Transactional Exactly-Once

Svend Frølund and Rachid Guerraoui[*]

Hewlett-Packard Laboratories,

1501 Page Mill Road,

Palo Alto, CA 94304

July 14, 1999

Abstract

A *three-tier* application is organized as three layers. Front end clients (e.g., browsers) with which human user interact; middle-tier servers (e.g., web servers) that contain the core business logic of the application; and back-end database servers against which application servers perform transactions. Although three-tier applications are nowadays mainstream, they usually fail to provide sufficient reliability guarantees to the end users. Usually, ad-hoc replication and transactional techniques are developed for specific parts of the application, but these techniques are not combined to provide some global notion of reliability.

The aim of this paper is precisely to define a desirable, yet realistic, specification of end-to-end reliability in three-tier applications. We present the specification in the form of a problem called *Transactional Exactly-Once* which encompasses both safety and liveness properties in such environments. We also describe a practical protocol that solves the problem and we discuss its implementation and performances in a practical setting.

Keywords: End-to-end reliability, exactly-once, transactions, replication, fault-tolerance.

[*]Rachid Guerraoui is affiliated with Ecole Polytechnique Fédérale de Lausanne, Switzerland. This work was performed while he was Faculty Hire at Hewlett-Packard Laboratories, Palo Alto.

1 Introduction

A typical application, distributed or not, usually includes elements that handle *presentation*, *logic*, and *data*. A text editor that runs on a standalone PC offers a good example of these three application elements. Its user interface handles the user's keyboard input; its logic can then process the words, sentences, and paragraphs. It can store the results of the editing to a file, which then becomes persistent data. A *three-tier* application is one where the logical decomposition of the application is reflected both at the software and hardware level. A three-tier application has front-end clients (e.g., browsers), middle-tier application servers (web servers), and back-end database servers. Clients interact with users, and they send requests to application servers on behalf of users. Each request is sent to a single application server. The application server invoked by a client starts a transaction that captures the business logic of the application. The application servers may update multiple database servers within this transaction.

Until very recently, three-tier applications were at the leading edge of development. Today, they are mainstream. In particular, Web-based electronic-commerce applications typically follow the three-tier pattern. Consider for example a Web-based application to make travel arrangements. We will use this travel-booking example throughout the paper to ground the discussions. A user fills out a form through a browser (the client-tier) and then pushes a submit button. A web server (the middle-tier application server) receives that request and computes a result by interacting through a transaction with one or more database servers (the back-end tier). If no failure occurs, the user eventually receives that result.

Although three-tier applications are becoming nowadays mainstream, they usually fail to provide sufficient reliability guarantees to end users [1]. In practice, they typically provide *at-most-once* request-processing semantics. If an application server or database server fails during request processing, the user typically receives an error notification. The transactional properties of the server-database interaction guarantees that the permanent effect of the request is atomic. That is, either *all* or *nothing* happened to the persistent data. Actually, the user would like to have *all* happened. In fact, she cannot even tell if *all* or *nothing* in fact happened.[1] By manual request re-submission, the user can achieve *at-least-once* semantics. Intuitively, we would like to provide *exactly-once* request-processing semantics to end users. This would mask system failures and eliminate the need for manual request re-submission. However, the notion of *"exactly-once"* is not very precise as an end-to-end guarantee—it is primarily associated with the side-effect on the persistent data in databases. For example, we need to clarify the circumstances under which we can guarantee that the side-effect eventually happens. What if the client crashes immediately after the end user has pushed the submit button, but before the client can submit the request to a server or store it in stable storage ? We have to be more precise and specify how the exactly-once side-effect guarantee relates to the submission of requests, and reception of replies, by a client that may crash.

In this paper we give a formal definition of a practical end-to-end reliability guarantee with exactly-once flavor. We describe the guarantee as a distributed systems problem, called *Transactional Exactly-Once*. Essentially, the Transactional Exactly-Once problem requires that we mask failures in middle and back-end tiers. This objective is very sensible in practice. If we consider the travel application example, it means that if a user submits a travel request through a browser, then, unless the user's machine (i.e., the client) crashes, the application ensures that the request will be processed *exactly-once* and a result will eventually be received by the end-user. If the user's machine crashes, she can assume at-most-once semantics and it is up to her to figure out what indeed has happened. Moreover, the crash of one client does not affect the reliability of other clients: Transactional Exactly-Once prevents a crashed client from blocking the database for other clients.

Unlike existing reliability concepts for three-tier systems, Transactional Exactly-Once includes all the players (clients, application servers, and database servers) in a single specification of reliability. Existing approaches focus on specific parts of the overall reliability guarantee. A transactional system [2, 3] typically orchestrates the interactions between the application servers and the

[1]Thus, besides the tedium of manual request re-submission, the user may not even know if re-submission is a safe action and does not lead to duplication of transactional updates.

databases. It provides *all-or-nothing (at-most-once)* guarantees for this interaction, and does not capture the actual liveness guarantee that an end-user expects (i.e., *exactly-once*). Similarly, group communication mechanisms [4, 5, 6, 7] are typically defined with a group of replicated application servers in mind. They capture the interactions between a client and a group of replicated servers, but they do not address the safety of the interaction with third-party (non-deterministic) databases.

It is not trivial to combine the liveness properties of replicated services with the safety properties of transaction systems. The liveness of replicated services is concerned with the request-result interaction with clients. It does not address the liveness of actions that a service performs against third parties. Similarly, the safety properties of a transaction system only guarantees that either all or nothing happens, it does not guarantee that a third-party retry logic knows if all or nothing happened.

Transactional Exactly-Once addresses end-to-end reliability, and contains strong elements of both safety and liveness. The specification can be used as a metric to evaluate the correctness of reliability protocols for three-tier applications [8, 1], and help better understand how transactional and group communication mechanisms should complement each other in this context.

We show that our specification is realistic by describing a protocol that satisfies the specification. We designed our protocol with a very practical objective in mind: its implementation integrates with existing off-the-shelf technologies. In particular, we assume that the functionality of a database server is given: it is a stateful, autonomous resource that runs the XA interface [9] (the X/Open standard that database vendors are supposed to comply with in distributed, transaction-processing applications). Moreover, we assume that the client can be an applet that does not have access to client's disk, and therefore cannot serve as a traditional participant or coordinator in a distributed commit protocol. We discuss the implementation of our protocol and we point out its scalability features.

The rest of the paper is organized as follows. Next section presents a general model of a three-tier architecture. Section 3 presents the specification of Transactional Exactly-Once. Section 4 describes a protocol that satisfies the specification and proves its correctness. Section 5 presents the performances of an implementation of our protocol in a practical setting. Section 6 contrasts our specification and our protocol with related work. Finally, Section 7 concludes the papers by pointing out some open questions.

2 A Three-Tier Model

We consider a distributed system with a finite set of processes that communicate by message passing. Processes fail by crashing. At any point in time, a process is either *up* or *down*. A crash causes a transition from up to down, and a recovery causes the transition from down to up. The crash of a process has no impact on its stable storage. When it is up, a process behaves according to the algorithm that was assigned to it: processes do not behave maliciously. We say that a process is *correct* if eventually it is always up [2]. We assume that communication is *reliable*: that is, messages are not duplicated and a message m *sent* by a process p_i to a process p_j is eventually *received* by p_j, if both p_i and p_j stay up after m is sent by p_i.[3]

In the following, we outline our representation of the three types of processes in a three-tier application.

2.1 Clients

We model clients as processes, denoted by $c_1, c_2, \ldots c_k$ ($c_i \in Client$). We do not model the client-to-user interaction. We assume a domain, "Request", of request values, and we describe how requests in this domain are submitted to application servers. Clients have an operation *issue()*, which is invoked with a request as parameter. We say that the client *issues* a request when it invokes the operation *issue()*. The *issue()* primitive is supposed to *return* a result value from the domain "Result". When it does so, we say that the client *delivers* the result. We assume that each request and each result are uniquely identified and a client only delivers a result if it has issued a request.

[2]The period of interest of this definition is the duration of a request processing protocol.

[3]This assumption does not exclude link failures, as long as we can assume that any link failure is eventually repaired. In practice, the abstraction of reliable channels is implemented by retransmitting messages.

In practice, the request can be a vector of values. In the case of a travel application, the request typically indicates a travel destination, the travel dates, together with some information about hotel category, the size of a car to rent, etc. A corresponding result typically contains information about a flight reservation, a hotel name and address, the name of a car company, etc.

2.2 Application Servers

The application servers are processes, denoted by $a_1, a_2, \ldots, a_m$ ($a_i \in \mathit{AppServer}$). Application servers are *stateless* in the sense that they do not maintain states accross request invocations: requests do not have side-effects on their states, only on the database state. Thus, a request cannot make any assumption about previous requests in terms of application-server state changes. Having stateless application servers means that we can replicate them without synchronizing state updates. We do not model the chained invocation of application servers. In our model, a client invokes a single application server, and this server does not invoke other application servers. Chained invocation does not present additional challenges from a reliability standpoint because application servers are stateless. We ignore this aspect in our model to simplify the discussion.

In terms of transactions, we only explicitly model the commitment processing, not the business logic or SQL queries performed by application servers. We use a function, called *compute()*, to abstract over the (transient) database manipulations performed by the business logic. For example, in our travel example, *compute()* would query the database to determine flight and car availabilities, and perform the appropriate bookings. However, the *compute()* function does not commit the changes made to the database. Instead, it returns a result, which application servers can use to commit the transaction. This allows us to explicitly model the commit processing without modeling the SQL processing. A result is a value in the "Result" domain, and it represents two aspects of transaction processing: (1) information computed by the business logic, such as reservation number and hotel name, that must be returned to the user, and (2) a transaction identifier that an application server can use to commit the updates performed against the database within *compute()*.

Since the commitment processing can fail, we may call *compute()* multiple times for the same request. However, *compute()* is non-deterministic because its result depends on the database state. We assume that each result returned by *compute()* is non-nil and can be used for commit processing. In particular, we model user-level aborts as regular result values. A user-level abort is a logical error condition that occurs during the business logic processing, for example if there are no more seats on a requested flight. Rather than model user-level aborts as special error values returned by *compute()*, we model them as regular result values that the databases then can refuse to commit.

2.3 Database Servers

We represent database servers as processes, denoted by $s_1, s_2, \ldots, s_n$ ($s_i \in Server$). Since we want our approach to apply to off-the-shelf database systems, we view a database server as an XA [9] engine. In particular, a database server is a "pure" server: it does not invoke other servers, it only responds to invocations. We do not represent full XA functionality, we only represent the transaction commitment aspects of XA (*prepare* and *commit*). We use two primitives, *vote()* and *decide()*, to represent the transaction commitment functionality:

- The *vote*() primitive takes as a parameter a result value computed by an application server, and returns a vote. A vote is a value in the domain Vote = $\{\mathsf{yes}, \mathsf{no}\}$. If the function returns yes, we say that the database server *accepts* the result. Roughly speaking, a yes vote means that the database server agrees to commit the result.[4]

- The *decide*() primitive takes two parameters: a result and an outcome. An outcome is a value in the domain Outcome = $\{\mathsf{commit}, \mathsf{abort}\}$. The *decide*() primitive returns an outcome value such that: (a) if the input outcome value is abort, then the returned value is also abort; and (b) if the database server has accepted a result, and the input outcome value is commit, then the returned value is also commit. If the returned outcome value is abort (resp. commit), we say the database server aborts (resp. commits) the result.[5]

[4]In terms of XA, the *vote*() primitive corresponds to a prepare operation. The results play also the role of transaction identifiers.

[5]The *decide*() primitive is patterned after the commit operation in the XA interface.

3 The Transactional Exactly-Once Problem

Roughly speaking, Transactional Exactly-Once requires that, whenever a client issues a request, then unless it crashes, there is a corresponding result computed by an application server, the result is committed at every database server, and eventually delivered by the client. The servers might go through a sequence of aborted results until a good result is committed and the client delivers it. Ensuring database consistency requires that all database servers agree on the outcome of every result, either they all abort the result or they all commit that result. Client-side consistency requires that a result is only returned to the client if it has been committed by all database servers.

In the following, we first give an intuitive view of the Transactional Exactly-Once Problem, and then we give a formal specification of the problem.

3.1 Intuition

Consider our canonical online travel-booking application. A user fills out a form about a travel request, indicating the travel destination, the travel dates, together with additional preference information about airline, hotel category, and type of vehicle to rent at the destination. After completing the form, the user pushes a *submit* button, and a request is sent to an application server. The application server queries one or more databases within a transaction to fulfill the travel request. Or, in our terminology, the application server computes a result for the request. The result is typically composed of several partial results, (1) one for the flight reservation, (2) one for the hotel reservation (e.g., a hotel name and address) and (3) another one for renting a car (e.g., the name of a car company). Each of these partial results corresponds to an update of a database server, e.g., s_1, s_2, and s_3, and the result can only be delivered by the client if every database server commits the result.

Since database servers might be temporarily down, or just unable to commit an update due to some execution problem (e.g., deadlock) or a logical problem (e.g., no more seats in a flight), the database servers might need several tries (several intermediate results) before reaching a commit decision about a valid result to be returned to a client. It is important that if any server aborts a result, all servers abort that result before proceeding to the next try. To see why this is important, assume that the user is planning a trip to Paris. Paris has two international airports: Roissy and Orly. If the first try fails to book a seat in a flight to Roissy, it is important to also cancel the reservation of a car from that airport. This should be ensured no matter how many failures occur in the system. It is possible that the second try will succeed in finding a seat in a flight to Orly, and the car should then be reserved there. Since the user might probably be unhappy to be charged twice (even if she ends up with two tickets), it is important to guarantee that no database server commits more than one result (for the same request).

The desired guarantee has a notion of exactly-once: we want to ensure that the user does not get charged twice, but we also want to ensure that the user eventually gets a result, not just an error notification. Of course, the result may be that the flight is full, but at least the user then knows what happened. The guarantee also has a notion of transactional consistency: we cannot update the database servers independently since there may be some application-level dependencies between the database updates. Booking a car and a flight relative to the same airport is an example of such a dependency.

3.2 Specification

For the sake of presentation simplicity, we consider here only one client and assume that the client issues only one request. We hence omit explicit identifiers to distinguish different clients and different requests, together with identifiers that relate different results to the same request.

We define the Transactional Exactly-Once problem with three categories of properties: *Termination*, *Agreement*, and *Validity*. Termination captures liveness guarantees by preventing blocking situations, Agreement captures safety guarantees by ensuring the consistency of the client and the databases, and Validity restricts the space of possible results to exclude meaningless ones.

- **Termination.**

  (T.1) If the client issues a request, then unless it crashes, it eventually delivers a result; (T.2) If any database server votes for a result, then it eventually commits or aborts the result.

- **Agreement.**

  (A.1) No result is delivered by the client unless it is committed by all database servers; (A.2) No database server commits more than one result; (A.3) No two databases decide differently on the same result;

- **Validity.**

  (V.1) If the client issues a request and delivers a result, then the result has been computed by an application server with the request as a parameter; (V.2) No database server commits a result unless all database servers have voted yes for that result.

Termination ensures that (T.1) a client does not remain indefinitely blocked. This provides *at-least-once* request processing guarantee to the caller of the *issue*() primitive, and frees the caller from the burden of having to retry requests. Termination also ensures that (T.2) no database server remains blocked forever waiting for the outcome of a result, no matter what happens to the client. This *non-blocking* property is also important because a database server that has voted yes for a result might have locked some resources. These remain inaccessible until the result is committed or aborted.[6] The agreement property ensures the consistency of the result (A.1) and the databases (A.3). It also guarantees *at most-once* request processing (A.2). The latter property does not prevent a database server from committing (resp. aborting) more than once the same result, which is acceptable by most distributed database systems we know about. The first part of Validity (V.1) excludes trivial solutions to the problem where the client *invents* a result. The second part (V.2) conveys the classical constraint of transactional systems, that no result can be committed if at least some database server refuses to do so. It is important to notice that Transactional Exactly-Once

[6]Subproperty T.2 corresponds to the termination property of non-blocking atomic commitment protocols [10].

```
function issue(Request request)
    Result result;                                              /* expected result */
    AppServer $a_1$ := thePrimary;                            /* the default primary */
    list of AppServer alist := theAppServers;         /* list of all application servers */
    TimeOut period := thePeriod;                                  /* back-off period */

    send [Request,request] to $a_1$;
    while (true) do
        set-timeout-to period;
        wait until (receive [Deliver,result]);
            return(result);                          /* delivers the result and exits */
        on-timeout
            send [Request,request] to alist;
```

Figure 1: Client protocol

expresses safety and liveness requirement on the databases, even if the client or any application server crash, e.g., the crash of a client does not prevent other clients from accessing the databases, nor does it lead to any inconsistency among databases.

4 A Transactional Exactly-Once Protocol

In the following, we describe a protocol that solves the Transactional Exactly-Once problem. Our protocol consists of a set of sequential algorithms. We first give an overview of the protocol, then we describe the protocol in more detail and give the pseudo-code of its component algorithms. We describe the algorithms in Figure 1, Figure 2, Figure 3, and Figure 4.

4.1 Overview

Our Transactional Exactly-Once protocol consists of several parts. One is executed at the client, one is executed at the application servers, and one at the database servers. The client interacts with the application servers, which themselves interact with database servers. Basically, the client retransmits the request to the application servers until it receives back a result (Clients do not have access to stable storage). The application servers execute a variant of a primary-backup replication protocol [11][7]. The primary application server computes a result for the client's request and orchestrates a distributed atomic commitment protocol among the database servers to commit or abort that result. The primary application server stores crucial information (the result and the outcome of the voting phase of the commitment protocol) at the backups. If the primary fails, one of the backups takes over. The database server responds to messages sent by application servers by sending back votes or decision acknowledgments.

4.2 Assumptions

After being issued by a client, a request is processed without further input from the client. Furthermore, the client issues requests one-at-a-time and, although issued by the same client, two consecutive requests are considered to be unrelated. Clients cannot communicate directly with databases, only through application servers. We assume that at least one application server is always up. This assumption is needed here to keep the protocol relatively simple: we do not deal with application server recovery. Similarly, and for the same reason, we do not deal with client recovery.

Every application server has access to a local failure detector module which provides it with information about the crash of other application servers. Let a_1 and a_2 be any two application servers. We say that server a_2 *suspects* server a_1 if the failure detector module of a_2 suspects a_1 to have crashed. We abstract the suspicion information through a predicate *suspect()*. Let a_1 and a_2 be any two application servers. The execution of $\mathit{suspect}(a_1)$ by server a_2 at t returns `true` if and only if a_2 suspects a_1 at time t. Our protocol requires that an application server's failure detector with respect to other application servers is perfect in the sense of [12]. In other words, we assume that the following properties are satisfied: *Completeness:* if any application server crashes at time t, then there is a time $t' > t$ after which it is permanently suspected by every application server; *Accuracy:* no application server is suspected unless it has crashed. Since we consider a a variant of

[7]Primary replication schemes do however typically not include an interaction with a third-party component, e.g., a database.

a primary-backup replication scheme, the assumption of perfect failure detection is not surprising to prevent concurrent primaries [11].

We do not assume perfect failure detection between clients and servers. Clients are free to re-submit requests at any time, and they use a simple timeout mechanism to determine when to re-submit a request. This design decision reflects our expectation that clients can communicate with servers across the Internet, and we do not want to assume perfect failure detection across the Internet.

We assume that all database servers always recover after crashes, and eventually stop crashing. In practice, this assumption needs only hold during the processing of a request. For example, in practice, we only need to assume that for each request, every database server will eventually stay up long enough to successfully commit the result of that request. Ensuring the recovery of every database server (within a reasonable time delay) is typically achieved by running databases in clusters of machines [13, 14]. With a cluster, we can ensure that databases always recover (within a reasonable delay), but we must still assume that the system reaches a "steady state" where database servers stay up *long enough* so that we can guarantee the progress of the request processing. In an asynchronous system however, with no explicit notion of time, the notion of *long enough* is impossible to characterize.

Furthermore, we assume that there is a time after which every result computed by an application server is accepted by all database servers. In practice this means that there is a time after which all transactions run to completion. If we take our canonical example of online travel arrangements, our assumption does not mean that there will eventually be a seat on a full flight. It means that an application server will eventually stop trying to book a seat on a full flight, and instead execute a transaction that can actually run to completion, for example a transaction whose result informs the user of the booking problem.

In many cases, servers will acknowledge receipt of messages. We assume that the receiver of an acknowledgment message can correlate it with the message being acknowledged. This can be achieved by appropriate tagging of acknowledgment messages. However, to simplify the presenta-

```
function ServerProtocol(Bool recovery)
    Result result;                               /* result from an application server */
    Outcome outcome;                             /* outcome of a result: commit or abort */
    AppServer $a_i$;                             /* a primary */
    list of Appserver alist := theAppServers;    /* list of all application servers */

    if (recovery) then                           /* distinguish recovery from the initial starting case */
        send [Ready] to alist;                   /* recovery notification */
    while (true) do
        cobegin
            || wait until (receive [Result,result] from $a_i$)
                send [Vote,result,vote(result)] to $a_i$;
            || wait until (receive [Decide,result, outcome] from $a_i$);
                decide(result, outcome);
                send [AckDecide,result, outcome] to $a_i$;
        coend
```

Figure 2: Database server protocol

tion, we do not describe this tagging and correlation in our protocol.

Finally, we assume a closed system: the only entities in the system are the client, the application servers, and the database servers. Moreover, these entities behave according to their respective protocols.

4.3 Pseudo-Code

We use **send** and **receive** primitives to represent message passing. For example, if p is a process, the statement "**send** [Request,*request*] **to** p" captures the action of sending the message [Request,*request*] to process p. A message [Request,*request*] is of type "Request" and contains the value *request*. If the destination of a message is a list of processes, a send operation multi-casts the message to all processes in the list (we make no assumptions about the indivisibility of such operations).

The statement "**receive** [Deliver,*result*] **from** a" captures the action of waiting for a message of type "Deliver". When such a message arrives, the variable *result* is assigned to the contents of the message, and the variable a is assigned to the sender's identity. We also use the receive primitive without a "from" part if we do not need to assign the sender's identity to a variable.

As a convenient notation, we introduce the predicate *received()*. Let p_i and p_j denote any two processes and *plist* a list of processes. Then, the execution by process p_i of "*received*([AckDecide]) **from** p_j" is true if p_i has received a message of the form [AckDecide] from p_j. Similarly, the execution by process p_i of "*received*([AckDecide]) **from** *plist*" evaluates to true if p_i has received [AckDecide] from every process in *plist*.

Besides message passing, we also use various synchronization primitives. We use "**wait until**" statements to wait for a collection of events to occur. Events can be the reception of messages and detection of failures. We use **and** and **or** combinators to specify these event sets. Moreover, we can bound the waiting time with timeouts. We use the statement **set-timeout-to** to set the expiration time of a timer, and the statement **on-timeout** describes the actions to take if and when the timer expires.

Traditional control structures, such as branches and loops, are used with their usual semantics. In addition, we also use **cobegin** and **coend** to capture concurrent executions. The **cobegin** statement terminates when any of the contained activities terminates. We use "=" to compare values for equality and ":=" for assignment.

In addition to the domains introduced in Section 2, we also use a domain called Bool, which contains the Boolean values true and false. Furthermore, the domain Timeout contains values, such as real numbers, that can be used to describe elapsed time.

4.4 Protocol Description

The client part of the protocol is encapsulated within the implementation of the *issue()* primitive (Figure 1). This primitive is invoked with a request as an input parameter and is supposed to eventually return a result. The client executes a retransmission protocol to ensure that, despite crashes of some application servers, at least one server receives the request. To optimize the failure-free scenario, the client does not send the request to all backups unless it does not receive a result after a back-off period.

The application servers execute a primary-backup scheme, which ensures that at least one

```
  function primaryProtocol(Result $result$,Outcome $outcome$)
      Client $c$;                                                          /* the client */
      Request $request$;                                      /* request from the client */
      AppServer $a_i$;                                               /* a member of blist */
      list of AppServer $blist$ := backups;    /* list of the backups of the process executing this code
*/
      Server $s_k$;                                                  /* a database server */
      list of Server $slist$ := theServers;                  /* list of all database servers */

1     while (true) do
2         wait until (receive [Request,$request$] from $c$);
3             if ($outcome$ = commit) then
4                 send [Decide,$result$] to $c$;
5             else
6                 $result$ := compute($request$);
7                 send [Result,$result$] to $blist$;
8                 wait until (for every $a_i \in blist$:
                              (receive [AckResult] from $a_i$) or (suspect($a_i$)));
9                 send [Result,$result$] to $slist$;
10                wait until (for every $s_k \in slist$:
                              (receive [Vote,$result, vote_k$] or [Ready] from $s_k$));
11                if (for every $s_k \in slist$: (received([Vote,$result$,yes]) from $s_k$)) then
12                    $outcome$ := commit;
13                send [Decide,$result, outcome$] to $blist$;
14                wait until (for every $a_i \in blist$:
                              (receive [AckDecide] from $a_i$) or (suspect($a_i$)));
15                repeat
16                    send [Decide,$result, outcome$] to $slist$;
17                    wait until (for every $s_k \in slist$:
                              (receive [AckDecide] or [Ready] from $s_k$));
18                until (received([AckDecide]) from $slist$);
19                if ($outcome$ = commit) then
20                    send [Decide,$result$] to $c$;
```

Figure 3: Primary application server protocol

application server is available to compute a result and interact with the database servers: if a primary crashes, a backup takes over. The replicas coordinate their activities in such a way that at most one application server is primary at any time and hence at most one result is computed at any given time (Figure 3 and Figure 4). None of the client or the application server protocols contain explicit recovery procedures. This is not surprising since these entities are not supposed to recover after crashes.

We say that an application server is primary (resp. backup) at time t, if the application server is up at time t and it is executing the code of Figure 3 (resp. Figure 4) at t. The default primary server is a_1. In other words, unless it crashes, a_1 executes the code of Figure 3. The parameters passed to the function $primaryProtocol$ captures the status of request processing when the current primary became primary. The initial primary (a_1) invokes the function with a status of $(\mathsf{nil}, \mathsf{abort})$. Thus, when a_1 executes the code in Figure 3, the initial value of $result$ is nil and the initial value of *outcome* is abort.

Every other application server a_i starts by executing the code of Figure 4. If a_1 crashes, then a_2 is supposed to take over as the new primary, unless it itself crashes, in which case a_3 becomes the primary, etc. Roughly speaking, when a_i is the primary, then it *stores* crucial protocol information at all application servers $a_{i+1}, a_{i+2}, \ldots, a_m$ (Figure 3). An application server a_i does not become primary unless all application servers $a_1, a_2, \ldots, a_{i-1}$ have crashed, and a_i makes sure that any information it might have, is also shared by all backups that are still up. (Figure 4).

The primary application server orchestrates a distributed atomic commitment protocol to ensure that all database servers agree on the outcome of every result. If the primary crashes, a backup takes over and terminates the protocol.[8]

Figure 2 illustrates the functionality of database servers. A database server is a pure server (not a client of other servers): it waits for messages from application servers to either vote or decide on results.

[8]The resulting scheme can be viewed as a Two-Phase Commit protocol [10] with a replicated coordinator [15]. The coordinator does not store crucial information on disk, but rather uses the backup replicas as a stable storage.

```
function backupProtocol()
    Result $result$ := nil;                                   /* expected result */
    Outcome $outcome$;                    /* outcome of a result: commit or abort */
    AppServer $a_i$;                                   /* an application server */
    list of AppServer $plist$ := primaries;   /* list of the primaries of the process executing this code */
    list of AppServer $blist$ := backups;   /* list of the backups of the process executing this code */
    Server $s_k$;                                           /* a database server */
    list of Server $slist$ := theServers;             /* list of all database servers */

    while (true) do
        cobegin
        || wait until (receive [Result,$result$] from $a_i$);
            send [AckResult] to $a_i$;
        || wait until (receive [Decide,$result, outcome$] from $a_i$);
            send [AckDecide] to $a_i$;
        || wait until (for every $a_i \in plist$: (suspect($a_i$)));
            if ($result \neq$ nil) then
                send [Decide,$result, outcome$] to $blist$;
                wait until (for every $a_i \in blist$:
                            ((receive [AckDecide] from $a_i$) or (suspect($a_i$)));
                repeat
                    send [Decide,$result, outcome$] to $slist$;
                    wait until (for every $s_k \in slist$:
                        (receive [AckDecide] or [Ready] from $s_k$));
                until (received([AckDecide]) from $slist$)
            primaryProtocol($result, outcome$);                 /* become primary */
        coend
```

Figure 4: Backup application server protocol

A database server executes the function *ServerProtocol*. The parameter passed to this function indicates whether the function is called initially or during recovery. The parameter is bound to the variable *recovery* that is then used in the body of *ServerProtocol* to take special recovery actions. During recovery, a database server informs the application servers about its "coming back".[9]

5 Protocol Correctness

In the following, we show that the protocol composed of the algorithms described in Figure 1, Figure 2, Figure 3, and Figure 4, solves the Transactional Exactly-once problem.

Lemma 1. *No primary application server remains blocked forever in one of the wait statements of line 8, 10, 14 and 17, in Figure 3.*

PROOF. Assume by contradiction that some primary application server a_j remains blocked forever in one of the *wait* statements of Figure 3. By the algorithm of Figure 3, there are two cases to consider: (1. lines 8 and 14 in Figure 3) a_j is blocked waiting either to receive a message of type AckResult or AckDecide as a response to a message (of type Result or Decide) sent to a backup a_i, or to suspect a_i; (2. lines 10 and 17 in Figure 3) a_j is blocked waiting either to receive a message of type Vote or AckDecide as a response to a message of type Result or Decide sent to a database server s_k, or to receive a message of type Ready from s_k.

Consider case 1. If (1.1) a_i has not crashed, then by the assumption of reliable channels and the algorithm of Figure 4 (lines 4 and 6), a_i sends back a message of type AckResult or AckDecide and a_j unblocks. If (1.2) a_i crashes, then by the completeness property of the failure detector, a_j suspects a_i, which unblocks a_j. Both subcases 1.1 and 1.2 lead to a contradiction.

Consider case 2. If the database server s_k has been up since a_j sent its message (of type Result or

[9]This notification is an abstract representation of a failure detection scheme where application servers can always tell when a database server has crashed and recovered. Such a failure-detection scheme is a realistic assumption. In practice, application servers would detect database crashes because the database connection breaks when the database server crashes. Application servers would receive an exception (or error status) when trying to manipulate the database. Furthermore, we can implement that failure-detection scheme without requiring the database servers to know the identity of the application servers.

Decide) and remains up, then by the assumption of reliable channels and the algorithm of Figure 2 (lines 6 and 9), s_k sends back a message of type Vote or AckDecide to a_j, which unblocks a_j. If s_k has crashed since a_j sent its message, then by the assumption that all database servers are correct, and the algorithm of Figure 2 (line 2), s_k eventually recovers and sends a message of type Ready to a_j, which unblocks a_j. □

Lemma 2. *Let t be any time. (1) At most one application server is the primary application server at t, and (2) there is a time $t' \geq t$ after which some application server remains primary forever.*

PROOF. Let t be any time. Consider property 1. Assume by contradiction that two different application servers, a_i and a_j, are primary at t, and assume that $i < j$. By the assumption that the default primary is a_1, a_j initially executes the algorithm of Figure 4. By that algorithm, the only possibility for a_j to become primary is by suspecting a_i (line 7 in Figure 4). By the accuracy property of the failure detector, a_i must have crashed by time t: a contradiction with the assumption that a_i is primary at time t (remember that an application server is said to be primary at time t if, and only if, at time t, it is both up and executing the code of Figure 3).

Consider property 2. Assume that i is the smallest integer such that a_i is always up. By our assumption that at least one application server is always up, such an integer i does exist and $1 \leq i \leq m$. We distinguish two cases: (2.1) $i = 1$ and (2.2) $i > 1$. In case 2.1, a_1 is always up. By the algorithm of Figure 3, a_1 remains primary forever (it will remain within the infinite loop, i.e., lines 1 to 20 in Figure 3).

Consider case 2.2. Per assumption, every a_k, such that $k < i$, eventually crashes. By the completeness property of the failure detector, a_i eventually suspects every application server a_k for which $k < i$. If (2.2.1) *result* is nil, then a_i directly becomes primary (line 15 in Figure 4) and, by the algorithm of Figure 3, a_i remains primary forever. If (2.2.2) *result* is not nil, then a_i sends [Result,*result*, *outcome*] to each of its backups a_j (line 9 in Figure 4) and then waits either to receive [AckDecide] from a_j or to suspect a_j (line 10 in Figure 4). By the completeness property of the failure detector, the assumption of reliable channels, and the algorithm of Figure 4, a_i eventually sends [Decide,*result*, *outcome*] to every database server (line 12 in Figure 4). By the assumption that all database servers are correct, the assumption of reliable channels, and the algorithm of Figure 2, eventually, a_i receives [Decide,*result*, *outcome*] from every database server (line 14 in Figure 4), and becomes primary. By the algorithm of Figure 3, a_i remains primary forever. In both subcases 2.2.1 and 2.2.2, there is a time t' after which a_i remains primary forever. □

Lemma 3 (Termination T.1). *If the client issues a request, then unless it crashes, the client eventually delivers a result.*

Proof. Assume by contradiction that the client issues a request, remains up forever, but never delivers a result. Let t be the time after which the primary application server remains up forever, and all database servers remain up forever and accept all results computed by the primary application server. By Lemma 2 and our assumptions, time t does exist. As we assume (by contradiction) that the client does not receive back a result, then by the algorithm of Figure 1, the client keeps sending the request to all application servers (line 7 in Figure 1), and in particular, the client does so after time t. By the assumption of reliable channels, the primary application server eventually receives the request (line 2 in Figure 3). There are two cases to consider: (1) the value of *outcome* at the primary is commit, or (2) the value of *outcome* at the primary is abort. In case 1, the primary sends back a result to the client (line 4 in Figure 3), and by the assumption of reliable channels, the client receives that result and delivers it: a contradiction.

Consider case 2. The primary computes a result and sends it to all its backups (lines 6 and 7 in Figure 3). By Lemma 1, the primary cannot remain blocked waiting for an AckResult message from some backup. Hence the primary sends the result to every database server (line 9 in Figure 3). By the assumption of reliable channels and the assumption that all database servers remain up after time t, every database server receives the result. By the algorithm of Figure 2 and the assumption that, after time t, all database servers are up and accept every result, every database server sends back a message [Vote,*result*,yes] to the primary (line 6 in Figure 2). Hence the primary eventually assigns *outcome* to commit and sends message [Decide,*result*,commit] to all database servers. (lines

12 and 13 in Figure 3). By Lemma 1, and the algorithm of Figure 3, the primary eventually exits from the *repeat* loop of Figure 3 (line 15 to 19) and sends back a result to the client (line 20 in Figure 3). By the assumption of reliable channels, the client eventually receives the result and delivers it (lines 4 and 5 in Figure 1): a contradiction. □

Lemma 4 (Termination T.2). *If any database server votes for a result, then it eventually commits or aborts the result.*

PROOF. Let s_k be any database server that accepts a result r. To accept r, s_k must have received [Result,r] from a primary application server (lines 7 and 8 in Figure 2). We distinguish two cases: (1) the primary remains up after it has sent [Result,r] to s_k, or (2) the primary crashes after sending that message to s_k. Consider case 1. By Lemma 1 and the algorithm of Figure 3, the primary eventually sends [Decide,r,commit] or [Decide,r,abort] to s_k (line 16 in Figure 3), and, by the assumption of reliable channels, s_k eventually receives the message and commits or aborts r accordingly (lines 7 and 8 in Figure 2).

Consider case 2. By the completeness property of the failure detector and the assumption that at least one application server is always up, a backup a_j eventually suspects the primary and proceeds executing line 8 of Figure 4. Since s_k has accepted r from a primary, then by the algorithm of Figure 3 (lines 7 and 8), a_j must have received r from the primary (and $r \neq$ nil). By the algorithm of Figure 4, and since r is not nil, a_j sends either [Decide,r, *outcome*] to s_k (line 12 in Figure 4), By the assumption of reliable channels, s_k eventually receives the message and commits or aborts r accordingly (lines 7 and 8 in Figure 2). □

Lemma 5 (Agreement A.1). *No result is delivered by the client, unless the result is committed by all database servers.*

PROOF. Consider property A.1 and assume the client delivers a result r. By the algorithm of Figure 1 (line 4), the client must have received r from an application server. By the algorithms of Figure 3 and Figure 4, only a primary application server can send a result to the client and it can only do so after receiving a request from the client. By the algorithm of Figure 3, when the primary receives a request from the client, it can either (1. line 20 in Figure 3) send the result r to the client after going through an interaction with the backups and the database servers (if it finds out that *outcome* is initially abort) or (2. line 4 in Figure 3) send the result to the client without going through an interaction with the backups and the database servers (if *outcome* is initially commit)

Consider case 1. By the algorithm of Figure 3, the primary application server can only send the result if all database servers have accepted the result and all have acknowledged receipt of [Decide,r,commit], i.e., the primary must have received [AckDecide] from all database servers. Hence, all database servers must have committed the result.

Consider case 2. Two subcases are possible: (2.1) the primary is the the one which computed the result and assigned *outcome* to commit (in an earlier round of the while(true) loop of Figure 3), or (2.2) the primary did not compute the result but was acting as a backup at the time when the result was computed, i.e., the application server that computed the result has crashed since then. In case 2.1, after computing the result, the primary must have sent message [Decide,r,commit] to all database servers and must have received message [AckDecide] from all of them (line 18 in Figure 3) before repeating again the loop. All database servers have thus committed the result. In case 2.2, by the algorithm of Figure 4, before becoming primary, the backup must have sent message [Decide,r,commit] to all database servers and must have received message [AckDecide] from all of them (line 4 in Figure 4). All database servers have thus committed the result r. □

Lemma 6 (Agreement A.2). *No database server commits more than one result.*

PROOF. Assume that database server s_k commits a result r. By the algorithm of Figure 2, s_k must have received [Decide,r,commit] from an application server. Assume by contradiction that s_k also commits a result $r' \neq r$. By the algorithm of Figure 2, s_k must have received another message, [Decide,r',commit], from an application server. We distinguish between two cases: (1) the messages [Decide,r,commit] and [Decide,r',commit] have been sent to s_k by the same application server a_i; and (2) the messages [Decide,r,commit] and [Decide,r',commit] have been sent to s_k by different application servers a_i and a_j $(i \neq j)$.

Consider case 1, and assume that a_i sends [Decide,r,commit] before [Decide,r',commit] to s_k. If a_i has sent [Decide,r,commit] while it was acting as a backup, then by the algorithm of Figure 4, a_i can only send [Decide,r',commit] after becoming primary. This is impossible because by the algorithms of Figure 3 and Figure 4, a_i would have assigned *outcome* to commit (lines 12 and 15 in Figure 4) and would not send message [Decide,r',commit] to s_k (line 3 in Figure 3).

Consider case 2, assume $i < j$, and let t be the time at which a_i has sent [Decide,r,commit] to s_k. Whether a_i was acting as a primary or a backup at time t, by the algorithms of Figure 3 and Figure 4, and the accuracy property of the failure detector, a_i must have sent [Decide,r,commit] to a_j before time t (line 13 in Figure 3 and line 9 in Figure 4). Furthermore, before time t, a_i waits either to suspect a_j or to receive an acknowledgment from a_j that it has received [Decide,r,commit]. Either (2.1) a_i suspects a_j by time t, or (2.2) not. In case (2.1), by the accuracy property of the failure detector, a_j must have crashed and a_j cannot later send [Decide,r',commit] to s_k: a contradiction. In case (2.2), by the algorithm of Figure 4, a_j can neither send [Decide,r',commit] to s_k as a backup nor as a primary: a contradiction. □

Lemma 7 (Agreement A.3). *No two database servers decide differently for the same result.*

Proof. Assume that some database server s_l commits a result r. Assume by contradiction that some database server s_k aborts r. By the algorithm of Figure 2, s_k must have received [Decide,r,abort] from an application server a_i, whereas s_l must have received [Decide,r,commit] from an application server a_j. We distinguish two cases: either (1) $i = j$ or (2) $i \neq j$. Consider case 1. By the algorithms of Figure 3 and Figure 4, if a_i sends [Decide,r,abort] to database server s_k and [Decide,r',commit] to a database server s_l, then $r \neq r'$. In other words: s_l cannot receive message [Decide,r,commit] from a_i: a contradiction.

Consider case 2. By Lemma 2 and the algorithms of Figure 3 and Figure 4, a_i must have started sending [Decide,r,abort] to all database servers and then crashed, and then later, a_j must have suspected a_i and has sent [Decide,r,commit] to all database servers. By the algorithm of

Figure 3, a_j can only send [Decide,r,$outcome$] to s_k if $r \neq \textsf{nil}$. This can only be possible if a_j has received [Decide,r] or [Decide,r,abort] from a_i (line 3 or line 6 in Figure 4). By the algorithm of Figure 4, the only message that a_j can send to s_l is [Decide,r,abort]: a contradiction. □

Lemma 8 (Validity V.1). *If the client issues a request and delivers a result, then the result has been computed by an application server with the request as a parameter.*

PROOF. By the algorithm of Figure 1, a client does not deliver a result r until the result was received from an application server (line 4 in Figure 1). By the algorithms of Figure 3 and Figure 4, only a primary can send r to the client. Let a_i be that primary. Either a_i has itself computed r after receiving a request from the client, or a_i was backup (when the result was computed) and has received r earlier from a primary a_j $(j < i)$. In the latter case, either a_j has itself computed r after receiving the request from the client, in which case, V.1 holds, or a_j was backup and received r earlier from a primary a_k $(k < j)$. Since the number of application servers is finite, ultimately, by the algorithm of Figure 3, some application server must have computed r with the client's request as a parameter. □

Lemma 9 (Validity V.2). *No database server commits a result unless all database servers have voted yes for that result.*

PROOF. By the algorithm of Figure 2, a database server s_k can only commit a result r if it has received a message of the form [Decide,r,commit] from an application server (lines 7 and 8 in Figure 2). The database server s_k can either receive this from a primary application server or from a backup application server. By the algorithm of Figure 4, a backup can only send such a message if it has received message [Decide,r,commit] from the primary (otherwise, r is nil and the backup does not send any message). By the algorithm of Figure 3, a primary can only send a message [Decide,r] (either to a backup or to a database server), if it has received [Vote,r,yes] from all database servers; that is, if all database servers have accepted r. □

Proposition 1. *The algorithms of Figure 1, Figure 2, Figure 3 and Figure 4 solve the Trans-*

actional Exactly-Once problem.

Proof. Termination follows from Lemma 3 and Lemma 4, Agreement follows from Lemma 5, Lemma 6, and Lemma 7, and Validity follows from Lemma 8 and lemma 9. □

6 Performance

This section describes the performance of our Transactional Exactly-Once protocol (or simply *TEO*) in a practical setting. Our implementation uses off-the-shelf middleware components: Orbix 2.3 Object Request Broker [16] and Oracle 8.0.3 [17].

A client communicates with a remote application server using Orbix. The application server has a backup on a separate machine (also running Orbix) and communicates with a remote back-end database. We use Oracle for the back-end database and we have the database server run in an MC/ServiceGuard cluster of 2 machines [13].

We describe measurements from two performance tests:

1. A *latency* test that measures client response time. In this test, a single back-end database (running on a cluster) is involved. This configuration is, we believe, representative of current three-tier architectures where a single database is typically involved. We compare the performance of our protocol with the performance of two alternative protocols: (1) a baseline protocol that does not address reliability at all, and (2) a traditional 2PC protocol that guarantees only *at-most-once* semantics *(all-or-nothing)* [10].

2. A *scalability* test. This test measures the scalability of our protocol with respect to the number of databases being manipulated.

The two tests quantify the fundamental cost of providing the Transactional Exactly-Once guarantee in three-tier applications. In terms of latency, we show that our protocol introduces an overhead of 16% over a baseline unreliable protocol (that does not offer any guarantee) (Section 6.1). That overhead is actually lower that the overhead of a 2PC protocol, which we show is around

23% in our environment. This might look surprising at first glance because our protocol also ensures a non-blocking property of databases besides the *exactly-once* guarantee (2PC is blocking [10] and ensures at best *at-most-once* request delivery). However, in contrast to 2PC, our protocol does not induce any forced disk IO. The very same primary-backup scheme used to ensure client's outcome determination is also used to guarantee non-blocking. Our second test shows that this technique also makes our protocol scalable (Section 6.2). We do not introduce any additional communication overhead with respect to databases. As a consequence, the cost of our protocol is a linear function of the number of databases (or transactional files).

6.1 Testing Environment

Our implementation is built exclusively for testing purposes. Our aim was to quantify the performance in a realistic setting, not to build a complete implementation of our protocol. In particular, we consider the steady-state, failure-free performance, and we did not implement all the failure-handling and re-try logic. We assume that none of the components of the three-tier architecture fails, and we even exclude the case where the client time-outs the application server and retries its request, i.e., we exclude performance failures as well. These are the executions that are the most likely to occur in practice and for which protocols are usually optimized.

Our experiments quantify the contention-free performance of our protocol. The contention-free performance is measured on a system where one request is processed at a time. Thus, there is no contention for resources between requests. We measure the end-to-end response time as seen by the client application software. Since we conduct our experiments in a contention-free environment, our measurements do not include throughput numbers, or other multi-request metrics.

Our measurements are obtained from probes embedded in the test software. The probes collect measurements during the experiments, and communiciate these measurements to a central measurement collector after the experiment is complete. This minimizes the pertubation of the system since there is no communication of measurements during the experiment itself. We execute multiple requests for each experiment to quantify the variation in the measured response times.

We consider each response time measurement an independent observation, and compute the 90 % percent confidence interval for the average response time. We only include the average response times, but the 90 % confidence interval was less than 10 % wide in all runs. [10]

The application server object is a simple bank account with a deposit operation. The state of this bank-account object is its current balance. The deposit operation is read-write: it reads the current balance of the bank account and increases it with a certain amount. The client and application servers run as separate operating system processes on separate machines. There is only one backup server, which runs as a separate process: it executes on the same machine as the client. The client and servers execute on HP C180 PA-RISC workstations, running HP-UX 10.20. The machines are connected by a 10 Mbit/Sec. ethernet. The machines were lightly loaded during the experiments, the standard UNIX daemons ran on them, but no other significant applications were running. The network is a production ethernet, but we obtained the measurements in the late evening when it is lightly loaded.

6.2 The Latency Test

The application server communicates with the database using Oracle's implementation of SQL*net and XA. The server uses XA directly to demarcate transactions (xa_start and xa_end), and uses the Oracle Core Interface (OCI) to execute SQL statements that implement the business logic of the bank account object. The database cluster machines are K-class PA-RISC servers, running HP-UX 10.20.

We compare the performance of our protocol with those of a baseline unreliable protocol, and a standard 2PC protocol. This comparison is made with a basic three-tier application that contains a client, a server, and a database. Figure 5 depicts the communication steps of our protocol in the failure-free case. We contrast these with the communication steps of a baseline unreliable protocol. In both protocols, a client submits requests to a middle-tier application server. The application

[10]Statistically speaking, this means that there is a 90 % chance that the "real" average lies within 10 % of the measured average.

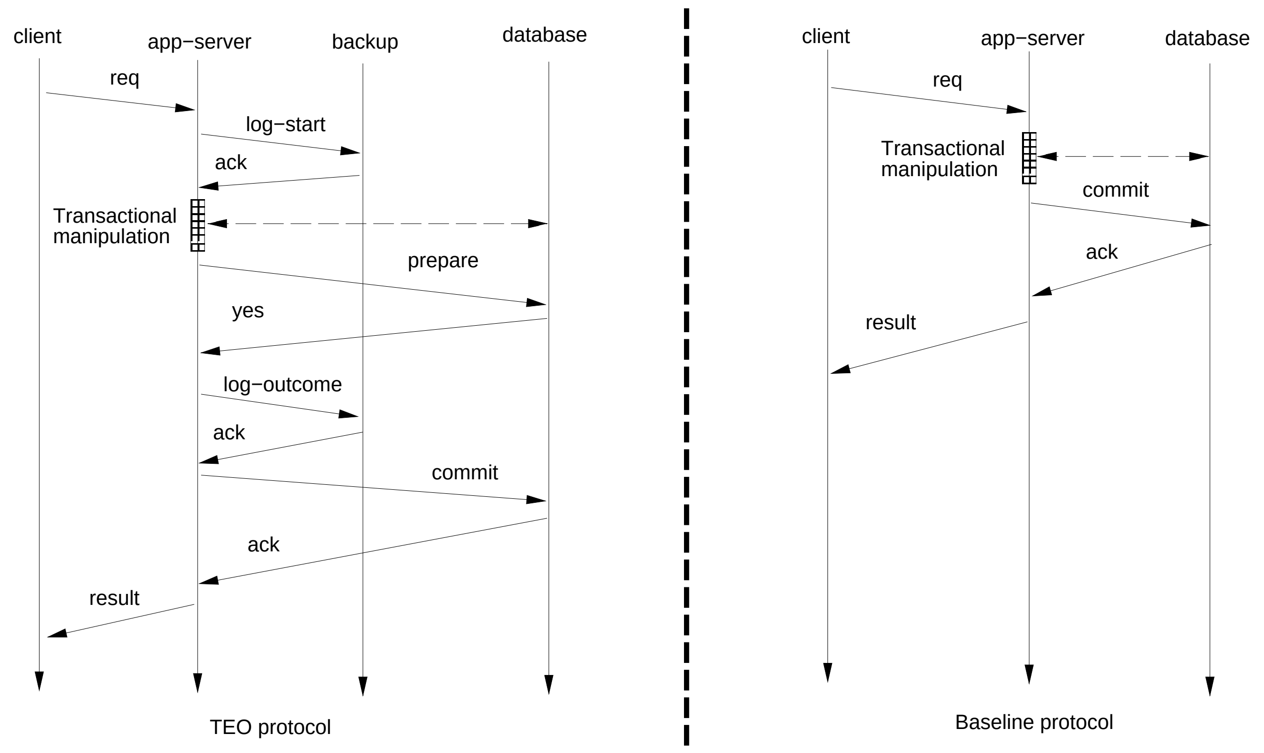

Figure 5: Communication steps in a failure-free execution

server processes a request by executing a transaction against a back-end database. The actual data manipulation by the application server is the same in both protocols: the application server starts a transaction, executes some SQL statements depending on the request type, and ends the transaction. We illustrate the transactional manipulation as a box at the application server. In the baseline protocol, the server activates this box immediately after recieving a request. After executing the transactional manipulation, the server asks the database to commit the transaction, and returns the result of the manipulation to the client. In our TEO protocol, the server performs reliable request processing. This involves storing recovery information at a backup. (In the more general TEO protocol we presented in Section 4, a set of backup servers are used.)

To implement the 2PC, we used the local disk file of the coordinator application server, which is the traditional approach taken by most transaction processing monitors. The application server

protocol	baseline	TEO	2PC
start	3.4	3.5	3.5
end	3.4	3.5	3.4
commit	18.6	18.8	17.5
prepare	0	19.0	21.2
SQL	187.0	193.2	190.6
log-start	0	4.5	12.5
log-outcome	0	4.7	12.7
other	5.0	5.1	5.1
total	217.4	252.3	266.5
cost of reliability	0%	+16%	+23%

Figure 6: Comparing the latency of the protocols

logs information about the transaction before it is started and after the outcome has been determined. Logging is a synchronous operation, the application server waits for the logging operation to complete before it continues the protocol execution.

The measurements in Figure 6, show the request processing response time for the TEO protocol, the baseline protocol, and the 2PC protocol. In addition to the client-side elapsed time, we also allocate portions of this time to specific software components that service requests. Time is measured in milliseconds. We measure the following response-time components:

- *total*: The total, end-to-end response time as seen by the client. This is the elapsed time from submitting a request to receiving a result.

- *start*: The time it takes to start a transaction. This is the elapsed time it takes to execute xa_start by the application server. This includes time spent in the XA server-side library, remote communication with the database, and executing the start operation at the database. The information communicated to the database includes unique transaction identifier (UUID). In our version of XA, these are represented as 128 byte text strings.

- *end*: The time it takes to end a transaction. This is the elapsed time to execute xa_end by the server. The server communicates an XA UUID to the database.

- *commit*: The time it takes for the application server to execute commit against the database. The server communicates an XA UUID to the database.
- *prepare*: The time it takes for the application server to execute prepare against the database. The server communicates an XA UUID to the database.
- *SQL*: The time it takes to execute the SQL statements by the server. This includes constructing the statements as datastructures and calling into the OCI library. It also includes remote communication with the database and query processing by the database.
- *log-start*: The time it takes to log start information about a transaction. This information includes a client-generated UUID, which in our system is represented as a 37 byte text string. We use these to generate the XA UUIDs.
- *log-outcome*: The time it takes to log the outcome of a transaction. The logged information includes a 37 byte UUID.

The "other" category in Figure 6 is the amount of time which is unaccounted for after allocating the response time to the listed components. Since the listed component times are all measured at the application server, the "other" category includes the communication cost of the client-server interaction. A round-trip Orbix RPC without parameters takes about 3-5 milliseconds in our environment, so the client-server communication accounts for most of the time in the "other" category.

We computed the cost of reliability, which is the percentage increase in end-to-end response time relative to the baseline protocol. When we use a backup server (i.e., in our TEO protocol), this increase is about 16%, and when logging to a local disk file (i.e., in the 2PC) the increase is about 23%. The increase is due to the prepare operation against the database and the logging operations. All the other component times are about the same.

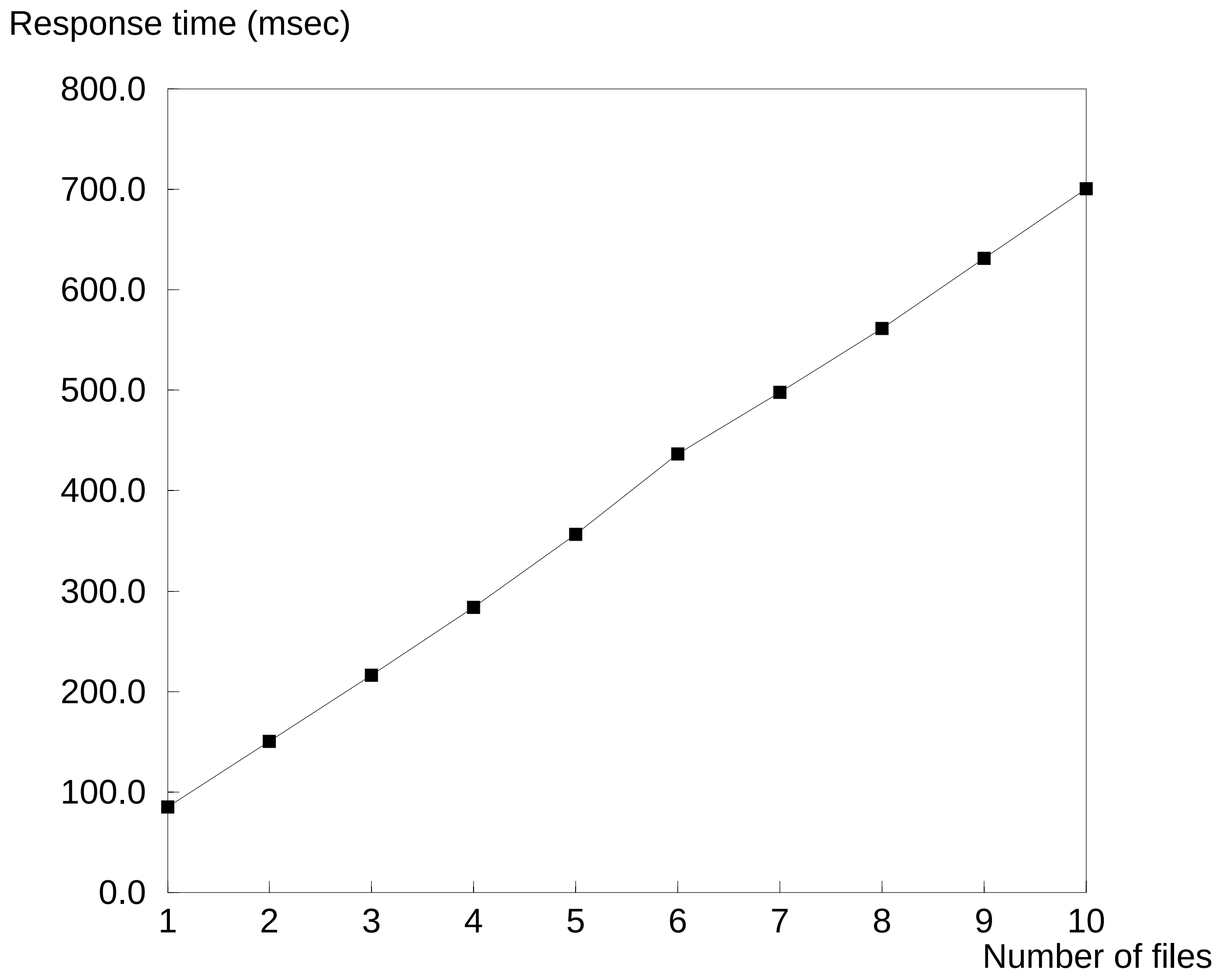

Figure 7: Response time as a function of the number of transactional files

6.3 The Scalability Test

The aim of this test was to measure the scalability of our protocol with respect to the number of databases (or transactional files) being manipulated. Since we did not have a large number of Oracle databases to perform this test, we applied our protocol to transactional files that support the XA interface.

We implemented a transactional file concept. We did not implement however concurrency control for the file access, only all-or-nothing update semantics. Essentially, starting a transaction creates a new copy of the file. Ending a transaction closes the new file, and a prepare operation flushes the new file to disk. A commit atomically replaces the old file by the new file. We implemented the files as CORBA objects, but for simplicity they all reside in the application server's address space. Thus, we do not have remote communication with the transactional files.

To quantify the scalability of our protocol, we measured the end-to-end response time with our protocol as a function of the number of transactional files being manipulated. We use a backup server to log transaction information. The client and server run on separate machines, and the backup server runs on the client machine. We use Orbix as the communication infrastructure.

We show the results of our scalability experiment in Figure 7. The server is single threaded, so the file operations are serialized. Thus, the best we could expect is a linear function of the number of transactional files, which is indeed what is depicted in the figure. As we already pointed out, this is not actually surprising because the TEO protocol does not introduce extra communication overhead with respect to the back-end tier.

7 Related Work

The Transactional Exactly-Once problem has an *agreement* flavor, and relates to the well-known atomic commitment problem as described in [10]. Transactional Exactly-Once can best be viewed as a sequence of inter-related instances of agreement problems. Furthermore, whereas atomic commitment is specified among a set of equally-weighted participants, and aims at reaching a commit or abort decision, Transactional Exactly-Once includes the processing of a result, and aims at eventually reaching a commit decision among a set of participants with different roles. For example, clients participate in a light-weight way since they do not vote, but they still need to obtain the result and outcome.

As we already mentionned, there are numerous mechanisms and solutions to deal with reliability in three-tier applications. However, we know of no protocol description that matches a correctness specification, precisely because no such specification has been proposed so far. In the following, we compare the idea underlying our protocol with the approaches behind those solutions.

7.1 Transaction Monitors

Most commonly used reliability solutions in three-tier architectures are based on transaction processing techniques. Transaction Monitors [18] or Object Transaction Services (such as OTS [2] or MTS [19]) are typically used in such a way that the middle-tier server encapsulates the processing of the request inside an atomic transaction and guarantees *at-most-once* semantics. As we pointed out earlier in the paper, nothing prevents the situation where the client does not know whether the request was indeed processed (*unknown outcome*) and the situation where the reply is lost (e.g., if the middle-tier server crashes): as a consequence, by retrying requests in an arbitrary way, the end user usually ends up with *at-lest-once* semantics.

The approach of [1] circumvents the issues of *outcome determination* and *lost reply* by encapsulating, inside the same transaction, both the processing of the request and the storage of the reply. This is achieved by including the client inside the transaction boundaries: in the specific case of [1], the client is considered a recoverable resource that participates in a 2PC protocol. There are two fundamental differences between [1] and our approach:

1. In [1], the transaction can only commit if the reply is saved in the client's stable storage. Hence, if the client crashes and recovers, it can find the reply locally. In our case, a client that crashes and recovers might need to go through the middle-tier server in order to retrieve

the reply. Whereas [1] targets applications where the reply of a request is a document or a cookie granting access to a newspaper site, we consider applications where the reply is for example a record identifying a flight ticket reservation. As a consequence, our protocol aleviates the need for having the client involved in the atomic commitment interaction and the need for assuming local stable storage at the client side, i.e., our protocol is suited for *thin* clients accessing middle-tier servers through light-weight browsers.

2. In [1], nothing prevents the situation where the transaction coordinator crashes and all participants remain blocked. In our protocol, the very same replication scheme used to ensure the high availability of the reply, is used to ensure non-blocking atomic commitment and orchestrate transaction retries.

7.2 Persistent Queues

The approach described in [8] uses *persistent queues* to ensure exactly-once request processing in client-server systems. The client submits a request to a server through a persistent *client-queue*. The server gets the request from the queue, processes it, and stores the reply into a persistent *server-queue*. The sequence < *request processing* - *reply storage* > is executed inside a transaction which resolves the issues of *outcome determination* and *lost reply*. There are two fundamental differences between our approach and [8]:

1. To ensure high-availability in [8], both the client-queue and the server-queue needs to be replicated with the additionnal cost of the mechanisms needed to maintain their consistency. Furthermore, the atomic commitment mechanism employed must be non-blocking.

2. In our approach, instead of having the client store the request in a persistent queue and have the server pull that queue, we provide a push mechanism that keeps sending the request, until the server (or some of its replicas) receive and process that request. Similarly, we do not store the reply in a persistent queue but we use the replication mechanism that ensures server availability to also make the reply highly available.

7.3 Message Logging

The recovery mechanism in [20] uses message logging to recover from failures in multi-tier architectures. The system model is that multiple, stateful clients interact with a single database server. To recover from failures, clients log requests from the outside world and requests sent to the server. The server logs incoming requests and replies sent to clients and hence guarantees that request processing is idempotent: the same request can be sent multiple times without repeating its server-side side-effect and the reply will be the same each time. In addition to requests and replies, the server also stores information about all read and write operations performed against the database, to allow the server to replay these operations during recovery of incomplete invocations.

The approach in [20] provides efficient client-side recovery against a single server, whereas our approach provides efficient server-side replication without client-side recovery. The two approaches reflect different target domains. The approach in [20] is targeted at systems where clients have inter-transaction state, such as CAD design systems. In contrast, our approach is targeted at highly interactive clients, such as web browsers, where client-side state recovery is much less important.

7.4 Object Groups

Several authors suggested the use of an *object group* abstraction to mask failures in the context of three-tier architectures [4, 5, 6, 21, 7]. The Object Management Group is in the process of standardizing a CORBA Object Group Service [22] to make CORBA applications highly-available.

Roughly speaking, a group appears to its clients to be a single, highly available entity. A group is made highly available through replication and a coordination protocol to ensure that all group members process all requests in some coordinated manner and thus contain the same state. Relying (only) on groups in a *pure* three-tier architecture (where the application state is stored in databases) to mask failures would actually imply paying the overhead of the coordination of every group of (stateless) middle-tier server replicas and building a highly available database group out of each single database. Using replication for the back-end database tier makes it complicated, if not impossible, to use standard, off-the-shelf database systems. Other issues such as how to coordinate many-to-many communication (between a group of servers and a group of databases) are not obvious and might induce considerable performance penalties [23].

In our case, we rely on off-the-shelf clustering technology to provide quick recovery for databases. However, we do not assume that each database can be viewed as a failure-free entity: we still need to handle the case of transaction aborts because of a database crash. Our primary backup replication mechanism at the middle-tier makes use of the assumption of stateless servers without the overhead of replica coordination [15].

8 Concluding Remarks

This paper defines a desirable, yet realistic, specification of end-to-end reliability in three-tier applications. We present that specification in the form of a problem called Transactional Exactly-Once which encompasses both safety and liveness properties in such environments. Transactional Exactly-Once addresses end-to-end reliability and includes all the components in a single specification, namely, clients, application servers, and database servers. The specification can be used as a metric to evaluate the correctness of reliability protocols for three-tier applications [8, 1]. It can also help building new protocols for those applications, for example by composing transactional and group communication mechanisms, along the lines suggested in [24].

Transactional Exactly-Once is meaningful because it captures a very useful exactly-once guarantee for the end-user, in all failure cases that are not related to her machine. It is also sensible because, as we show in the paper, we have devised a realistic Transactional Exactly-Once protocol.

Our protocol is different from alternative protocols that address similar issues, e.g., [8, 1, 20] in that we do not rely on any stable storage at the client side. This makes our protocol particularly well-suited for *pure* three-tier applications, where the only stable state is at the back-end databases and file systems. As we pointed out in the paper, the overhead introduced by our protocol is reasonable in a practical setting: we do not introduce extra disk accesses, and we use the very same replication scheme to ensure both the high availability of the result and the non-blocking property of atomic commitment.

The protocol we describe intimately relates three subprotocols: (1) a request retransmission protocol executed by the client; (2) a primary-backup replication protocol executed by the application servers; and (3) a distributed commit protocol. Several variations of each of these subprotocols have been discussed in the literature (e.g., [10, 25, 11]) but, to our knowledge, their integration in a practical context has never been discussed. Evaluating the feasibility of a modular solution to the Transactional-Exactly Once problem, where the three subprotocols would appear as black-boxes, and comparing the efficiency of that approach with our current solution, is the subject of our current investigation.

9 Acknowledgements

We are very grateful to Jim Pruyne and Joe Sventek for their comments on an earlier draft of this paper, to Meichun Hsu for her helpful comments about the structure of our protocol, and to Jim Gray for sharing with us his views about the meaning of reliability in three-tier architectures.

References

[1] M. C. Little and S. K. Shrivastava, "Integrating the object transaction service with the web," in *Proceedings of the Second International Workshop on Enterprise Distributed Object Computing (EDOC)*, IEEE, 1998.

[2] Object Management Group, *CORBA Services—Transaction Service*, 1.1 ed., November 1997.

[3] D. Chappell, "How microsoft transaction server changes the com programming model," *Microsoft Systems Journal*, January 1998.

[4] S. Maffeis, "Adding group communication and fault-tolerance to corba," in *Proceedings of the USENIX Conference on Object-Oriented Technologies*, June 1995.

[5] P. Felber, B. Garbinato, and R. Guerraoui, "The design of a corba group communication service," in *Proceedings of 15th Symposium on Reliable Distributed Systems*, pp. 150–159, October 1996.

[6] P. Narashimhan, L. E. Moser, and P. M. Melliar-Smith, "Exploting the internet inter-orb protocol interface to provide corba with fault tolerance," in *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, June 1997.

[7] C. Karamanolis and J. Magee, "Client-access protocols for replicated services," *IEEE Transactions on Software Engineering*, vol. 25, January 1999.

[8] P. Bernstein, M. Hsu, and B. Mann, "Implementing recoverable requests using queus," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, May 1990.

[9] x/Open Company Ltd, *Distributed Transaction Processing: The XA Specification*, 1991. XO/SNAP/91/050.

[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Reading, Mass.: Addison-Wesley, 1987.

[11] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed Systems* (S. Mullender, ed.), Addison-Wesley, 1993.

[12] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[13] P. S. Weygant, *Clusters for High-Availability: A Primer of HP-UX Solutions.* Prentice-Hall, Hewlett-Packard Professional Books., 1996.

[14] 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," in *Proceedings of FTCS'98*, June 1998.

[15] P. K. Reddy and M. Kitsuregawa, "Reducing the blocking in two-phase commit protocol employing backup sites," in *Proceedings of the Third IFCIS Conference on Cooperative Information Systems (CoopOS '98)*, IEEE, 1998.

[16] IONA Technologies Ltd, *Orbix 2.2 Programming Guide*, 1997.

[17] Oracle Corporation, *Oracle8 Application Developer's Guide.* Chapter 18, Oracle XA, Relase 8.0, A58241-01.

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

[19] D. Chappell, "The microsoft transaction server," *Distributed Computing Monitor*, June 1997. Patricia Seybold Group.

[20] D. Lomet and G. Weikum, "Efficient transparent application recovery in client-server information systems," in *Proceedings of SIGMOD'98*, 1998.

[21] A. Montresor, R. Davoli, , and O. Babaoglu, "Group-enhanced remote method invocations," Tech. Rep. UBLCS-99-5, Dept. of Computer Science, University of Bologna, April 1999.

[22] Object Management Group, *The Common Object Request Broker: architecture and specification*, revision 2.0 ed., July 1995.

[23] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni, "System support for object groups," in *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pp. 244–258, October 1997.

[24] M. C. Little and S. K. Shrivastava, "Java transactions for the internet," *Distributed Systems Engineering*, vol. 5, pp. 156–167, December 1998.

[25] B. W. Lampson, "Reliable messages and connection establishment," in *Distributed Systems* (S. Mullender, ed.), Addison-Wesley, 1993.