. In this article, we propose an innovative distributed protocol providing e-Transaction guarantees in the general case of multiple, autonomous back-end databases (typical of scenarios with multiple parties involved within a same business process). Differently from existing proposals coping with the e-Transaction framework, our protocol does not rely on any assumption on the accuracy of failure detection. Hence, it reveals suited for a wider class of distributed systems. 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. Beyond providing the proof of protocol correctness, we also discuss hints on the protocol integration with conventional systems (e.g., database systems) and show the minimal overhead imposed by the protocol.

Index Terms—Three-tier systems, reliability, failure detection, distributed transactions.

1 Introduction

1.1 Context

Modern transactional applications (e.g., e-Commerce applications) are typically structured according to a multitier system organization, where middle-tier application servers have the responsibility to interact with back-end databases on behalf of the client. The partitioning of an application into multiple tiers provides the potentialities to achieve high modularity and flexibility. On the other hand, the multiplicity and diversity of the employed components and their interdependencies make it not trivial to achieve meaningful forms of reliability [11].

One aspect having a deep impact on the design of reliability solutions is whether middle-tier servers are statefull or stateless entities. In fact, as recently discussed in [2], it is not clear how to adapt (and make them efficient) solutions designed for one case to the other one. In the former case, middle-tier servers can retain some state information across different client invocations. Instead, in the latter one, the temporal scope of any middle-tier state information is limited to the processing of a single client request. On the other hand, the relevance of devising reliability mechanisms tailored for either scenario derives from that both application server models (statefull and stateless) are representative of mainstream design choices and system organizations. Generally speaking, the statefull model has the advantage of being less restrictive since it straightforwardly fits the requirements of applications where some constraints exist, which preclude the possibility to maintain all state information outside the middle tier (e.g., due to restrictive security or privacy issues). On the other hand, the stateless model has been shown to be more prone to performance-oriented (or even QoS oriented) system design since the lack of affinity between clients and application servers makes load balancing and redirection techniques much more lightweight and effective [29].

1.2 Focus

Our focus is on multitier systems with stateless middle-tier servers, in their most general configuration where the application logic is allowed to execute atomic transactions against a set of autonomous distributed back-end databases, e.g., as in the context of multiple parties involved within the same business process. For this kind of systems, Frølund and Guerraoui have recently proposed a reliability framework called e-Transaction (exactly once Transaction) [12], [14], which is specified via a set of seven properties belonging to the following three categories: *Termination*, *Agreement*, and *Validity*. Termination guarantees the liveness of client-initiated interactions from a twofold prospective: not only it is guaranteed that a client does not remain indefinitely waiting for a response, but also that no database server maintains precommitted data locked for an arbitrarily long time interval. Agreement embodies safety properties, ensuring both atomicity of the distributed transaction, and at most once semantic for the processing of client requests. Validity restricts the results' space to exclude meaningless results, e.g., invented ones.

1.3 Contribution

In this paper, we present an e-Transaction protocol that beyond revealing efficient (i.e., it requires less, or at most the same, message rounds and eager logs as literature protocols) has the distinguishing feature of not relying on

- *P. Romano is with INESC-ID, Room 615, Rua Alves Redol 9, 1000-029 Lisboa. E-mail: quaglia@dis.uniroma1.it.*
- *F. Quaglia is with the Dipartimento di Informatica e Sistemistica, Sapienza Universita' di Roma, via Ariosto 25, 00185 Rome, Italy. E-mail: quaglia@dis.uniroma1.it.*

Manuscript received 18 Jan. 2007; revised 7 July 2008; accepted 8 Dec. 2008; published online 7 Jan. 2009.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0005-0107.
Digital Object Identifier no. 10.1109/TDSC.2009.2.

any assumption on the accuracy of failure detection [7]. Compared to state-of-the-art e-Transaction protocols [12], [14], [24], all based on specific assumptions on the accuracy of failure detection, our proposal results suited for a wider class of distributed systems, encompassing general (large scale) infrastructures layered on public networks over the Internet, possibly owned by providers offering no guarantee on, e.g., the message transmission delay.

We note that complete 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 reissuing requests, like in [12], [24], 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, reissuing requests with no extermination might lead to blocking situations (due to precommit locks) involving newly activated and previously activated work carried out by falsely suspected servers. In both cases, liveness would 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-Precommit (MIP). With this scheme, we allow a falsely suspected server to proceed with transaction processing and precommit (i.e., no abort of its work takes place). Also, any server performing request fail-over is granted access to the preimage of any uncommitted data item updated by (falsely) suspected servers previously processing that same client request. Overall, fail-over work does not force the abort of on going work, and the two do not block each other, which provides liveness guarantees. The different (precommitted) work instances are finally reconciled at commit time to maintain application safety (e.g., at most once semantic for request processing).

We also note that from a pragmatical perspective, the absence of the extermination phase favors the fail-over timeliness independently of whether fail-over is triggered by a false or a correct failure suspicion. Hence, our protocol can provide performance benefits even in contexts where the failure detection service can actually offer some guarantees on its accuracy.

Beyond presenting the protocol and providing its correctness proof, we also discuss hints on its integration with conventional systems (e.g., database systems) and provide an evaluation of the protocol performance compared to some existing solutions.

The remainder of this paper is structured as follows: In Section 2, we present the protocol. Section 3 is devoted to the correctness proof. Some additional issues, e.g., garbage collection of unneeded recovery information, are discussed in Section 4. Performance and system integration issues are addressed in Section 5. Related work is presented and analyzed in Section 6.

2 The Protocol

2.1 System Model

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.

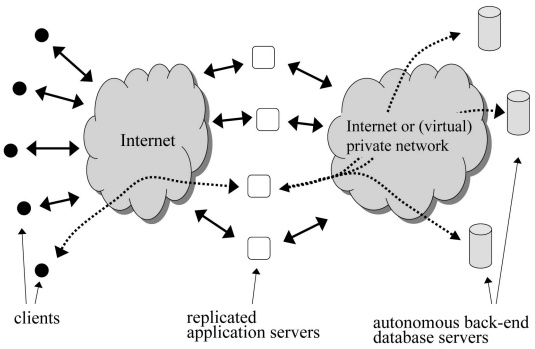

Fig. 1. Schematized System Architecture.

The processing of the client request at the application server consists of the execution of a nonidempotent-distributed transaction against a set of autonomous, back-end database servers $\{DB_1, \ldots, DB_m\}$, and the computation of a response message to be delivered to the client. A high-level schematization of the system architecture complying with our system model and of the interactions between different components is shown in Fig. 1. The response message destined to the client carries the result of the execution of the transactional business logic, which we assume to be nondeterministic since it depends on the state of the databases and, possibly, of other sources of nondeterminism (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 can only determine changes in the state of the databases.

Client and application server processes are assumed to fail according to the crash failure model. On the other hand, database servers are assumed to adhere to the crash/recovery failure model [8]. 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, with stateless application servers, the crash of the single application server contacted by the client is equivalent to the crash of any of the application servers in a chained invocation scheme [12], [14].

We model transactional interactions between the application server and 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. We abstract over this phase via a `compute` primitive, which is assumed to be nonblocking, i.e., it eventually returns unless the application server crashes during its execution.

ACP phase. During this phase, the application server executes an Atomic Commit Protocol (ACP) [16] 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).

2.2 Multi-Instance Precommit

Transaction demarcation. Back-end database servers associate with each transaction an identifier, namely an 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 $< instance\_number, category >$, where

- $instance\_number$ is an integer value greater than or equal to zero, namely a value within the domain $\mathbb{N}$;
- *category* is an identifier belonging to the domain $\{client, DB_1, \ldots, DB_m\}$, which distinguishes between a generic client and one of the back-end database servers.

We assume that category values are ordered according to the following relation: $category < category'$ if $category$ identifies a client process and $category'$ identifies a database server. Also, if both the values identify two database servers, we say that $category < category'$ if $category$ 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 (e.g., lexicographically). By exploiting the previous ordering relation on category values, we assume the following ordering relation on $inst\_id$ values: $inst\_id =< instance\_number, category >$ is less than $inst\_id' =< instance\_number', category' >$ if: 1) $instance\_number < instance\_number'$ or 2) $instance\_number = instance\_number'$ and $category < category'$. Also, $MinimumInstanceID =< 0, client >$ is used to denote the minimum element in the ordering. 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 in input an XID and returns a value in the domain $\{prepared, abort\}$ reflecting whether the database server is able to commit the transaction or not. In the former case, we say that the database server votes yes for the transaction, while in the latter case, the database server votes negatively and the transaction is aborted. The primitive `decide` takes in input an 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 prepared (i.e., precommitted) transaction if the input decision is $commit$. We assume that if invoked with the $commit$ indication for a transaction $XID =< req\_id, inst\_id >$, `decide` also aborts any precommitted transaction with the same req_id having a transaction instance identifier $inst\_id'$ different from $inst\_id$. Both `prepare` and `decide` are assumed to be nonblocking.

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 transaction T', T is granted access to the preimage of d with respect to the execution of T' if T and T' share the same req_id (i.e., they are sibling transactions). This occurs even if T' is in the precommit state. 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 nonsibling transactions.

Multi-instance precommit tables. All back-end database servers maintain an MIP Table (MIPT) 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 $req\_id = x$. (We will also use the notation MIPT_x^i for referring to the table maintained by a specific database server DB_i.) We assume that the entries of this table are not preallocated; hence, the database server allocates them whenever required. The yth entry of MIPT_x, namely $\text{MIPT}_x[y]$, if exists, 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 $\{prepared, abort\}$ reflecting the transaction state at that database and 2) *result*: the (nondeterministic) output produced by the execution of the compute phase for the whole distributed transaction (*null* indicates that no valid result has been computed).

Each MIPT_x also keeps a special field $\text{MIPT}_x.req$ recording the client request content that gave rise to the transactions with $req\_id = x$.

All MIPT manipulations occur with ACID guarantees. Hence, MIPT updates survive to database server crashes, and the updated MIPT is available right after database server recovery.

Finally, we assume the existence of the nonblocking method `GetMaxInstID`, which takes in input an MIPT and returns the maximum transaction instance identifier for which the corresponding MIPT entry has been allocated. We introduce this method exclusively for structuring the proof of the protocol correctness in Section 3.3, not for protocol construction.

2.3 Database Server Behavior

The pseudocode for the behavior of the back-end database server is shown in Fig. 2. We do not explicitly describe 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-message-triggered tasks and an additional background/upon recovery task:

Task 1. Upon the arrival of the Prepare $[req, < req\_id, inst\_id >, result]$ message from an application server, the `vote` method is invoked, which performs the following operations with ACID guarantees. 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. If the entry of MIPT_{req_id} with index $inst\_id$ does not exist (this always holds in case MIPT_{req_id} did not exist and has been just created), the database creates this entry and attempts to prepare the transaction with $XID =< req\_id, inst\_id >$ by invoking `prepare`. If the transaction is successfully prepared, the entry $\text{MIPT}_{req\_id}[inst\_id]$ is updated to

```
Class DatabaseServer{
  List ASlist = {AS_1, ..., AS_n}; ApplicationServer AS; TypeMIPT MIPT;
  Request req; State state; InstanceIdentifier inst_id; Outcome outcome;
  on stable storage Counter counter = 0;

  void main(){
   while(true){
    cobegin
1.   || wait receive Prepare[req, <req_id, inst_id>, result] from AS_i // Task 1
2.      MIPT=vote(req, <req_id, inst_id>, result);
3.      send Vote[req_id, MIPT] to AS_i;

4.   || wait receive Decide[<req_id, inst_id>, decision] from AS_i// Task 2
5.      decide(<req_id, inst_id>,decision);
6.      send DecideACK[<req_id, inst_id>] to AS_i;

7.   || wait receive Resolve[<req_id, inst_id>] from AS_i// Task 3
8.      MIPT=resolve(<req_id, inst_id>);
9.      send Vote[req_id, MIPT] to AS_i;

10.  || background and upon recovery: // Task 4
11.     for every transaction <req_id, -> pre-committed longer than TIMEOUT {
12.        req = MIPT_req_id.req;
13.        AS = ASlist.next();
14.        inst_id =<counter++, GetMyCategory()>;
15.        send Request[req, <req_id, inst_id>] to AS;
16.        reset TIMEOUT for that transaction;
17.     } // end for every
    } // end while
   } // end main

   TypeMIPT vote(Request req, XID < req_id, inst_id >, Result result){
18. execute with ACID guarantees{
19.  if (MIPT_req_id does not exist) {create MIPT_req_id; MIPT_req_id.req = req;}
20.  if (MIPT_req_id[inst_id] does not exist) {
21.   state = prepare (req_id, inst_id) ;
22.   create MIPT_req_id[inst_id];
23.   if (state = prepared)
24.      MIPT_req_id[inst_id].(state, result) = (prepared, result);
25.   else
26.      MIPT_req_id[inst_id].(state, result) = (abort, null);
27.  } //end if
28. } //end of ACID statements
29. return MIPT_req_id;
   } //end vote

   TypeMIPT resolve(XID < req_id, inst_id >){
    InstanceIdentifier x;
30. execute with ACID guarantees{
31.  ∀x < inst_id do {
32.   if (MIPT_req_id[x] does not exist) {
33.      create MIPT_req_id[x];
34.      MIPT_req_id[x].(state, result) = (abort, null);
35.   } //end if
36.  } //end of for all
37. } //end of ACID statements
38. return MIPT_req_id;
   } //end resolve

} // end class
```

Fig. 2. Database server behavior.

store the *prepared* state value and the *result* carried by the Prepare message. Otherwise, $\text{MIPT}_{req\_id}[inst\_id]$ is updated with the *abort* value. 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 Decide $[< req\_id, inst\_id>, decision]$ message from an application server, the `decide` primitive is invoked to determine the requested outcome (*commit* or *abort*) for the transaction. We recall that if the requested outcome is *commit*, `decide` (according to its specification) also enforces the abort of any other precommitted transaction having the same req_id. Finally, a DecisionACK message is sent back to the application server.

Task 3. Upon the arrival of the Resolve $[< req\_id, inst\_id>]$ message from an application server, the `resolve` method is invoked, which performs the following operations with ACID guarantees. For all the instance identifiers x less than $inst\_id$, it checks whether $\text{MIPT}_{req\_id}[x]$ does not exist. In this case, that entry is created and its value is set to $(abort, null)$. Finally, when the `resolve` method returns, MIPT_{req_id} is sent back to the application server via a Vote message.

Task 4. This is a background task, also executed upon recovery after a crash, which avoids maintaining any precommitted transaction blocked indefinitely. Within this task, the database server checks whether there are transactions that are maintained in the precommit state longer than a time-out period.[1] For each of these transactions, the original request content req is retrieved from the corresponding MIPT. Then, that same request is resent 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 pseudocode) and with a transaction instance identifier $inst\_id$ obtained by using: 1) an incremented counter value as the instance number and 2) the database server identity as the category (i.e., the return value of `GetMyCategory`). The used counter is assumed to be maintained on stable storage, thus ensuring its monotonic increase even in case of recovery after a crash.

Observations. The below observations immediately follow from the structure of the database server pseudocode.

Observation 2.1. Whichever $\text{MIPT}_{req\_id}[inst\_id]$ entry can be set at most once. In fact, by lines 20-27 and lines 32-35 of the database server pseudocode, no update can occur in case that entry already exists.

Observation 2.2. By lines 20-21 of the database server pseudocode, whichever transaction $XID = < req\_id, inst\_id>$ can ever be prepared (i.e., the primitive `prepare` can ever be invoked for this transaction) only if the corresponding entry $\text{MIPT}_{req\_id}[inst\_id]$ does not already exist at that database.

Observation 2.3. By lines 21-24 of the database server pseudocode, whichever entry $\text{MIPT}_{req\_id}[inst\_id]$ is set to maintain the *prepared* state value only if the `prepare` primitive is successfully executed for the transaction $XID = < req\_id, inst\_id>$.

Observation 2.4. By lines 31-33 of the database server pseudocode, after the completion of the `resolve` method with input $XID = < req\_id, inst\_id>$, each entry $\text{MIPT}_{req\_id}[j]$ with $j < inst\_id$ exists.

Observation 2.5. By lines 21-22 of the database server pseudocode, after the completion of the `prepare` method with input $XID = < req\_id, inst\_id>$, the entry MIPT_{req_id} $[inst\_id]$ exists.

2.4 Client and Application Server Behavior

Fig. 3 shows the pseudocode 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 the nonblocking primitive `SetId`) and the transaction instance identifier $inst\_id$ formed by a counter value (i.e., the instance number)

1. The notation $< req\_id, ->$ indicates that the instance identifier is a don't care value.

```
Class Client{
  List AStlist = {AS_1, ..., AS_n}; ApplicationServer AS; Result result;
  Outcome outcome; RequestIdentifier req_id;
  InstanceIdentifier inst_id; Counter counter = 0;

  Result issue(Request req){
1.   outcome = abort;
2.   req_id = SetId(req);
3.   while(outcome = abort){
4.    AS = AStlist.next();
5.    set TIMEOUT;
6.    inst_id =<counter++, GetMyCategory()>;
7.    send Request[req, <req_id, inst_id>] to AS;
8.    wait ( receive Outcome[<req_id, ->, outcome, result] from
            any AS_i ∈ AStlist or TIMEOUT );
9.   } // end while
10.  return result;
  } // end issue
} // end class
```

Fig. 3. Client behavior.

maintained by the client application, and a category value identifying the client process. Then, the client waits for an Outcome message.[2]

If the Outcome message arrives, carrying the *commit* indication, `issue` simply returns the result of the compute phase. On the other hand, if the outcome is *abort*, the client retransmits the same request after having incremented by one the instance number. Otherwise, if the contacted application server does not respond within a time-out period, the client selects a different application server and retransmits its request to this server, also in this case after having incremented the instance number. Then, it waits again for an Outcome message or time-out expiration.

The application server pseudocode is shown in Fig. 4. The application server waits for a Request message from either a client or a database server.[3] In any case, the Request message carries the request identifier (req_id) univocally associated with that client request, and the transaction instance identifier ($inst\_id$) defined by a monotonically increasing counter value and by the identity of the sending process (client or database server). 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 distributed transaction and determines the result to be delivered to the client. Then, it activates the first phase of the ACP protocol, during which it retransmits Prepare messages (tagged with the XID associated with the computed transaction) to all the back-end database servers on a time-out basis, until Vote messages are received from all of them. In our protocol, a Vote message from DB_i carries the $\text{MIPT}^i_{req\_id}$ 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 sent by the application server), but also about the state of sibling transactions, if any, at all the database servers.

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

 void main(){
  while(true){
1.  wait receive Request[req, <req_id, inst_id>] from client or from DB_i;
2.  result =compute(req, < req_id, inst_id >);

3.  repeat { // ACP vote phase
4.   send Prepare[req, <req_id, inst_id>, result] to each DB_i ∈ DBlist;
5.   wait (receive Vote[req_id,MIPT^i_req_id] from each DB_i∈DBlist) or TIMEOUT;
6.  } until (received Vote[req_id,MIPT^i_req_id] from each DB_i ∈ DBlist);

7.  repeat{
8.   S = { inst. id. j : ∀i ∈ [1,m] MIPT^i_req_id[j].state = prepared};
9.   if (S ≠ ∅){ // PC verified
10.   PreparedInstance = min(S);
11.   if (∀ inst. id. j < PreparedInstance, ∃i ∈ [1,m] :
                MIPT^i_req_id[j].state = abort){ // CC verified
12.     InstanceToDecide = PreparedInstance;
13.     outcome = commit;
14.     break; //goes to the ACP decision phase with positive outcome
15.   }
16.  }
17.  else{  // PC is not verified
18.   InstanceToDecide = inst_id;
19.   outcome = abort;
20.   break; //goes to the ACP decision phase with negative outcome
21.  }

22.  repeat { // ACP resolve phase (CC not yet verified), re-collect updated MIPTs
23.   send Resolve[<req_id, PreparedInstance>] to each DB_i ∈ DBlist;
24.   wait (receive Vote[req_id,MIPT^i_req_id] from each DB_i∈DBlist) or TIMEOUT;
25.  } until (received Vote[req_id,MIPT^i_req_id] from each DB_i ∈ DBlist);
26. }until (TRUE);

27. repeat{ // ACP decision phase
28.  send Decide[<req_id,InstanceToDecide>,outcome] to each DB_i∈DBlist;
29.  wait (receive DecideACK[req_id] from each DB_i ∈ DBlist) or TIMEOUT;
30. } until (received DecideACK[req_id] from each DB_i ∈ DBlist);

31. if (Request was received from client){
32.  if (outcome = commit)
33.   set result to whichever received MIPT^i_req_id[InstanceToDecide].result;
34.  else result = null;
35.  send Outcome[<req_id, ->, outcome, result] to client;
36. }
  } // end while true
 } // end main
} // end class
```

Fig. 4. Application server behavior.

Once collected $\text{MIPT}^i_{req\_id}$ 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. This requires several steps. First, the application server checks whether there is at least one sibling transaction that is precommitted at all the back-end databases. Formally, this means verifying if the following condition holds.

Prepare condition— $\mathcal{PC}$. Let S be the set of instance identifiers j, built using $MIPT^i_{req\_id}$ values, with $i \in [1,m]$, as follows:

$$S = \{j : \forall i \in [1,m], MIPT^i_{req\_id}[j].state = prepared\}, \quad (1)$$

we say that the Prepare Condition ($\mathcal{PC}$) is satisfied iff $S \neq \emptyset$. (The verification of $\mathcal{PC}$ is performed in lines 8-9 of the application server pseudocode.)

If no transaction instance associated with req_id has been found prepared at all databases (i.e., S is an empty set; hence, $\mathcal{PC}$ does not hold), the application server aborts the currently managed transaction instance (i.e., the one associated with the received $inst\_id$). This is done by setting $InstanceToDecide$ to the value $inst\_id$, $outcome$ to the value $abort$, and then sending Decide messages with $abort$ for the

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

3. If the request comes from a database server, say DB_i, it means that there is at least one precommitted transaction instance associated with that same request, which remained in the precommit state at DB_i longer than a time-out period (see **Task 4** of the database server pseudocode in Fig. 2).

transaction $XID =< req\_id, InstanceToDecide >$ to all the databases. These messages are resent on a time-out basis until acknowledgments are received from all the database servers. On the other hand, if $\mathcal{PC}$ is verified (i.e., $S \neq \emptyset$), the application server goes on checking whether the following condition holds:

Commit condition—$\mathcal{CC}$. Given a not empty set S as in expression (1), and let $PreparedInstance = min(S)$, we say the the Commit Condition ($\mathcal{CC}$) holds iff

$$\forall j < PreparedInstance, \exists i \in [1, m] : MIPT^{i}_{req\_id}[j].state = abort. \quad (2)$$

(The verification of $\mathcal{CC}$ is performed in lines 10-11 of the application server pseudocode, after having assigned to the variable $PreparedInstance$ the minimum value in the set S according to the ordering relation defined on instance identifiers in Section 2.3—recall that S is not empty upon the verification of $\mathcal{CC}$ since $\mathcal{PC}$ has been already preventively verified.)

By Observation 2.2, if the $abort$ value is recorded by whichever $\text{MIPT}^{i}_{req\_id}[j]$ entry, the corresponding transaction instance cannot be ever prepared at DB_i. Hence, if $\mathcal{CC}$ is verified, it means that no sibling transaction with $XID = < req\_id, j >$, such that $j < PreparedInstance$ may ever become prepared at all the back-end database servers. In such a case, the application server sets $InstanceToDecide$ to the value $PreparedInstance$, $outcome$ to the value $commit$, and then sends Decide messages with $commit$ for the transaction $XID =< req\_id, InstanceToDecide >$ to all the database servers. Also, these messages are resent on a time-out basis until acknowledgements arrive from all the database servers.

In Fig. 5a, we show an example where the application server AS_1 receives and processes the original request from the client (tagged with $inst\_id =< 0, client >$) and then finds for this request both $\mathcal{PC}$ and $\mathcal{CC}$ verified when receiving $MIPT^{1}_{req\_id}$ and $MIPT^{2}_{req\_id}$ during the ACP phase from the two back-end database servers. A close case is in Fig. 5b, where AS_2 receives and processes a request retransmitted by the client (tagged with $inst\_id = < 1, client >$ [4] and during the ACP finds both $\mathcal{PC}$ and $\mathcal{CC}$ verified for the original request (tagged with $inst\_id =< 0, client >$). In this case, AS_2 commits the transaction associated with the original request.

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., S is not empty; hence, $\mathcal{PC}$ holds), but it is still in doubt whether a transaction associated with the same req_id, and having instance identifier $j < PreparedInstance$, with $PreparedInstance = min(S)$, can eventually become prepared at all the databases (i.e., $\mathcal{CC}$ does not currently hold). In this case, the application server sends Resolve messages to all the database servers (with the indication that we want to resolve doubts on instance identifiers up to $Prepared\text{-}\ Instance$), and then recollects again Vote messages from each DB_i with the updated $\text{MIPT}^{i}_{req\_id}$. (By Task 3 of the database server pseudocode—see Fig. 2—the Resolve message triggers the creation of nonexisting MIPT entries with index less than $PreparedInstance$, which are set to the $abort$ value in order to prevent the corresponding transaction instances from being eventually prepared. Hence, the Vote messages will carry MIPTs comprising the updates triggered by the Resolve message, plus any update triggered by other messages, e.g., Prepare messages, sent to the database servers by whichever application server.) Such a resolve phase ends when the application server detects that $\mathcal{CC}$ becomes satisfied. At this point, the final part of the ACP is executed for some transaction instance via Decide messages, just as explained above. An example pattern involving Resolve messages is shown in Fig. 5c, where the resolve phase executed by AS_2 marks $MIPT^{2}_{req\_id}[< 0, client >].state$ to $abort$, thus allowing $\mathcal{CC}$ to become satisfied for the retransmitted request tagged with $inst\_id =< 1, client >$ (i.e., the one handled by AS_2).

As already hinted, the Request message triggering the activities at the application server might come from either the client or a database server. If it comes from the client, the application server sends back to the client the final outcome right after the conclusion of the ACP, together with the result of the distributed transaction for which the decision has been taken (the result is retrieved from some $\text{MIPT}^{i}_{req\_id}[InstanceToDecide]$ received by the application server from whichever DB_i). On the other hand, if 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 precommit state longer than a time-out period (see Task 4).

3 Protocol Correctness

3.1 e-Transaction Specification

For the reader's convenience, we report the specification [12], [14] of the three categories of e-Transaction properties:

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, the request issued by the client. **V.2:** No database server commits a result unless all database servers have voted positively for that result.

The intuitive meaning of these properties has been pointed out in Section 1. As an additional note, they 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 precommitted data blocked forever—see **T.2**) independently of what happens to the client. This makes recovery capabilities at the client side not mandatory.

3.2 Correctness Assumptions

Communication channels between processes are assumed to be reliable, with the meaning that a sent message eventually arrives at the destination process unless either

4. In the example, the retransmission occurs due to the crash of the originally contacted application server AS_1, which prevents the outcome message to be delivered at the client side.

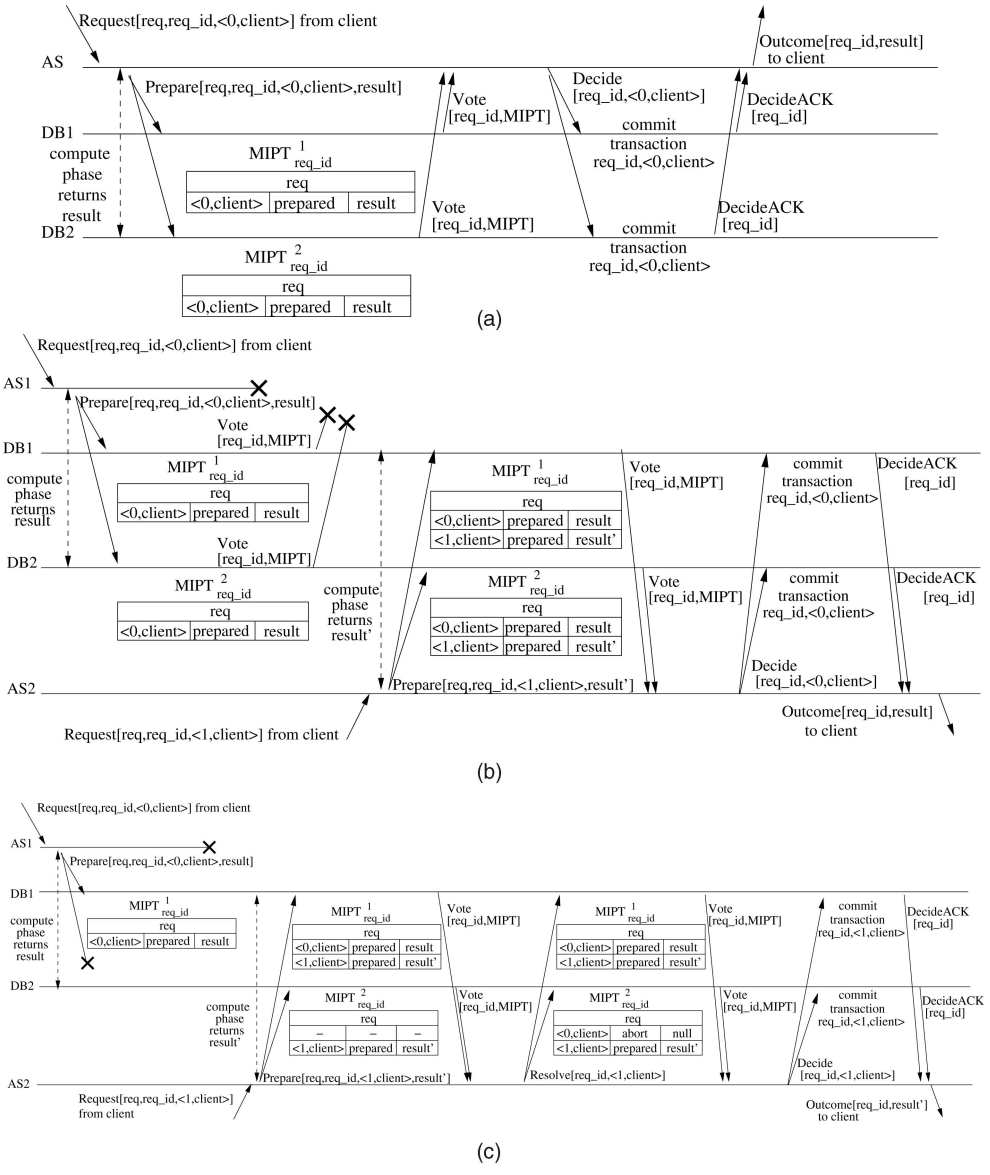

Fig. 5. Protocol example patterns.

the sender or the receiver crashes during the transmission. However, we do not assume the existence of any bound on message delay, clock drift, or process relative speed, i.e., we consider a classical asynchronous distributed system [8], [22], [9]. Note that assuming no level of synchrony and supporting failure detection via time-outs (as our protocol does) means making no assumptions on the accuracy of failure detection among processes.

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 simplifies the protocol proof since it allows us not to explicitly handle application server recovery. In practical settings, our protocol can guarantee the e-Transaction properties even in the 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. Anyway, the formal requirement for a single application server to be correct provides our protocol with optimal middle-tier failure resilience.

Like in [12], [14], 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 retrying transactions, these are eventually prepared. Concerning point 1, in practice, we are 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. Furthermore, in such a case, no protocol could ever guarantee that a client request is eventually successfully processed, thus leading to a violation of property T.1 in the e-Transaction specification. We also note that assuming all database servers eventually recover and stay up allows us to circumvent well-known results proving the impossibility to solve agreement problems (such as consensus [9] and nonblocking atomic commit [18]) in asynchronous systems with no failure

detection accuracy if (transactional) processes are not allowed to recover after a crash, or only a subset of these processes are allowed to recover and stay up. Concerning point 2, the ability to eventually prepare transactions if the application servers keep retrying them does not contrast with the structure of our protocol. In fact, sibling transactions (possibly activated due to request retransmissions) never block each other in the access to the same data items (see the MIP concurrency control scheme in Section 2.3). Hence, the progress of none of these transactions is indefinitely prevented due to mutual direct dependencies.

3.3 Correctness Proof

Lemma 3.1. *At finite time, the number of entries of whichever* $MIPT^{i}_{req\_id}$ *is finite.*

Proof (By Contradiction). Assume that at some finite time, some $\mathrm{MIPT}^{i}_{req\_id}$ has an infinite number of entries. These infinite entries can be created by DB_i only in the following three cases: 1) DB_i receives an infinite number of Prepare messages tagged with req_id and different values for $inst\_id$ from the application servers and creates the entries via the `vote` method executed after the receipt of these messages (see lines 20-26 of the database server pseudocode). Given that an application server can send a `Prepare` message tagged with req_id and with some $inst\_id$ only after having received the Request message with that $inst\_id$ value from either the client or some database server (see lines 1-4 of the application server pseudocode), and given that the client and the database servers send out Request messages tagged with req_id and different values for $inst\_id$ on a time-out basis, for the database server to receive an infinite number of Prepare messages we need to be at infinite time. Hence, the assumption is contradicted and the claim follows. 2) DB_i receives an infinite number of Resolve messages tagged with req_id and different values for $inst\_id$ from the application servers and creates the entries via the `resolve` method executed after the receipt of these messages (see lines 32-33 of the database server pseudocode). Given that an application server sends out Resolve messages tagged with req_id and $inst\_id$ to the database servers only after having verified that $\mathcal{PC}$ holds, with $inst\_id = min(S)$, and given that, by Observation 2.1, the value of $\mathrm{MIPT}^{i}_{req\_id}[inst\_id].state$, once set to *prepared* cannot be updated, the only possibility for the application servers to send an infinite number of Resolve messages tagged with req_id and different values of $inst\_id$ is that $min(S)$ is sometimes found by some application server to be equal to the instance identifier $< instance_number, -> $, with $instance\_number \rightarrow \infty$. Hence, a transaction with $XID =< req\_id, < instance\_number, -> >$, such that $instance\_number \rightarrow \infty$, must have been computed and prepared by some application server. However, given that the application server computes and prepares the transaction only after having received the corresponding Request message (see lines 1-6 of the application server pseudocode), and given that the client and the database servers send out Request messages tagged with req_id and different values for $inst\_id$, with incremented instance number, on a time-out basis, for the application server to receive the Request message with $XID =< req\_id, < instance\_number, -> >$, such as $instance\_number \rightarrow \infty$, we need to be at infinite time. Hence, the assumption is contradicted and the claim follows. 3) DB_i receives a Resolve message tagged with req_id and $inst\_id$ from some application server, causing the creation of an infinite number of entries within $\mathrm{MIPT}^{i}_{req\_id}$ via the `resolve` method executed after the receipt of this message (see lines 30-35 of the database server pseudocode). In this case, it needs to hold that $inst_id =< instance_number, -> $, with $instance\_number \rightarrow \infty$. Given that the application server can send out a Resolve message tagged with req_id and $inst_id =< instance_number, -> $, such that $instance\_number \rightarrow \infty$, only after having verified that $\mathcal{PC}$ holds, with $inst\_id = min(S)$, a transaction with $XID =< req\_id, < instance\_number, -> >$, such that $instance\_number \rightarrow \infty$, must have been computed and prepared by some application server. However, as for case 2), this cannot occur at finite time. Hence, the assumption is contradicted and the claim follows. □

Lemma 3.2. *At finite time, the test of $\mathcal{PC}$ in lines 8-9 of the application server pseudocode is nonblocking (Part A). The same holds for the test of $\mathcal{CC}$ in lines 10-11 of the application server pseudocode (Part B).*

Proof —Part A (By Contradiction). The statement in line 8, which expresses the construction of set S in a functional way starting from $\mathrm{MIPT}^{i}_{req\_id}$ values (with $i \in [1, m]$)), can be implemented by means of the following simple algorithm.

```
SetOfInstanceIdentifiers buildSetS(MIPT^1_req_id, ..., MIPT^m_req_id) {
    InstanceIdentifier j, GlobalMaxInstID, MinimumInstanceID =<0,client>;
    SetOfInstanceIdentifiers S;

1.  S = ∅;
    // Determine the maximum inst_id value for which a corresponding
    // MIPT^i_req_id[inst_id] entry exists for at least one value of i ∈ [1, m]
2.  GlobalMaxInstID = max_{i∈[1,m]} { MIPT^i_req_id.GetMaxInstID() };
3.  for each j ∈ [MinimumInstanceID, GlobalMaxInstID] {
4.      if (∀i ∈ [1, m], MIPT^i_req_id[j] exists ∧ MIPT^i_req_id[j].state = prepared)
5.          S = S ⋃ j;
6.  } //end for each
7.  return S;
}
```

Assume by contradiction that this algorithm is blocking, i.e., if executed at some finite time, it does not eventually get completed by a correct application server. Given that: 1) the statement in line 1 is a plain assignment; 2) the `GetMaxInstID` method on MIPTS is nonblocking; and 3) the statement in line 2 can be implemented by means of a finite number of comparisons, both the statements in lines 1 and 2 are nonblocking. Hence, for the algorithm to be blocking, it means that a correct application server never exits the **for each** cycle in lines 3-6. Given that by Lemma 3.1, each $\mathrm{MIPT}^{i}_{req\_id}$ is bounded at finite time, the statement in line 4 can be executed via a finite set of comparisons; hence, it is nonblocking. Also, the statement in line 5 is nonblocking, being implementable as an insertion into a list. Hence, for the **for each** cycle in lines 3-6 not to be eventually completed, it needs to hold that the cardinality of the set comprising the instance identifiers within the interval $[MinimumInstanceID,\ GlobalMaxInstID]$ is infinite. In this case, there is at least one $\mathrm{MIPT}^{i}_{req\_id}$ with a valid entry $\mathrm{MIPT}^{i}_{req\_id}[j]$, whose state value is set to *prepared*, such that $j =< instance_number, -> $, with $instance\_number \rightarrow \infty$. However, for this to occur, it must hold that some Request message, tagged with $XID =< req\_id, < instance\_number, -> >$, such that $instance\_number \rightarrow \infty$, must have been received by some application server, and that the corresponding

transaction must have been computed and prepared at some database. Given that the client and the database servers send out Request messages tagged with req_id and different values for $inst\_id$, with incremented instance number, on a time-out basis, for the application server to receive a Request message tagged with $XID =< req\_id, < instance\_number, -> >$, such that $instance\_number \rightarrow \infty$, we need to be at infinite time. Hence, the assumption is contradicted. Also, given that by the previous algorithm, S is a subset of the set comprising the instance identifiers within the interval $[MinimumInstanceID, GlobalMaxInstID]$, and this latter set has been shown to be finite, then S is finite as well. Therefore, the statement in line 9, which verifies whether S is not empty is nonblocking since it can be implemented by, e.g., an enumeration of that finite set. Hence, the claim follows.

Part B—(By Contradiction). By Part A, we have that at finite time, after the test of the $\mathcal{PC}$ property, the set S has a bounded number of elements (given that the cardinality of the set $[MinimumInstanceID, GlobalMaxInstID]$ is finite, and hence, the **for each** cycle in lines 3-6 of the above algorithm iterates a finite number of times). Therefore, we trivially have that the determination of the minimum value over a finite set and the assignment to the variable $PreparedInstance$ in line 10 of the application server pseudocode is nonblocking. Hence, to prove that the test of $\mathcal{CC}$ in lines 10-11 of the application server pseudocode is nonblocking, we only need to prove that the verification in line 11 is nonblocking. Assume by contradiction that the verification in line 11 is blocking. Given that, by Lemma 3.1, at finite time each $\mathrm{MIPT}^{i}_{req\_id}$ has a bounded number of entries, for this verification to be blocking we need to have $PreparedInstance =< instance\_number, - >$, with $instance\_number \rightarrow \infty$. Hence, a transaction with $XID =< req\_id, < instance\_number, -> >$, such that $instance\_number \rightarrow \infty$, must have been computed and prepared by some application server. However, as already shown in Part A, this is impossible at finite time. Hence, the assumption is contradicted and the claim follows. □

Lemma 3.3. *If an application server sends a* Decide $[< req\_id, inst\_id>, commit]$ *message, then no application server can ever send a* Decide $[< req\_id, inst\_id'>, commit]$ *message with* $inst\_id' \neq inst\_id$.

Proof (By Contradiction). Assume that an application server sends a Decide $[< req\_id, inst\_id>, commit]$ message and some application server sends a Decide $[< req\_id, inst\_id'>, commit]$ message with $inst\_id' \neq inst\_id$. Without loss of generality, let us consider the case $inst\_id' < inst\_id$. The application server sends out a Decide $[< req\_id, inst\_id>, commit]$ message only after having collected $\mathrm{MIPT}^{i}_{req\_id}$ from each DB_i, with $i \in [1, m]$ (see lines 3-6 of the application server pseudocode), and after having verified that both $\mathcal{PC}$ and $\mathcal{CC}$ hold, with $inst\_id = min(S)$ (see lines 8-11 of the application sever pseudocode). In such a case, by the definition of $\mathcal{CC}$, the following relation holds on the $\mathrm{MIPT}^{i}_{req\_id}$ values collected by the application server

$$\forall j < inst\_id, \exists i \in [1, m] : \mathrm{MIPT}^{i}_{req\_id}[j].state = abort. \quad (3)$$

On the other hand, some application server sends a Decide $[< req\_id, inst\_id'>, commit]$ message only after having collected $\mathrm{MIPT}^{i}_{req\_id}$ from each DB_i, with $i \in [1, m]$, and after having verified that both $\mathcal{PC}$ and $\mathcal{CC}$ hold, with $inst\_id' = min(S)$. However, by the construction of set S in expression (1), by expression (3), and given that $inst\_id' < inst\_id$, it follows that $inst\_id' \notin S$, unless the state value of $\mathrm{MIPT}^{i}_{req\_id}[inst\_id']$ has changed at some database server DB_i from $prepared$ to $abort$ (or vice versa). Since, as explained by Observation 2.1, this is impossible, the assumption is contradicted and the claim follows. □

Lemma 3.4. *If an application server sends a* Decide $[< req\_id, inst\_id>, commit]$ *message, then no application server ever sends a* Decide $[< req\_id, inst\_id>, abort]$ *message, and vice versa.*

Proof (By Contradiction). Assume that an application server sends a Decide $[< req\_id, inst\_id>, commit]$ message and some application server sends a Decide $[< req\_id, inst\_id>, abort]$. The application server sends a Decide $[< req\_id, inst\_id>, commit]$ message only after having collected $\mathrm{MIPT}^{i}_{req\_id}$ from each DB_i, with $i \in [1, m]$ (see lines 3-6 of the application server pseudocode), and after having verified that both $\mathcal{PC}$ and $\mathcal{CC}$ hold, with $inst\_id = min(S)$ (see lines 8-11 of the application sever pseudocode). In such a case, by the construction of set S in (1), the following relation holds on the $\mathrm{MIPT}^{i}_{req\_id}$ values collected by the application server

$$\forall i \in [1, m] : \mathrm{MIPT}^{i}_{req\_id}[inst\_id].state = prepared. \quad (4)$$

Some application server can send a Decide $[< req\_id, inst\_id>, abort]$ message (see lines 18-19) only if this server has received the Request message tagged exactly with $XID =< req\_id, inst\_id>$ (see line 1 and lines 18-19), has executed the `compute` primitive (see line 2), and has completed the vote phase (see the **repeat** loop in lines 3-6) for this XID value. This implies that every database server DB_i has received the Prepare message tagged with $XID =< req\_id, inst\_id>$, has executed the `prepare` method for this XID value, and has sent back the Vote message to the application server carrying the $\mathrm{MIPT}^{i}_{req\_id}$. Note that, by Observation 2.5, after the completion of the `prepare` method at DB_i with input $XID =< req\_id, inst\_id>$, $\mathrm{MIPT}^{i}_{req\_id}[inst\_id]$ exists. Also, the $abort$ decision is taken by the application server only in case once collected $\mathrm{MIPT}^{i}_{req\_id}$ from each DB_i, with $i \in [1, m]$, $\mathcal{PC}$ is found not to hold (see line 17), implying

$$\nexists\, j : \forall i \in [1, m], \mathrm{MIPT}^{i}_{req\_id}[j].state = prepared. \quad (5)$$

Hence, by the fact that $\mathrm{MIPT}^{i}_{req\_id}[inst\_id]$ exists $\forall i \in [1, m]$, and by (5), we have that the following relation must hold:

$$\exists i \in [1, m] : \mathrm{MIPT}^{i}_{req\_id}[inst\_id].state = abort. \quad (6)$$

In order for the expressions in (4) and (6) to be both verified, some database server DB_i should have updated $\mathrm{MIPT}^{i}_{req\_id}[inst\_id].state$ from $abort$ to $prepared$ (or vice versa). However, by Observation 2.1, this cannot occur. Hence, the assumption is contradicted and the claim follows. □

Lemma 3.5. *If a correct application server finds $\mathcal{PC}$ verified while processing a transaction $XID =< req\_id, inst\_id >$, this server eventually sends* Decide *messages with the commit indication for a transaction $XID' =< req\_id, ->$ to all the databases.*

Proof (By Contradiction). Assume by contradiction that a correct application server which finds $\mathcal{PC}$ verified while processing a transaction having $XID =< req\_id, inst\_id >$ does not eventually send Decide messages with the *commit* indication for a transaction $XID' =< req\_id, ->$ to all the database servers. Given that $\mathcal{PC}$ is verified, the set S defined as in (1) is not empty (see lines 8-9 of the application server pseudocode). After having found $\mathcal{PC}$ verified, the application server tests $\mathcal{CC}$ (see lines 10-11 of the application server pseudocode). This test gets eventually completed by Lemma 3.2. The following two cases are possible: 1) The application server also finds $\mathcal{CC}$ verified for some $PreparedInstance = min(S)$. In this case, it sets $InstanceToDecide$ to $PreparedInstance$ and $outcome$ to $commit$ (see lines 12-13 of the application server pseudocode), and sends out Decide $[< req\_id, InstanceToDecide >, outcome]$ messages (see line 28 of the application server pseudocode). Hence, the assumption is contradicted and the claim follows. 2) The application server finds $\mathcal{CC}$ not verified. In this case, it goes on executing the **repeat** loop in lines 22-25, by sending out, on a time-out basis, Resolve messages to all the database servers, with the indication to resolve instances up to $Prepared\,Instance = min(S)$, and waiting for Vote replies from all of them. There are two cases: 2.1) The application server eventually receives the Vote message from each database server DB_i, carrying the updated $\text{MIPT}^i_{req\_id}$ value. In this case, the application server goes on executing the **repeat** loop starting in line 7 one more time, by testing again $\mathcal{PC}$ (see lines 8-9). Such a test gets eventually completed by Lemma 3.2. Given that by Observation 2.1, each $\text{MIPT}^i_{req\_id}$ entry can only be updated once by DB_i, then S is again not empty; hence, $\mathcal{PC}$ is again found verified. At this point, the correct application server goes on testing $\mathcal{CC}$ (see lines 10-11). Such a test gets eventually completed by Lemma 3.2. Given that the database server DB_i replies to the Resolve message only after having executed the `resolve` method, and given that, by Observation 2.4, after the execution of this method, each entry $\text{MIPT}^i_{req\_id}[j]$, with $j < min(S)$, exists, then for each $j < min(S)$ exists $i \in [1, m]$ such that $\text{MIPT}^i_{req\_id}[j].state = abort$, thus $\mathcal{CC}$ is verified for $PreparedInstance = min(S)$. Hence, we fall in case 1 and the claim follows. 2.2) The application server does not eventually receive the Vote message from all the database servers, hence indefinitely remaining in the execution of the **repeat** loop in lines 22-25. This means that it indefinitely remains retransmitting Resolve messages to the database servers on a time-out basis. However, given that there is a time t after which the database servers are up and do not crash, the application server is correct and the communication channels are reliable, each database server DB_i eventually receives the Resolve message, which triggers the execution of the `resolve` method (see lines 7-8 of the database server pseudocode). Also, given that all the statements within the `resolve` method are nonblocking, each database server DB_i eventually sends back the Vote reply to the correct application server (see line 9 of the database server pseudocode). This message is eventually received by the correct application server from each DB_i since we are at time after t and the communication channels are reliable. Hence, the only case possible is 2.1 and the claim follows. □

Lemma 3.6. *If a correct application server keeps on receiving* Request *messages for a given req_id, with different transaction instance identifiers, it eventually sends* Decide *messages with the commit indication for a transaction $XID =< req\_id, ->$ to all the database servers.*

Proof (By Contradiction). Assume that a correct application server, which keeps on receiving Request messages tagged with a given req_id, and with different transaction instance identifiers, does not eventually send Decide messages with the *commit* indication to all the database servers for a transaction with $XID =< req\_id, ->$. In this case, the correct application server keeps on receiving those Request messages even after time t, after which, by the goodness assumption, the back-end database servers remain up and do not crash. After the receipt of whichever of these Request messages, the correct application server executes the `compute` primitive. Given that this primitive is nonblocking, after its execution, the correct application server executes the **repeat** loop in lines 3-6, by sending out, on a time-out basis, Prepare messages to all the database servers, tagged with req_id, and waiting for Vote replies from all of them. Given that after time t, the database servers are up and do not crash, the application server is correct, and the communication channels are reliable, each database server DB_i eventually receives the Prepare message, which triggers the execution of the `prepare` method (see lines 1-2 of the database server pseudocode). Also, given that all the statements within the `prepare` method are nonblocking, each database server DB_i eventually sends back the Vote reply to the correct application server, carrying the $\text{MIPT}^i_{req\_id}$ (see line 3 of the database server pseudocode). This message is eventually received by the correct application server from each DB_i since we are after time t and the communication channels are reliable.

After having collected $\text{MIPT}^i_{req\_id}$ from each DB_i, the application server executes the **repeat** loop starting in line 7 by initially testing whether $\mathcal{PC}$ holds. Such a test gets eventually completed by Lemma 3.2. The following two cases are possible: 1) The correct application server finds $\mathcal{PC}$ verified. Then, by Lemma 3.5, the correct application server eventually sends out Decide messages with the *commit* indication, tagged with req_id to all the database servers. Hence, the assumption is contradicted and the claim follows. 2) The correct application server finds $\mathcal{PC}$ not verified. In this case, no Decide message with the commit indication is sent out tagged with req_id.

Given that case 2 is the only one that does not contradict the assumption, we only need to prove that this case cannot occur indefinitely. Assume by contradiction that the correct application server, which keeps on receiving

Request messages tagged with req_id and different values of the instance identifier j, always falls in case 2. This implies that, after having collected $\text{MIPT}^i_{req\_id}$ from each DB_i, with $i \in [1,m]$, within the vote phase for whichever transaction $XID =< req\_id, j >$, the correct application server always finds $\mathcal{PC}$ not verified. Note that completion of the vote phase at the application server implies that every database server DB_i has received the Prepare message tagged with $XID =< req\_id, j >$, has executed the `prepare` method for this XID value, and has sent back the Vote message to the application server carrying the $\text{MIPT}^i_{req\_id}$. By Observation 2.5, after the completion of the `prepare` method at DB_i with input $XID =< req\_id, j >$, $\text{MIPT}^i_{req\_id}[j]$ exists. Hence, given that $\mathcal{PC}$ is not verified, it must hold that

$$\exists i \in [1,m] : \text{MIPT}^i_{req\_id}[j].state = abort. \quad (7)$$

However, whichever $\text{MIPT}^i_{req\_id}[j]$ can be found to keep the *abort* state value only in the following two cases: 1) The `prepare` method returns negatively at DB_i for the transaction $XID =< req\_id, j >$ (see lines 21-26 of the database server pseudocode). However, due to the goodness assumption of back-end databases, which are eventually able to prepare transactions if the correct application server keeps on retrying them, this case cannot occur indefinitely, thus eventually leading to a violation of expression (7). Hence, the assumption is contradicted and the claim follows. 2) The `resolve` method is executed at DB_i with $XID' =< req\_id, j' >$ as input, where $j < j'$, which leads to the creation of $\text{MIPT}^i_{req\_id}[j]$ and the setting of the corresponding state value to *abort* (see lines 31-34 of the database server pseudocode). By the database server pseudocode, the `resolve` method with $XID' =< req\_id, j' >$ as input is executed only upon receipt of a Resolve [$< req\_id, j' >$] message from some application server. By the application server pseudocode, an application server can send a Resolve [$< req\_id, j' >$] message only if that server has found $\mathcal{PC}$ verified in lines 8-9 and set $PreparedInstance$ to j', implying that

$$\forall i \in [1,m] : \text{MIPT}^i_{req\_id}[j'].state = prepared. \quad (8)$$

However, given that the correct application server keeps on receiving requests, retrying the corresponding transactions, and recollecting Vote messages for req_id indefinitely, and given that, by Observation 2.1, $\text{MIPT}^i_{req\_id}[j'].state$ once set to $prepared$ will not eventually change on any DB_i, the correct application server will eventually find $\mathcal{PC}$ verified for $XID' =< req\_id, j' >$. Hence, we eventually fall in case 1 and the claim follows. □

Termination T.1. *If the client issues a request, then, unless it crashes, the client eventually delivers a result.*[5]

5. 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". Also, the delivery of the result at the client side expresses that the `issue` method returns a result associated with a committed transaction.

Proof (By Contradiction). Assume by contradiction that a client issues a request, does not crash and does not eventually deliver a result. In this case, no Outcome message with the *commit* indication for a transaction associated with that request, i.e., tagged with the corresponding req_id, is received by the client. Hence, by lines 3-9 of the client pseudocode, it keeps on sending Request messages tagged with req_id as the request identifier, and with different transaction instance identifiers, to the application servers indefinitely. Hence, by reliability of communication channels, a correct application server will keep on receiving Request messages tagged with that req_id, and with different transaction instance identifiers, from the client. By Lemma 3.6, this application server will eventually send Decide messages with the *commit* indication, tagged with req_id, to all the back-end databases. Also, given that there is a time t after which all the database servers stop crashing and remain up, the Decide messages are resent on a time-out basis by the correct application server (see lines 27-30 of the application server pseudocode), and the communication channels are reliable, the database servers will eventually receive the Decide messages, which trigger the execution of the `decide` primitive at the databases (see lines 4-5 of the database server pseudocode). Also, given that this primitive is nonblocking and we are at time after t, the database servers will eventually reply with DecideACK messages tagged with req_id (see line 6 of the database server pseudocode), which are eventually received by the correct application server. Since this server does not crash, and the Request came from a client, by the application server pseudocode (see lines 31-36), an Outcome message, tagged with req_id, with the *commit* indication and the corresponding transaction result is eventually sent back and received by the client due to communication channel reliability. That result is therefore eventually delivered by the client since it does not crash. Hence, the assumption is contradicted and 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 (By Contradiction). Assume that a database server DB_i votes for a transaction $XID =< req\_id, inst\_id >$ (i.e., executes the `prepare` primitive in line 21 of its pseudocode) and never commits/aborts this transaction. There are two cases: 1) The `prepare` method returns with the *abort* state value. In this case, the transaction $XID = < req\_id, inst\_id >$ is aborted autonomously by DB_i. Hence, the assumption is contradicted and the claim follows. 2) The `prepare` method returns with the *prepared* state value, indicating that the transaction $XID = < req\_id, inst\_id >$ has been precommitted at DB_i. In this case, by lines 4-5 of the database server pseudocode and the semantic of the `decide` primitive, for DB_i not to eventually commit/abort that transaction it must occur that even after time t, when DB_i stops crashing and remains up, it does not eventually receive any Decide-message with either: 1) a *commit* indication for a transaction $XID' =< req\_id, - >$ or 2) an *abort* indication for the transaction $XID =< req\_id, inst\_id >$. In this case,

by **Task 4** of the database server pseudocode, DB_i keeps on sending Request messages on a time-out basis to the application servers indefinitely, tagged with req_id as request identifier, and with different transaction instance identifiers. Hence, by reliability of communication channels, a correct application server will keep on receiving Request messages tagged with that req_id, and with different transaction instance identifiers. By Lemma 3.6, this application server will eventually send a Decide message with the $commit$ indication to DB_i for a transaction $XID'' =< req\_id, - >$. Given that we are at time after t, by channel reliability, this message is eventually received by DB_i which eventually invokes `decide` for $XID'' =< req\_id, - >$ (see lines 4-5 of the database server pseudocode). By the specification of the `decide` primitive, DB_i commits/aborts $XID =< req\_id, inst\_id >$ independently of the instance identifier associated with XID''. Hence, the assumption is contradicted and the claim follows. □

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

Proof. The client delivers a result only if it receives an Outcome message with the $commit$ indication, tagged with the identifier associated with its request, namely req_id, from an application server. By lines 27-35 of the application server pseudocode, the application server sends the Outcome message with the $commit$ indication, tagged with some $XID =< req\_id, inst\_id >$, to the client only after it has sent Decide messages with the $commit$ decision for the corresponding transaction to all the database servers, and has received DecideACK messages from all of them. Hence, by lines 4-6 of the database server pseudocode, all the database servers have executed the `decide` primitive for the transaction $XID =< req\_id, inst\_id >$, with $commit$ as input parameter. Therefore, all we need to prove is that, prior to the execution of `decide` with the $commit$ indication: 1) that transaction had been prepared at all the databases and 2) had not been aborted at any database. Concerning point 1, by lines 8-14 of the application server pseudocode, the application server sends out the Decide messages tagged with $XID= < req\_id, inst\_id >$ only after having verified that both $\mathcal{PC}$ and $\mathcal{CC}$ hold, with $inst\_id = min(S)$. Hence, by the definition of the set S in (1), it must hold that $\forall i \in [1, m]$, $\mathrm{MIPT}^i_{req\_id}[inst\_id].state$ is found set to $prepared$ after the MIPTs have been collected in lines 3-6 of the application server pseudocode, or recollected in lines 22-25 of the application server pseudocode. However, by Observation 2.3, the value $prepared$ for that entry means that the corresponding transaction had been successfully prepared; hence, point 1 is proved. Concerning point 2, a transaction $XID =< req\_id, inst\_id >$ can be aborted by a database server only because the database server receives a Decide message with the $commit$ indication for a transaction having $XID' =< req\_id, inst\_id' >$, where $inst\_id \neq inst\_id'$, which is excluded by Lemma 3.3, or because the database server receives a Decide message with the $abort$ indication for that same transaction, which is excluded by Lemma 3.4. Hence, the claim follows. □

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

Proof. By lines 4-6 of the database server pseudocode, a database server can commit two different results, i.e., two different transactions $XID =< req\_id, inst\_id >$ and $XID' =< req\_id, inst\_id' >$ associated with the same client request only if it receives a Decide [$< req\_id, inst\_id >, commit$] message and a Decide [$< req\_id, inst\_id' >, commit$] message, where $inst\_id \neq inst\_id'$. In this case, some application servers must have sent the Decide messages with the $commit$ indication for the two different transactions $XID = < req\_id, inst\_id >$ and $XID' =< req\_id, inst\_id' >$. However, this is impossible by Lemma 3.3. Hence, the claim follows. □

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

Proof. By lines 4-5 of the database server pseudocode, a database server can decide $commit$ for a result, i.e., for a transaction $XID =< req\_id, inst\_id >$, only if it receives a Decide message with the $commit$ indication for that transaction from some application server. Whereas it can decide $abort$ only if it receives from some application server a Decide message with the $abort$ indication for that same transaction or a Decide message with a $commit$ indication for a different transaction associated with the same req_id. By Lemma 3.4, it follows that no two database servers can ever receive Decide messages with a $commit$ indication for two different transactions associated with the same req_id. Finally, by Lemma 3.5, no two application servers can send a Decide message with a $commit$ indication and a Decide message with an $abort$ indication for the same transaction. Hence, the claim follows. □

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. The client delivers a result only when it receives an Outcome message with the $commit$ indication, tagged with the identifier associated with its request, namely req_id, from an application server. By lines 27-35 of the application server pseudocode, the application server sends the Outcome message with the $commit$ indication, tagged with some $XID =< req\_id, inst\_id >$, to the client only after it has sent Decide messages with the $commit$ decision for the corresponding transaction to all the database servers and has received DecideACK messages from all of them. By lines 8-14 of the application server pseudocode, the application server sends out the Decide messages tagged with $XID =< req\_id, inst\_id >$ only after having verified that both $\mathcal{PC}$ and $\mathcal{CC}$ hold, with $inst\_id = min(S)$. Hence, by the definition of the set S in (1), it must hold that $\forall i \in [1, m]$, $\mathrm{MIPT}^i_{req\_id}[inst\_id].state$ is found set to $prepared$ after the MIPTs have been collected in lines 3-6 of the application server pseudocode, or recollected in lines 22-25 of the application server pseudocode. By lines 1-3 and lines 18-29 of the database server pseudocode, if the value of $\mathrm{MIPT}^i_{req\_id}[inst\_id].state$

is set to $prepared$, then DB_i must have received a Prepare message tagged with $XID =< req\_id, inst\_id >$ from some application server. Given that by lines 1-4 of the application server pseudocode, this server sends out Prepare messages tagged with $XID =< req\_id, inst\_id >$ only after it has received a Request message tagged with $XID =< req\_id, inst\_id >$, also carrying the request req as a parameter, the following two cases are possible: 1) the Request message tagged with $XID =< req\_id, inst\_id >$, and carrying req, came from the client, i.e., the category value associated with $inst\_id$ identifies the client process (see lines 6-7 of the client pseudocode). In this case, the claim trivially follows. 2) The Request message tagged with $XID =< req\_id, inst\_id >$, and carrying req, came from a database server, i.e., the category value associated with $inst\_id$ identifies some database server process (see lines 14-15 of the database server pseudocode). Given that the database server does not spontaneously issue requests, the transmission of the Request message tagged with $XID =< req\_id, inst\_id >$ can occur only in case, the `prepare` primitive has been previously executed with success for some transaction $XID' =< req\_id, - >$ at that database (see lines 11-17 of the database server pseudocode). Let t be the minimum time at which some database server DB_i executes the `prepare` primitive for some transaction $XID' =< req\_id, - >$. This happens only after DB_i has received a Prepare message tagged with $XID' =< req\_id, - >$ and req from some application server (see lines 1-3 of the database server pseudocode). Given that by lines 1-4 of the application server pseudocode, for this server to send out the Prepare message tagged with $XID' =< req\_id, - >$ and req, it must have previously received a Request message tagged with $XID'' =< req\_id, - >$ and req, this message must have been received from the client since no Request message can ever be sent out by any database server at time before t, tagged with some identifier $< req\_id, - >$. Hence, the claim follows. □

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

Proof. By lines 4-5 of the database server pseudocode, a database server commits a result, i.e., invokes the `decide` primitive with the $commit$ indication for a transaction $XID =< req\_id, inst\_id >$, only after it receives the Decide message from some application server, tagged with $XID =< req\_id, inst\_id >$ and with the $commit$ indication. By lines 8-14 and lines 27-28 of the application server pseudocode, an application server can send such a message only after having verified that both $\mathcal{PC}$ and $\mathcal{CC}$ hold, with $inst\_id = min(S)$. In this case, by the definition of the set S in (1), it must hold that $\text{MIPT}^i_{req\_id}[inst\_id] = prepared$, with $i \in [1, m]$. On the other hand, by Observation 2.3, whichever entry $\text{MIPT}^i_{req\_id}[inst\_id]$ can be set to maintain the $prepared$ state value only if DB_i has successfully precommitted the transaction $XID =< req\_id, inst\_id >$; hence, its vote for that transaction is yes. Thus, if a database server commits a transaction, then all database servers must have precommitted, i.e., must have voted yes for, that same transaction. □

4 Coping with Garbage Collection and Client Crashes

For protocol presentation simplicity, we did not address the removal of unneeded recovery information. Indeed, our protocol can be extended with the following garbage collection mechanism.

As soon as the database server receives the Decide message with $commit$, it can immediately discard the whole content of the proper MIPT, except the entry associated with the committed transaction, for which the status should be updated to a special value (e.g., committed). Hence, during the ACP, any application server, possibly trying to process a sibling transaction, becomes aware that one instance of transaction associated with that same client request was already committed. This prevents the application server from committing different sibling transactions, while still allowing retrieval of the result associated with the committed transaction.

Further, in order to enable the databases to also discard the result associated with the committed transaction (which is expected to occupy most of the storage required by any MIPT entry), upon delivery of the result, the client should send an acknowledgment to the application server, either via an apposite message, or by piggybacking it on the next issued request. The application server should in its turn notify all the databases to discard the result from the corresponding MIPT entry. The removal of the result from the MIPT is safe since the client's acknowledgment ensures that the whole end-to-end interaction was successfully completed. In-flight Request messages generated by client retransmissions could be simply discarded by the application server once it finds out that the corresponding MIPT reports a committed transaction, but stores no associated result.

Through the above mechanisms, the only recovery information still stored by the databases is the identifier of the sibling transaction that has been committed (namely, just a few bytes in practice). It is however noteworthy to highlight that *complete* removal of recovery information can be achieved if neither the client nor the databases retransmit additional instances of the same client request. To this end, the database servers should piggyback on the DecideACK message an additional flag conveying information on that they did not perform any request retransmission. The client should send an analogous notification to the application server (after delivering the result), either via an apposite message or by piggybacking this information on the next issued request. The application server would then notify the databases to fully discard any recovery information associated with that client request. This is safe since neither the client nor the database servers will ever retransmit that request, and the only transmitted Request message has been already processed.

Finally, if stable storage capabilities were admitted at the client side, our protocol could be straightforwardly extended to permit correct recovery of clients after crashes. This could be achieved by having the client simply log the Request message sent out for each issued request, as well as the first received Outcome message carrying a $commit$ indication. Upon recovery, if no Outcome message were found in the log with a $commit$ indication, the client should retransmit a Request message tagged with the same request identifier and an increased $instance\_number$ value.

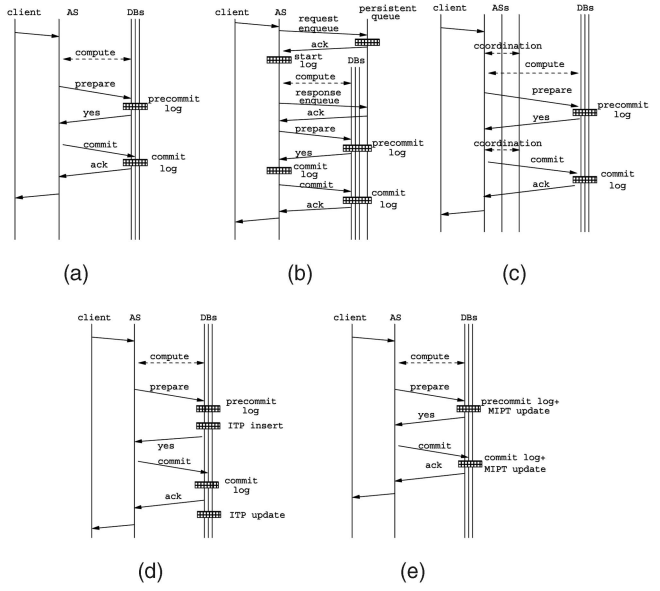

Fig. 6. Schematization of the (normal) behavior of the compared protocols (filled boxes represent eager disk accesses).

5 Performance and System Integration Issues

We compare our protocol with the following alternatives:

1. A baseline protocol, tolerating crashes, with recovery, of the back-end databases only, based on 2PC without logs on the coordinator (see Fig. 6a).

2. The persistent queue approach (PQ) [5], whose behavior is schematized in Fig. 6b. This approach enqueues the client request as the first action and uses START and PRECOMMIT logs at the application server, i.e., classical 2PC, to guarantee atomicity of the distributed transaction. This approach also requires the enqueue of the result of the data manipulation performed during the distributed transaction compute phase.

3. The primary backup replication scheme (PBR) in [14] and the asynchronous replication scheme (AR) in [12]. The behavior of both these e-Transaction protocols is schematized in Fig. 6c, 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 a consensus object (write once register), this holds for AR.

4. The e-Transaction protocol we have presented in [24], whose behavior is schematized in Fig. 6d. Analogously to the protocol presented in this paper, it avoids additional interactions in between the application server and other remote components (e.g., the coordination phase required by PBR and AR), and exploits additional recovery information locally logged by the database servers after the prepare phase, namely the Information on Transaction Processing (ITP). Compared to the baseline, this protocol requires an additional eager log at the database side for ensuring that the ITP is persisted before returning the Vote message to the application server. We refer this protocol to as ITP-proto.

We also show (see Fig. 6e) the schematized behavior of the protocol presented in this paper, which we will refer to as MIP-proto. Compared to the baseline, it does not impose any additional eager log since MIPT manipulations take place during the same disk accesses for precommit and commit logs.

TABLE 1
Protocols' Comparison

	# message rounds	# eager logs
baseline	2	2
PQ	4	4
PBR & AR	4	2
ITP-proto	2	3
MIP-proto	2	2

Actually, the work in [24] has presented performance models (parameterized to capture different scenarios, such as LAN versus WAN deployment of system components) to evaluate the benefits of ITP-proto. These models, and the related sensibility analysis, have shown how the avoidance of: 1) explicit coordination across middle-tier servers (required by PBR and AR) and 2) the interaction with the persistent queuing system (required by PQ), can increase system scalability and reduce end-to-end latency. Given that MIP-proto shows the same message pattern as ITP-proto, the analysis in [24] is still representative of the performance benefits that MIP-proto can provide in a wide spectrum of realistic system settings, when compared to the aforementioned solutions. Hence, the comparison we carry out in this section relies on two more classical cost metrics, namely, the number of (server side) message rounds needed before returning a response to the client, and the number of required eager logs. Table 1 reports the corresponding values for each of the considered protocols in the case of a (likely) nice run, not incurring failures or suspect of failures.[6] These values clearly show the effectiveness of our proposal. Specifically, MIP-proto achieves end-to-end reliability guarantees at the same cost of a baseline protocol tolerating only database server failures. We also note that MIP-proto exhibits the same number of server side message rounds as ITP-proto (in fact, as already mentioned, they show the same message pattern), and this number is lower than the one required by PQ and PBR/AR.

Additionally, MIP-proto avoids extermination-based fail-over, which is, instead, required in all the other schemes included in the comparison, and which was shown to negatively impact the latency of the end-to-end interaction in the presence of failure (or failure suspicions) [27].[7] Finally, it is also worth remarking that in MIP-proto, the transfer of the transaction coordinator role over the middle-tier takes place without explicit coordination among different application servers (transfer is triggered by the request retransmission logic). Hence, as soon as there is at least one available (i.e., up and working) application server replica, system availability depends only on the availability

6. 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 [19], 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. Hence, the COORDINATION phase can be assumed to require at least one round of messages in both PBR and AR.

7. Although the work in [27] copes with the simpler scenario involving a single back-end site, the performance results can be considered representative also for multiple sites.

of back-end database servers. This can be typically guaranteed via a set of solutions (e.g., [34], [36]).

Concerning the integration of MIP-proto with COTS systems, we find worthwhile to provide at least some hints on the practical aspects an implementor would face when adopting our solution. For what concerns the client retransmission logic, it could be integrated within a Web browser by relying on, e.g., a Java applet, Javascript technology or by exploiting browser proprietary technology, such as ActiveX in Microsoft's IE or ad hoc developed extensions in Mozilla Firefox. The same is true for what concerns the acquisition of information about the set of different application servers (among which to perform the selection before retransmitting the request). As an example, in case, the client logic is implemented via an applet, the set of edge servers could be made available upon download as a compile time filled array of entries. We also note that pragmatical approaches exist for supporting the extension of our protocol coping with clients having the ability to recover their state after a crash (see Section 4). In particular, message logging activities required at the client side could be implemented by exploiting, e.g., the approach shown in [28] for ActiveX technology, or via plug-in modules for open source browsers.

Regarding the implementation of the middle-tier logic, one could exploit the strong trend exhibited by modern application servers to be implemented on top of off-the-shelf industry middleware frameworks (e.g., Sun Microsystems' Java 2 Platform Enterprise Edition—J2EE, and Microsoft's .NET). For what concerns transaction management, current industry standard middlewares already embed ad hoc services, such as JTS for the J2EE platform, providing the abstraction of "Container Managed Transactions" [32]. In such a context, delegating the middleware container to host the application server logic of our protocol (which can be basically viewed as a nonconventional, MIPT-based transaction coordination scheme) appears as the most natural choice.

We note that the container can also be instrumented to transparently transform statefull application servers into stateless ones, thus widening the applicability of our proposal. In particular, the state maintained by middle-tier servers which is typically referred to as soft state (as it can be easily reconstructed, either automatically or with user help) could be migrated to the back-end tier by the container in a transparent manner for the overlying applications [32]. In a Web application, for instance, the container could store/retrieve the session state into/from the back-end tier. Users' identification within the session could then be achieved by inducing the client to automatically retransmit the user credentials (typically, provided only once by the user at the first interaction with the system) in each subsequent HTTP request. This can be achieved in a transparent way for the user, by using established mechanisms such as cookies, or parameters encoding in (hidden) HTML form parameters or hyperlinks' URLs. By storing the soft state within an ACID transaction on the back-end tier, it is made recoverable and available to any other application server replica. This has been shown to significantly decrease the impact of application servers' faults on the client perceived availability [15]. Also, manipulation of both soft state and back-end application data via MIP-proto provides strong mutual consistency. On the other hand, this technique requires additional interactions between the application server and the back-end layer for any client request entailing soft state update only.[8] An alternative option, also transparently supportable by the container, is to piggyback the soft state on the response for the client by using mechanisms such as cookies or parameters encoding. This scheme effectively avoids the need for any additional interaction with the back-end tier, at the cost of a (typically moderate) increase in the size of the exchanged requests/responses. On the other hand, the trade-off is toward weaker guarantees in terms of consistency of the soft state as, in case of application server crash, the latest update of the soft state could be lost. These weaker guarantees can be sometimes acceptable, for both scalability and performability purposes, especially in contexts where, as recently discussed in [6], they can be tackled with proper application level design techniques.

Finally, for what concerns the integration of the MIP scheme with COTS database systems, it could be relatively easily achieved in case of DBMSs 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/write 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 can be anyway used as the basis for the concurrency control scheme required to support the MIP semantic (which aims at increasing the concurrency level only among sibling transactions associated with the same client request). Actually, we have very recently developed [26] a complete prototype implementation of MIP facilities within the PostgreSQL (version 8.1.3) open source database system, which natively provides multiversion concurrency control. The implementation is based on modular, nonintrusive extensions of PostgreSQL data structures, and the addition of an ad hoc subsystem for managing MIPTs (implemented as kernel level database tables). In the same work, we have also presented an experimental evaluation (based on the TPC-W benchmark), showing the reduced overhead of MIP facilities (e.g., in terms of manipulation of MIPTs) on transaction execution latency, DBMS throughput, and storage usage.

We finally note that independently of the possibility to implement MIP facilities via modifications of existing DBMSs' internals, the MIP scheme is unique when compared to both traditional concurrency control, e.g., 2PL, Multi Versioning or OCC [5], and standard transactional demarcation schemes, e.g., ODBC or XA [33]. This is mainly due to the notion of sibling transactions, which is associated with demarcation and concurrency control schemes guaranteeing traditional isolation among different sibling transaction sets, and ad hoc management inside each single set.

8. In fact, in the statefull scenario, such an update is performed on data structures, e.g., session objects, locally maintained by the application server.

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 [16]. 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 [20] 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 PQs have also been proposed in literature [4], which are commonly deployed in industrial mission critical applications and supported by standard middleware technology (e.g., JMS in the J2EE architecture, Microsoft MQ, and IBM MQSeries). 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, PQs are typically not replicated at all these servers (because of the excessive overhead for maintaining their consistency). Hence, the log of the request/result on the PQ implies an additional interaction between remote systems, which might penalize the end-user perceived responsiveness.

There is some prior work addressing reliability in transactional systems [1], [10], which 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). Despite being originally designed for client-server applications, these proposals could also be 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 resubmitted 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).

Some works in literature have been aimed at reducing the messaging and logging overhead of the industrial standard ACP, namely 2PC. These encompass Presumed Commit/Abort [23], Early Prepare [31] and Coordinator Log [30] protocols. 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. In particular, we enrich the recovery information logged by the 2PC participants with the MIPT content, in order to ensure the testability of the distributed transaction outcome and retrieve the corresponding nondeterministic result.

The works in [2], [28], which provide extensions and generalizations of the client-server-tailored solutions in [21], address reliability in general multitier applications via the use of Interaction Contracts (ICs) between any two components, which specify permanent guarantees about state transitions, hence well-fitting requirements of statefull middle-tier applications. An IC is supported by logging sources of nondeterminism, e.g., exchanged messages, so 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 statefull parties. Hence, our solution copes with the alternative application server design paradigm.

The e-Transaction protocols in [12], [14], [24] respectively, require *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 [7]). Instead, our proposal requires no guarantee on the failure detection accuracy. In fact, in our protocol, failure detection is supported via a simple time-out-based mechanism operating in an asynchronous environment. Also, differently from our proposal, the solutions in [12], [14] require explicit coordination among the replicas of the application servers (i.e., an application server receiving the client request needs to notify the request to the replicas before performing any further operation), which imposes an additional overhead and reduces system scalability. 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.

The e-Transaction protocols in [11], [27] 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. Hence, differently from those solutions, our proposal can cope with the case

of, e.g., multiple parties involved in the same business process supported by the transactional application.

Our proposal has also relations with the notion of indulgence, expressed in [17] as the ability of a distributed protocol to preserve its correctness despite the possibility of errors in the failure detection module. In fact, our protocol can be seen as completely indulgent since it works correctly even when the underlying failure detector provides no guarantees on its accuracy.

Finally, the work in [13] presents a framework, named as X-ability, which defines correctness criteria based on exactly once execution semantic for replicated servers adhering the state machine model, whose actions may have side effects on external (transactional) components. Compared to X-ability, the e-Transaction framework (and hence, our protocol) is aimed at coping with the case of stateless replicated servers, where the only side effect associated with request processing occurs on external transactional components. In terms of X-ability, this means that no state machine's state is used to establish a context for subsequent request processing (i.e., the server side state machine actually has no internal state). Instead, request processing is made visible to subsequent requests only through external side effects.

7 Conclusions

In this paper, we have presented a distributed protocol which was formally proved to ensure e-Transaction guarantees in the general context of systems with multiple autonomous back-end databases. Compared to already existing solutions coping with this scenario, our proposal has the distinguishing feature of requiring no accuracy assumptions from the underlying failure detection mechanism. The protocol relies on an innovative distributed transaction management scheme, named as Multi-Instance Precommit, whose integration with conventional systems (e.g., database systems) has been hinted. Concerning performance, our proposal requires less than (or at most the same) message rounds and log operations as state-of-the-art solutions. Additionally, it requires no explicit coordination among server replicas over the middle tier, hence revealing highly scalable and well suited for large scale infrastructures.

Acknowledgments

An earlier version of this paper [25] appeared in the *Proceedings of the Fifth IEEE Symposium on Network Computing and Applications*, 2006. This paper was partially supported by the Pastramy (PTDC/EIA/72405/2006) project.

References

[1] R. Barga, D. Lomet, S. Agrawal, and T. Baby, "Persistent Client-Server Database Sessions," *Proc. Seventh Conf. Extending Database Technology (EDBT),* pp. 462-477, 2000.

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

[3] P.A. Bernstein and N. Goodman, "Multiversion Concurrency Control: Theory and Algorithms," *ACM Trans. Database Systems,* vol. 8, no. 4, pp. 465-483, 1983.

[4] P.A. Bernstein, M. Hsu, and B. Mann, "Implementing Recoverable Requests Using Queues," *Proc. 19th Conf. Management of Data (SIGMOD),* pp. 101-112, May 1990.

[5] P.A. Bernstein and E. Newcomer, *Principles of Transaction Processing: For The Systems Professional.* Morgan Kaufmann Publishers, Inc., 1997.

[6] G.D. Candia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," *SIGOPS Operating Systems Rev.,* vol. 41, no. 6, pp. 205-220, 2007.

[7] T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Comm. ACM,* vol. 43, no. 2, pp. 225-267, 1996.

[8] G.F. Coulouris and J. Dollimore, *Distributed Systems: Concepts and Design.* Addison-Wesley Longman Publishing Co., Inc., 1988.

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

[10] J. Freytag, F. Cristian, and B. Kähler, "Masking System Crashes in Database Application Programs," *Proc. 13th Conf. Very Large Data Bases (VLDB),* pp. 407-416, 1987.

[11] S. Frølund and R. Guerraoui, "A Pragmatic Implementation of e-Transactions," *Proc. 19th Symp. Reliable Distributed Systems (SRDS),* pp. 186-195, 2000.

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

[13] S. Frølund and R. Guerraoui, "X-Ability: A Theory of Replication," *Distributed Computing,* vol. 14, no. 4, pp. 231-249, 2001.

[14] S. Frølund and R. Guerraoui, "e-Transactions: End-to-End Reliability for Three-Tier Architectures," *IEEE Trans. Software Eng.,* vol. 28, no. 4, pp. 378-395, Apr. 2002.

[15] G. Gama, K. Nagaraja, R. Bianchini, R. Martin, W. Meira, Jr., and T. Nguyen, "State Maintenance and Its Impact on the Performability of Multi-Tiered Internet Services," *Proc. 23rd Symp. Reliable Distributed Systems (SRDS),* pp. 146-158, 2004.

[16] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1991.

[17] R. Guerraoui, "Indulgent Algorithms," *Proc. 19th Ann. ACM Symp. Principles of Distributed Computing (PODC),* pp. 289-297, 2000.

[18] R. Guerraoui and P. Kouznetsov, "On the Weakest Failure Detector for Non-Blocking Atomic Commit," *Proc. Int'l Federation for Information Processing (IFIP) 17th Int'l Conf. Theoretical Computer Science (TCS) (IFIP Conf. Proc.),* pp. 461-473, 2002.

[19] I. Keidar and S. Rajsbaum, "On the Cost of Fault-Tolerant Consensus When There Are No Faults," *SIGACT News,* vol. 32, no. 2, pp. 45-63, 2001.

[20] M. Little and S. Shrivastava, "Integrating the Object Transaction Service with the Web," *Proc. Second Int'l Workshop Enterprise Distributed Object Computing,* pp. 194-205, 1998.

[21] D.B. Lomet and G. Weikum, "Efficient and Transparent Application Recovery in Client-Server Information Systems," *Proc. 27th Conf. Management of Data (SIGMOD),* pp. 460-471, 1998.

[22] N.A. Lynch, *Distributed Algorithms.* Morgan Kaufmann, 1996.

[23] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System," *ACM Trans. Database Sstems,* vol. 11, no. 4, pp. 378-396, 1986.

[24] F. Quaglia and P. Romano, "Ensuring e-Transaction with Asynchronous and Uncoordinated Application Server Replicas," *IEEE Trans. Parallel and Distributed Systems,* vol. 18, no. 3, pp. 364-378, Mar. 2007.

[25] P. Romano and F. Quaglia, "Providing e-Transaction Guarantees in Asynchronous Systems with Inaccurate Failure Detection," *Proc. Fifth Symp. Network Computing and Applications (NCA),* pp. 155-162, 2006.

[26] P. Romano and F. Quaglia, "Integration and Evaluation of Multi Instance-Precommit Schemes within PostgreSQL," *Proc. 38th IEEE/IFIP Int'l Conf. Dependable Systems and Networks (DSN),* pp. 404-409, 2008.

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

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

[29] B.A. Shirazi, K.M. Kavi, and A.R. Hurson, *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE CS Press, 1995.
[30] J. Stamos and F. Cristian, "Coordinator Log Transaction Execution Protocol," *Distributed and Parallel Databases,* vol. 1, no. 4, pp. 383-408, 1993.
[31] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Trans. Software Eng.,* vol. 5, no. 3, pp. 188-194, May 1979.
[32] Sun Microsystems, JSR 220: Enterprise JavaBeans™, Version 3.0—Java Persistence API, May 2006.
[33] *Distributed TP: The XA+ Specification Version 2.* The Open Group, 1994.
[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. 28th Int'l Symp. Fault-Tolerant Computing Systems (FTCS),* pp. 422-431, 1998.
[35] H. Wu, B. Kemme, and V. Maverick, "Eager Replication for Stateful J2EE Servers," *Proc. Sixth CoopIS, DOA, and Ontologies, DataBases, and Applications of Semantics (ODBASE), OTM Confederated Conf.,* pp. 1376-1394, 2004.
[36] M. Xiong, K. Ramamritham, J. Haritsa, and J. Stankovic, "MIRROR: A State-Conscious Concurrency Control Protocol for Replicated Real-Time Databases," *Information Systems,* vol. 27, no. 4, pp. 27-297, 2002.

Paolo Romano received the MS degree in computer engineering from the University of Rome "Tor Vergata" in 2002 and the PhD degree in computer engineering from the University of Rome "La Sapienza" in 2007. Since February 2008, he has been a researcher at the INESC-ID in Lisbon, Portugal. His research interests are focused on dependability in parallel and distributed systems, and on performability modeling and evaluation. He regularly serves as a referee for several international conferences in the distributed systems and networking areas.

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 the Summer of 1999 to the Summer of 2000, he held an appointment as a researcher at the Italian National Research Council (CNR). Since January 2005, he has been as an associate professor in the School of Engineering of the University of Rome "La Sapienza," where he has previously worked as an assistant professor from September 2000 to December 2004. His research interests span from theoretical to practical aspects concerning distributed systems and applications, distributed protocols, middleware platforms, parallel discrete-event simulation, federated simulation systems, parallel computing applications, fault-tolerant programming, transactional systems, Web-based systems, and performance evaluation of software/hardware systems. He serves as a program committee member for prestigious international conferences. He has also served as the program cochair of PADS 2002, the program cochair of NCA 2007, the general chair of PADS 2008, and is currently a member of the PADS's Steering Committee. Starting from 2003, he regularly serves as a consultant expert of the National Center for Informatics in the Public Administration (CNIPA).