Implementing e-Transactions with Asynchronous Replication

Svend Frølund[1] Rachid Guerraoui[2]
[1] Hewlett-Packard Laboratories, 1501 Page Mill Rd, Palo Alto
[2] Swiss Federal Institute of Technology, CH 1015, Lausanne

Abstract

An *e-Transaction* is one that executes *exactly-once* despite failures. This paper describes a distributed protocol that implements the abstraction of *e-Transactions* in *three-tier architectures*. Three-tier architectures are typically Internet-oriented architectures, where the end-user interacts with front-end clients (e.g., browsers) that invoke middle-tier application servers (e.g., web servers) to access back-end databases. We implement the *e-Transaction* abstraction using an *asynchronous* replication scheme that preserves the three-tier nature of the architecture and introduces a very acceptable overhead with respect to unreliable solutions.

1 Introduction

Until very recently, three-tier architectures were at the leading edge of development. Only a few tools supported them, and only a small number of production-level applications implemented them. Three-tier applications are now becoming mainstream. They match the logical decomposition of applications (presentation, logic, and data) with their software and hardware structuring (PCs, workstations, and clusters). Clients are diskless (e.g., browsers), application servers are stateless, but contain the core logic of the application (e.g., web servers), and back-end databases contain the state of the applications. Basically, the client submits a request to some application server, on behalf of an end-user. The application server processes the client's request, stores the resulting state in a back-end database, and returns a result to the client. This simple interaction scheme is at the heart of the so-called *e-Business* game today.

Motivation. The partitioning of an application into several tiers provides for better modularity and scalability. However, the multiplicity of the components and their interdependencies make it harder to achieve any meaningful form of reliability. Current reliability solutions in three-tier architectures are typically *transactional* [1, 2]. They ensure *at-most-once* request processing through some form of "all-or-nothing" guarantee. The major limitation of those solutions is precisely the impossibility for the client-side software to accurately distinguish the "all" from the "nothing" scenario. If a failure occurs at the middle or back-end tier during request processing, or a timeout period expires at the client side, the end-user typically receives an exception notification. This does not convey what had actually happened, and whether the actual request was indeed performed or not.[1] In practice, end-users typically retry the transaction, with the risk of executing it several times, e.g., having

[1]The transactional guarantee ensures that if the request was indeed performed, all its effects are made durable ("all" scenario), and otherwise, all its effects are discarded ("nothing" scenario) [3].

the user charged twice. In short, current transactional technology typically ensures *at-most-once* request processing and, by retrying transactions, end-users typically obtain *at-least-once* guarantees. Ensuring *exactly-once* transaction processing is hard. Basically, some transaction outcome information should be made highly available, but it is not clear exactly *which* information should be preserved, *where* it should be stored, and for *how long*. The motivation of our work is to define and implement the abstraction of *exactly-once-Transaction (e-Transaction)* in a three-tier architecture. Intuitively, this abstraction masks (physical) transaction aborts. It adds a liveness dimension to transactional systems that also includes the client side, and frees the end-user from the burden of having to resubmit transactions.

Protocol. This paper presents a distributed protocol that implements the *e-Transaction* abstraction. We integrate a replication scheme that guarantees the *e-Transaction* liveness property with a transactional scheme that ensures the traditional safety counterpart. This integration involves the client, the application servers, and the database servers. To deal with the inherent non-determinism of the interaction with third-party databases, we make use of *write-once registers (wo-register)*. These are consensus-like abstractions that capture the nice intuition of *CD-ROMs* - they can be written once but read several times. Building on such abstractions leads to a modular protocol, and enables us to reuse existing results on the solvability of consensus in distributed systems, e.g., [4].[2] Indirectly, we contribute to better understand how the safety aspect of transactions can be practically mixed with the liveness feature of replication, and how a consensus abstraction can help achieve that mix.

Related work. Considerable work has been devoted to transaction execution on replicated data [3]. However, we know of no approach to replicate the actual "transaction processing-state" in order to ensure the fault-tolerance of the transaction itself, i.e., that it eventually commits *exactly-once*. Traditionally, it is assumed that a transaction that cannot access "enough" replicas is aborted [3], but the issue of how to reliably determine the transaction's outcome, and possibly retry it, is not addressed. In fact, addressing this issue requires a careful use of some form of non-blocking transaction processing, with some highly available *recovery information* that reflects the "transaction-processing state". In [6], the problem of *exactly-once* message delivery was addressed for communication channels. The author pointed out the importance of reliably storing some "message recovery information". In the context of *exactly-once* transaction processing, this recovery information should represent the transaction-processing state. Several approaches were proposed in the literature to store that state for recovery purposes, e.g., [7, 8, 9]. Nevertheless, those approaches do not guarantee the high-availability of that state. Furthermore, they rely on disk storage at the client or at some application server. Relying on the client's disk is problematic if the client is a Java applet that does not have the right to access the disk. Solutions based on disk storage at a specific application server would make that server host dependent, and three-tier architectures are considered scalable precisely because they prevent any form of host dependence at the middle-tier [10]. Our *e-Transaction* protocol uses the very same replication scheme, both as a highly available storage for the "transaction-processing state", and as an effective way to retry transactions behind the scenes. In contrast to most replication schemes we know about [11, 12, 13, 14], we assume stateless servers that interact with third-party databases - replication schemes have usually been designed in a client-server context: servers are stateful but do not interact with third-party entities. Another

[2] A wo-register can also be viewed as a distributed form of *software counter* [5].

characteristic of our replication scheme is its *asynchronous* nature: it tolerates unreliable failure detection and may vary, at run-time, between some form of primary-backup [12] and some form of active replication [11].

Practical considerations. Our *e-Transaction* protocol was designed with a very practical objective in mind. In particular, we assume that the functionality of a database server is given: it is a stateful, autonomous resource that runs the XA interface [15] - the X/Open standard that database vendors are supposed to comply with in distributed transaction-processing applications. We preserve the three-tier nature of the applications by not relying on any disk access at the client site, or any application server site. We do not make any assumption on the failure detection scheme used by the client-side software to detect the crash of application servers, and we tolerate failure suspicion mistakes among application servers. The overhead of our *e-Transaction* protocol is very acceptable in a practical setting where application servers are run by the Orbix 2.3 Object Request Broker [16], and database servers by the Oracle 8.0.3 database management system [17]. In terms of the latency, as viewed by a client, our protocol introduces an overhead of about 16% over a baseline protocol that does not offer any reliability guarantee.

Roadmap. The rest of the paper is organized as follows. Section 2 defines our system model. Section 3 describes the *e-Transaction* problem. Section 4 describes our protocol and the assumptions underlying its correctness. Finally, Section 5 puts our contribution in perspective through some final remarks. Appendix 1 describes the pseudo-code used to express our protocol, Appendix 2 discusses the protocol correctness, and Appendix 3 the performance of its implementation.

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.

In the following, we outline our representation of the three types of processes in a three-tier application: clients, application servers, and database servers.

Clients

Client processes are denoted by $c_1, c_2, \ldots, c_k$ ($c_i \in Client$). 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 (e.g., on behalf of an *end-user*). We say that the client *issues* a request when the operation *issue()* is invoked. 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 (e.g., to the *end-user*). A result is a value in the "Result" domain, and it represents information computed by the business logic, such as reservation number and hotel name, that must be returned to the user. In practice, the request can be a vector of values. In the case of a travel application for instance, 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.

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 each request and each result are uniquely identified. Furthermore, we assume that every result is uniquely associated with a transaction. When we say that a result is committed (resp. aborted), we actually mean that the corresponding transaction is committed (resp. aborted). For presentation simplicity we assume that a result and the corresponding transaction have the same identifier, and we simply represent such indentifiers using integers.

Application Servers

Application server processes are 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 the state of application servers, 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 is an important aspect of three-tier applications. Stateless servers do not have host affinity, which means that we can freely migrate them. Moreover, fail-over is fast because we do not have to wait for a server to recover its state. 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.

Application servers interact with the databases through transactions. For presentation simplicity, 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. In a 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. It simply returns a result. 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. 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.

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.

Database Servers

Database server processes are 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 [15] 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 identifier, and returns a vote in the domain Vote = {yes, no}. Roughly speaking, a yes vote means that the database server agrees to commit the result (i.e., the corresponding transaction). The *decide*() primitive takes two parameters: a result identifier and an outcome in the domain Outcome = {commit, abort}. The *decide*() primitive returns an outcome value such that: (a) if the input value is abort, then the returned value is also abort; and (b) if the database server has voted yes for that result, and the input value is commit, then the returned value is also commit. [3]

3 The Exactly-Once Transaction Problem

Roughly speaking, providing the *e-Transaction (exactly-once-Transaction)* abstraction comes down to ensure 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 then eventually delivered by the client. The servers might go through a sequence of aborted intermediate results until one commits and the client delivers the corresponding result. Ensuring database consistency requires that all database servers agree on the outcome of every result (abort or commit). Client-side consistency requires that only a committed result is returned to the end-user.

In the following, we state the specification of the *e-Transaction* problem. More details on the underlying intuition and the rationale behind the problem specification are given in [18]. For the sake of presentation simplicity, but without loss of generality, we consider here only *one* client, and assume that the client issues only *one* request. We assume the existence of some serializability protocol [3]. 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 *e-Transaction* 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. 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 two different results;
  (A.3) No two databases decide differently on the same result.

[3] In terms of XA, the *vote*() primitive corresponds to a *prepare()* operation while the *decide*() primitive is patterned after the *commit*() operation.

- **Validity.**

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

Termination ensures that a client does not remain indefinitely blocked (T.1). Intuitively, this property provides *at-least-once* request processing guarantee to the end-user, and frees her from the burden of having to retry requests. Termination also ensures that no database server remains blocked forever waiting for the outcome of a result (T.2), i.e., no matter what happens to the client. This *non-blocking* property is 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 [3]. Agreement ensures the consistency of the result (A.1) and the databases (A.3). It also guarantees *at most-once* request processing (A.2). The first part of Validity (V.1) excludes trivial solutions to the problem where the client *invents* a result, or delivers a result without having issued any request. The second part (V.2) conveys the classical constraint of transactional systems: no result can be committed if at least some database server "refuses" to do so. Basically, and as we point out in Section 5, the *e-Transaction* specification adds to the traditional termination properties of distributed databases, properties that bridge the gap between databases and clients on one hand, and between *at-least-once* and *exactly-once* on the other hand.

4 An Exactly-Once Transaction Protocol

Our protocol consists of several parts. One is executed at the client, one is executed at the application servers, and one at the database servers (Figure 1). The client interacts with the application servers, which themselves interact with database servers. The complete algorithms are given in Figure 2, Figure 3, Figure 4, Figure 5, and Figure 6. We describe the pseudo-code used in those algorithms in Appendix 1, and give their correctness proofs in Appendix 2.

Client Protocol

The client part of the protocol is encapsulated within the implementation of the *issue()* primitive (Figure 2). This primitive is invoked with a request as an input parameter and is supposed to eventually return a result. Basically, the client keeps retransmitting the request to the application servers, until it receives back a committed result. The client might need to go through several tries (intermediate results) before it gets a committed result. To optimize the failure-free scenario, the client does not initially send the request to all application servers unless it does not receive a result after a back-off period (line 7 in Figure 2).

Application Server Protocol

Application servers execute what we call an *asynchronous replication* protocol (Figure 5 and Figure 6). In a "nice" run, where no process crashes or is suspected to have crashed, the protocol goes as follows. There is a default primary application server that is supposed to initially receive the client's request. 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. Then the application server informs the client of the outcome of the result. The outcome might be commit or abort, according to the votes of the databases (Figure 1 (a) and (b)).

Any application server that suspects the crash of the primary becomes itself a primary and tries to terminate the result (Figure 6). If the result was already committed, the new primary finishes the commitment of that result and sends back the decision to the client (Figure 1 (c)). Otherwise, the new primary aborts the result, and informs the client about the abort decision (Figure 1 d).

Some form of synchronization is needed because (1) the result computation is non-deterministic and (2) several primaries might be performing at the same time - we do not assume reliable failure detection -. We need to ensure that the application servers agree on the outcome of every result. We factor out the synchronization complexity through a consensus abstraction, which we call *write-once registers* (or simply *wo-registers*). A wo-register has two operations: *read()* and *write()*. Roughly speaking, if several processes try to write a value in the register, only one value is written, and once it is written, no other value can be written. A process can read that value by invoking the operation *read()*. More precisely:

- *Write()* takes a parameter *input* and returns a parameter *output*. The returned parameter is either *input* - the process has indeed written its value - or some other value already written in the register.
- *Read()* returns a value written in the register or the initial value $\perp$. If a value v was written in the register, then, if a process keeps invoking the *read()* operation, then unless the process crashes, eventually the value returned is the value v.

Intuitively, the semantics of a wo-register looks very much like that of a CD-ROM. In fact, a wo-register is a simple extension of a so-called *consensus* object [19]. We simply assume here the existence of *wait-free* wo-registers [19]. It is easy to see how one could obtain a wait-free implementation of a wo-register from a consensus protocol executed among the application servers (e.g., [4]): every application server would have a copy of the register. Basically, writing a value in the wo-register comes down to *proposing* that value for the consensus protocol. To read a value, a process simply returns the decision value received from the consensus protocol, if any, and returns $\perp$ if no consensus has been triggered.

Database Server Protocol

Figure 3 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. The database server protocol has a parameter that indicates whether the protocol is called initially or during recovery. The parameter is bound to the variable *recovery*, that is then used in the body of the protocol to take special recovery actions (line 2 of Figure 3). During recovery, a database server informs the application servers about its "coming back".

Correctness Assumptions

We prove the correctness of our protocol in Appendix 2. The proofs are based on the following assumptions. We will discuss the practicality of these assumptions in Section 5.

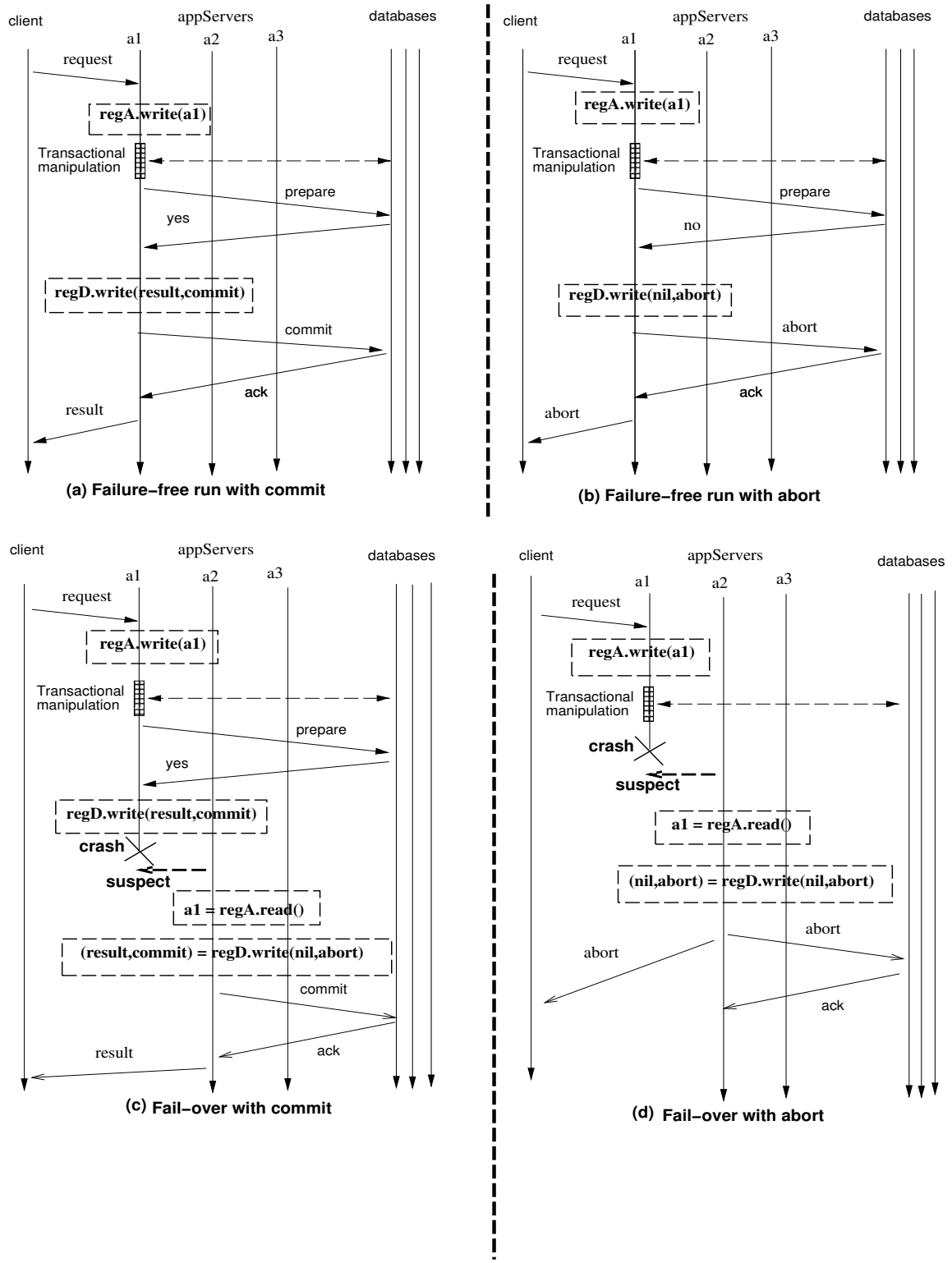

Figure 1: Communication steps in various executions

```
   Class ClientProtocol {
       list of AppServer alist := theAppServers;          /* list of all application servers */
       AppServer a_1 := thePrimary;                       /* the default primary */
       TimeOut period := thePeriod;                       /* back-off period */

       issue(Request request) {
           AppServer a_i;                                 /* an application server */
           Identifier j := 1;                             /* a result identifier */
           Decision decision;                             /* a pair (result,outcome) */

           begin
1              while true do
2                  send [Request,request, j] to a_1;
3                  timeout := period;                     /* set the timeout period */
4                  wait until (receive [Result,j, decision] from a_i) or expires(timeout);
5                      if expired(timeout) then
6                          send [Request,request, j] to alist;
7                          wait until (receive [Result,j, decision] from a_i);
8                      if (decision.outcome = commit) then
9                          return(decision.result);       /* delivers the result and exits */
10                     else j := j + 1;
           end
       }
   }
```

Figure 2: Client algorithm

```
   Class DataServer {
       list of AppServer alist := theAppServers;                    /* list of all application servers */

       main(Bool recovery) {
           Outcome outcome;                                  /* outcome of a result: commit or abort */
           AppServer a_i;                                                  /* an application server */
           Integer j;                                                        /* a result identifier */

           begin
1              if (recovery) then                          /* distinguish recovery from the initial starting case */
2                  send [Ready] to alist;                                        /* recovery notification */
3              while (true) do
4                  cobegin
5                      || wait until (receive [Prepare,j] from a_i)
6                          send [Vote,j,this.vote(j)] to a_i;
7                      || wait until (receive [Decide,j, outcome] from a_i);
8                          this.terminate(j, outcome);
9                          send [AckDecide,j] to a_i;
10                 coend
           end
       }

       terminate(Integer j, Outcome outcome) {..}                          /* commit or abort a result */

       vote(Integer j) {..}                                          /* determine a vote for a result */

   }
```

Figure 3: Database server algorithm

```
   Class AppServerProtocol {
       Client c;                                                          /* the client */
       list of AppServer alist := theAppServers;                          /* list of all application servers */
       list of DataServer dlist := theDataServers;                        /* list of all database servers */
       array of Decision WORegister regD;                                 /* array of decision WORegisters */
       array of AppServer WORegister regA;                                /* array of application server WORegisters */

       main(array of Decision WORegister r_A, AppServer WORegister r_D) {
           begin
1              regA := r_A;
2              regD := r_D;
3              while (true) do
4                  cobegin
5                      || this.compute();                                 /* computation thread */
6                      || this.clean();                                   /* cleanning thread */
7                  coend
           end
       }

       terminate(Integer j, Decision decision) {
1          begin
2              repeat
3                  send [Decide,j, decision.outcome] to dlist;
4                  wait until (for every d_k ∈ dlist:
5                                  (receive [AckDecide,j] or [Ready] from d_k));
6              until (received([AckDecide,j]) from every d_k ∈ dlist)
7              send [Result,j, decision] to c;
8          end
9      }

       prepare(Integer j) {
1          begin
2              send [Prepare,j] to dlist;
3              wait until (for every d_k ∈ dlist:
4                              (receive [Vote,j, vote_k] or [Ready] from d_k));
5              if (for every d_k ∈ dlist: (received([Vote,j,yes]) from d_k)) then
6                  return(commit);
7              else return(abort);
8          end
       }

       compute() {..}

       clean() {..}

   }
```

Figure 4: Application server algorithm

```
    AppServerProtocol::compute() {
        Request request;                                          /* request from the client */
        AppServer a_i;                                            /* an application server */
        Decision decision := (nil,abort);                         /* a pair (outcome,result) */
        Integer j;                                                /* a result identifier */

        begin
1           while (true) do
2               wait until (receive [Request,request, j] from c);
3                   if (decision.outcome = commit) then
4                       send [Result,j, decision] to c;           /* the result is already committed */
5                   else
6                       a_i := regA[j].write(this);
7                           if (a_i = this) then
8                               decision.result := this.compute(request);
9                               decision.outcome := this.prepare(j);
10                              decision := regD[j].write(decision);
11                              this.terminate(j, decision.outcome);
        end
    }
```

Figure 5: The computation thread

```
    AppServerProtocol::clean() {
        Decision decision := (nil,abort);                         /* a pair (outcome,result) */
        AppServer a_i;                                            /* an application server */
        list of Integer clist;                                    /* list of "cleaned" results */
        Integer j;                                                /* a result identifier */

        begin
1           while (true) do
2               for every a_i ∈ alist                             /* cleanning all results initiated by a_i */
3                   if (suspect(a_i)) then
4                       j := 1;
5                       while (regA[j].read() ≠⊥) do
6                           if (j ∉ clist) and (regA[j].read() = a_i) then
7                               decision := regD[j].write(nil,abort);
8                               this.terminate(j, decision);
9                               add j into clist;
10                              j := j + 1;
        end
    }
```

Figure 6: The cleanning thread

We assume that a majority of application servers are *correct*: they are always up. The failure detector among application servers is supposed to be *eventually perfect* in the sense of [4]. 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*) there is a time after which no correct application server is ever suspected by any application server. We also assume that all database servers are *good*: (1) they always recover after crashes, and eventually stop crashing, and (2) if an application server keeps computing results, a result eventually commits.[4]

We assume that clients, application servers, and database servers, are all connected through reliable channels. The guarantees provided by the reliable channel abstraction are captured by the following properties: (*termination*) if a process p_i *sends* message m to process p_j, then unless p_i or p_j crash, p_j eventually *delivers* m; (*integrity*) every process *receives* a message at most once, and only if the message was previously *broadcasts* by some process (messages are supposed to be uniquely identified).

5 Concluding Remarks

On the specification of e-Transactions. Intuitively, the *e-Transaction* abstraction is very desirable. If a client issues a request "within" an *e-Transaction*, then, unless it crashes, the request is executed *exactly-once*, and the client eventually delivers the corresponding result. If the client crashes, the request is executed *at-most-once* and the database resources are eventually released. As conveyed by our specification in Section 3, the properties underlying *e-Transactions* encompass all players in a three-tier architecture: the client, the application servers, and the databases. Not surprisingly, some of the properties are similar to those of *non-blocking transaction termination* [3]. In some sense, those properties ensure *non-blocking at-most-once*. Basically, the specification of *e-Transactions* extend them to bridge the gap between *at-most-once* and *exactly-once* semantics.

On the asynchrony of the replication scheme. The heart of our *e-Transaction* protocol is the *asynchronous* replication scheme performed among the application servers. Roughly speaking, with a "patient" client and a reliable failure detector, our replication scheme tends to be similar to a primary backup scheme [12]: there is only one active primary at a time. With an "impatient" client, or an unreliable failure detector, we may easily end up in the situation where all application servers try to concurrently commit or abort a result. In this case, like in an *active replication* scheme [11], there is no single primary and all application servers have equal rights. One of the characteristics of our replication protocol is precisely that it may vary, at run-time, between those two extreme schemes.

On the practicality of our protocol. Many of the assumptions we made are "only" needed to ensure the termination properties of our protocol (Appendix 2). These include the assumption of a majority of correct application servers, the assumption of an eventually perfect failure detector among application servers, the assumption that every database server being eventually always up,

[4]The assumption that results *eventually commit* 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 compute a result that can actually run to completion, for example a result that informs the user of the booking problem.

and the liveness properties of wo-registers and communication channels. In other words, if any of these properties is violated, the protocol might block, but would not violate any agreement nor validity property of our specification (Appendix 2). In practice, these termination-related assumptions need only hold during the processing of a request. For example, we only need to assume that, for each request, a majority of application servers remains up, and every database server will eventually stay up long enough to successfully commit the result of that request.[5] Furthermore, the assumption of a majority of correct processes is only needed to keep the protocol simple: we do not explicitly deal with application server recovery. Without the assumption of a majority of correct processes, one might still ensure termination properties by making use of underlying building blocks that explicitly handle recovery, as in [22, 23]. The assumption of reliable channels do 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 and tracking duplicates.

Finally, to simplify the presentation of our protocol, we did not consider garbage collection issues. For example, we did not address the issue of cleanning the wo-register arrays. To integrate a garbage collector task, one needs to state that the *at-most-once* guarantee is only ensured if the client does not retransmit requests after some known period of time. Being able to state this kind of guarantees would require a timed model, e.g., along the lines of [24].

On the failure detection schemes. It is important to notice that our protocol makes use of *three* failure detection schemes in our architecture, and this is actually not surprising given the nature of *three-tier* systems. (1) Among application servers, we assume a failure detector that is eventually perfect in the sense of [4]. As we pointed out, failure suspicions do however not lead to any inconsistency. (2) The application servers rely on a simple notification scheme to tell when a database server has crashed and recovered. 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. This can be implemented without requiring the database servers to know the identity of the application servers. (3) Clients use a simple timeout mechanisms to re-submit requests. This design decision reflects our expectation that clients can communicate with servers across the Internet, which basically gives rise to unpredictible failure detection.

On the practicality of our implementation Our current implementation was built using off-the-shelf technologies: the Orbix 2.3 Object Request Broker [16] and the Oracle 8.0.3 database management system [17]. Our prototype was however aimed exclusively for testing purposes. In terms of the latency, as viewed by a client, our protocol introduces an overhead of about 16% over a baseline protocol that does not offer any reliability guarantee (see Appendix 3). This overhead corresponds to the steady-state, failure and suspicion free executions. These are the executions that are the most likely to occur in practice, and for which protocols are usually optimized. Nevertheless, for a complete evaluation of the practicality of our protocol, one obviously needs to consider the actual

[5] Ensuring the recovery of every database server (within a reasonable time delay) is typically achieved by running databases in clusters of machines [20, 21]. 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, and is simply replaced with the term *always*.

response-time of the protocol in the case of various failure alternatives. This should go through the use of underlying consensus protocols that are also optimized in the case of failures and failure suspicions, e.g., [25, 23].

References

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

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

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

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

[5] J. H. Slye and E. N. Elnozahy, "Supporting nondeterministic execution in fault-tolerant systems," in *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, June 1996.

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

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

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

[9] 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.

[10] S. Frolund and R. Guerraoui, "Corba fault-tolerance: why it does not add up," in *Proceedings of the IEEE Workshop on Future Trends of Distributed Systems*, December 1999.

[11] F. B. Schneider, "Replication management using the state machine approach," in *Distributed Systems* (S. Mullender, ed.), Addison-Wesley, 1993.

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

[13] D. Powell, D. Seaton, G. Bonn, P. Verissimo, and F. Waeselynk, "The delta-4 approach to dependability in open distributed computing systems," in *International Symposium on Fault-Tolerant Computing Systems*, IEEE, June 1988.

[14] X. Défago, A. Schiper, and N. Sergent, "Semi-passive replication," in *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, October 1998.

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

[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] S. Frolund and R. Guerraoui, "Exactly-once-transactions," Tech. Rep. HPL-1999-105, Hewlett-Packard Laboratories, September 1999.

[19] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 123–149, January 1991.

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

[21] 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 the International Symposium on Fault-Tolerant Computing Systems*, June 1998.

[22] M. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," in *Proceedings of the International Workshop on Distributed Algorithms, Springer-Verlag (LNCS)*, April 1998.

[23] R. Boichat, S. Frolund, and R. Guerraoui, "Lazy consensus," Tech. Rep. EPFL-1999, Swiss Federal Institute of Technology, November 1999.

[24] C. Fetzer and F. Cristian, "The timed asynchronous model," Tech. Rep. CS97-519, UCSD, September 1997.

[25] M. Hurfin and M. Raynal, "A simple and fast asynchronous consensus protocol based on a weak failure detector," *Distributed Computing*, vol. 12, no. 4, 1999.

Appendix 1. The Pseudo-Code

We describe below the semantics the pseudo-code we use to describe our algoritms in Figure 2, Figure 3, Figure 4, Figure 5, and Figure 6.

A channel is specified by two primitives: **send** and **receive**. For example, the statement "**send** [Request,*request*] **to** p_j" captures the action of sending the message [Request,*request*] to process p_j. A message [Request,*request*] is of type "Request" and contains the value *request*. We assume that messages are uniquely identified. 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).

In many cases, servers 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 presentation, we do not describe this tagging and correlation in our protocol.

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.

Finally, we abstract the suspicion information through a predicate *suspect()*. The execution of *suspect*(a_1) by application server a_2 at t returns `true` if and only if a_2 suspects a_1 at time t.

Appendix 2. Protocol Correctness

In the following, we show that the protocol composed of the algorithms described in Figure 2, Figure 3, Figure 4, Figure 5, and Figure 6, solves the *e-Transaction* problem, as specified in Section 3. The correctness of the protocol is based on the assumptions we made in Section 4.

Termination properties

Lemma 1. *No correct primary application server remains blocked forever in one of the* **wait** *statements of Figure 4.*

PROOF (SKETCH). Assume by contradiction that some correct application server a_i remains blocked forever in one of the **wait** statements of Figure 4 (line 4 of function *terminate()* or line 3 of function *prepare()*). By the algorithm of Figure 4, this means that a_i has sent a message of type "Decide" or "Prepare" to all database servers, and a_i blocked forever waiting for a message of type "AckDecide", "Vote", or "Ready", from some database server d_k. There are two cases to consider. Either (1) d_k does not crash after a_i has sent its message, or (2) d_k has crashed after a_i has sent its message. In case (1), by the termination property of reliable channels and the algorithm of Figure 3, d_j receives a_i's message and sends back its "AckDecide" or "Vote" message. By the termination property of reliable channels, a_i eventually receives d_j's message: a contradiction. In case (2), by the assumption that all database servers are good, d_j eventually recovers, sends message "Ready" to a_i and does never crash again. By the termination property of reliable channels, a_i eventually receives d_j's message: a contradiction. □

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

PROOF (SKETCH). Let d_k be any database server that votes for a result j. By the algorithm of Figure 4 and the integrity property of communication channels, some application server a_i must have sent a "Vote" message to d_k in the context of the *prepare()* function in Figure 4. This function can only be called in the context of the *compute()* function in Figure 5. By the algorithm of Figure 5, a_i must have stored its identity in $regA[j]$. If a_i is correct, then by the algorithm of Figure 5 and the properties of wo-registers, a_i eventually calls the *terminate()* function of Figure 4, with j as an argument. By the algorithm of the *terminate()* function, a_i eventually sends a "Decide" message about j to d_k. As we assume a_i to be correct, then by the termination properties of the communication channels, d_k eventually receives the "Decide" message for result j and commits or aborts that result. Assume a_i crashes. By the assumption that a majority of application servers are correct, there is an application server a_k that keeps indefinitely executing the algorithm of Figure 6. By Lemma 1 and the properties of wo-registers, process a_k keeps forever executing the **while** loop of Figure 6. By the completeness property of the failure detector, there is a time after which a_k permanently suspects a_i. By the algorithm of Figure 6, and the properties of the wo-register, a_k eventually reads a_i in $regA[j]$ and calls the *terminate()* function of Figure 4, with j as an argument. By the algorithm of the *terminate()* function, a_k eventually sends a "Decide" message about j to d_k. As we assume a_k to be correct, then by the termination properties of the communication channels, d_k eventually receives the "Decide" message for result j and commits or aborts that result. □

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

PROOF (SKETCH). Assume by contradiction that the client issues a request, never crashes and never delivers a result. Let t_1 be the time after which all faulty application servers have crashed. By the accuracy property of the failure detector, there is a time t_2 after which no correct application server is ever suspected by any application server. By the algorithm of Figure 5 and the properties

of wo-registers, only one application server executes a given result. By Lemma T.2, there is a time t_3 after which all results executed by faulty processes are terminated. Let $t = max(t_1, t_2, t_3)$. After time t, no application server terminates a result in the context of a cleanning thread, i.e., Figure 6.

Let i and j be any two results that are initiated by the client c. Assume $i < j$. By the algorithm of Figure 2, result j is not started unless result i was terminated. Consider result i. As we assume that the client issues a request, never crashes and never delivers a result, then by the algorithm of Figure 2, the client keeps sending "Request" messages to application servers. By the assumption of a majority of correct application servers, eventually, some application server computes a result in the context of j and some database server d_k will vote for j. By Lemma 2 above, j will be terminated. Client c will thus receive a "Decide" message about j. Hence, the client keeps undefinitely initiating new results and sending "Request" messages to application servers.

As a consequence, there is a time after which all results are executed sequentially, every result is executed by a single application server and at least some application server keeps executing results. By the assumption that all database servers are good, eventually some result is committed. By the algorithm of Figure 6, the client eventually receives and delivers a result: a contradiction. □

Agreement properties

Lemma 4 (Agreement A.1). *No result is delivered by the client unless it is committed by all database servers.*

Proof (Sketch). Assume the client delivers a result. By the algorithm of Figure 2, the corresponding outcome must be commit. By the integrity property of the communication channels, the client must have received the result and the outcome from some application server as part of the *terminate()* function of Figure 4. By the algorithm of that function, this can only be done if all databases have committed the result. □

Lemma 5 (Agreement A.2). *No database server commits two different results.*

Proof (Sketch). Assume by contradiction that some database server d_k commits two different results j and j'. By the algorithm of Figure 3, and the integrity property of the communication channels, d_k must have received [Decide,j,commit] from some application server a_i and [Decide,j',commit] from some application server a'_i. These messages can only be sent as as part of the *terminate()* function of Figure 4. By the algorithms of Figure 5, and Figure 6, this can only be done if the values of $regD[j]$ and $regD[j']$ are both commit.

Assume without loss of generality that $j < j'$. By the algorithm of Figure 2, the client must have initiated result j' after j. Also by the algorithm of Figure 2, the outcome of result j, as viewed by the client, must have been abort. By the integrity property of the communication channels, some application server a_p must have sent message [Result,j,(result,abort)] to the client, as part of the *terminate()* function of Figure 4. By the algorithms of Figure 5 and Figure 6, this can only be done if the value of $regD[j]$ is abort at some time t. Hence, there is a time t at which the value of $regD[j]$ is abort and a time t at which the the value of $regD[j]$ is commit: a contradiction, given the properties of wo-registers. □

Lemma 6 (Agreement A.3). *No two database servers disagree on the outcome of a result.*

PROOF (SKETCH). Assume that some database server d_k commits a result j. Assume by contradiction that some database server d'_k aborts j. By the algorithm of Figure 3, and the integrity property of the communication channels, d_k must have received [Decide,j,abort] from some application server a_i, whereas d_k must have received [Decide,j,commit] from some application server a'_i. By the algorithms of Figure 5 and Figure 6, this can only be done if the value of $regD[j]$ is abort at some time t and commit at some time t': a contradiction, given the properties of wo-registers. □

Validity properties

Lemma 7 (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. By the algorithm of Figure 2, a client does not deliver a result r until the result was received from an application server (line 4 in Figure 2). Also by the algorithm of Figure 2, the outcome of result j, as viewed by the client, must have been commit. By the integrity property of the communication channels, some application server a_p must have sent message [Result,j,(r,commit)] to the client, as part of the *terminate()* function of Figure 4. By the algorithms of Figure 5 and Figure 6, this can only be done if the value of some $regD[j]$ is commit at some time t. By the properties of wo-registers, some application server a_i must have written commit in $regD[j]$. This can only be performed in the context of the algorithm of Figure 5, and only if a_i computes *result* out of some request received from the client. By the integrity property of the communication channels and the algorithm of Figure 2, the client must have issued that request. □

Lemma 8 (Validity V.2). *No database server commits a result unless all database servers have voted* yes *for that result.*

PROOF. Assume that some database server d_k commits a result j. By the algorithm of Figure 3, and the integrity property of the communication channels, d_k must have received message [Decide,j,commit] from some application server a_i. By the algorithms of Figure 5, and Figure 6, this can only be done if the value of $regD[j]$ is commit at some time t. By the properties of wo-registers, some application server a'_i must have written commit in $regD[j]$. This can only be performed in the context of the algorithm of Figure 5, and only if the *prepare()* function returns commit. By the algorithm of the *prepare()* function of Figure 4, application server a'_i must have received yes votes from all databases. By the integrity property of the communication channels and the algorithm of Figure 3, all database servers must have voted yes for result j. □

Appendix 3. Performance Measures

In the following, we contrast the performance of our protocol with that of alternative approaches that address similar issues.

Overview

Basically, we compare the performance of our protocol with those of a baseline protocol where no reliability is ensured (Figure 7 a), a traditional 2PC protocol that ensures at-most-once request

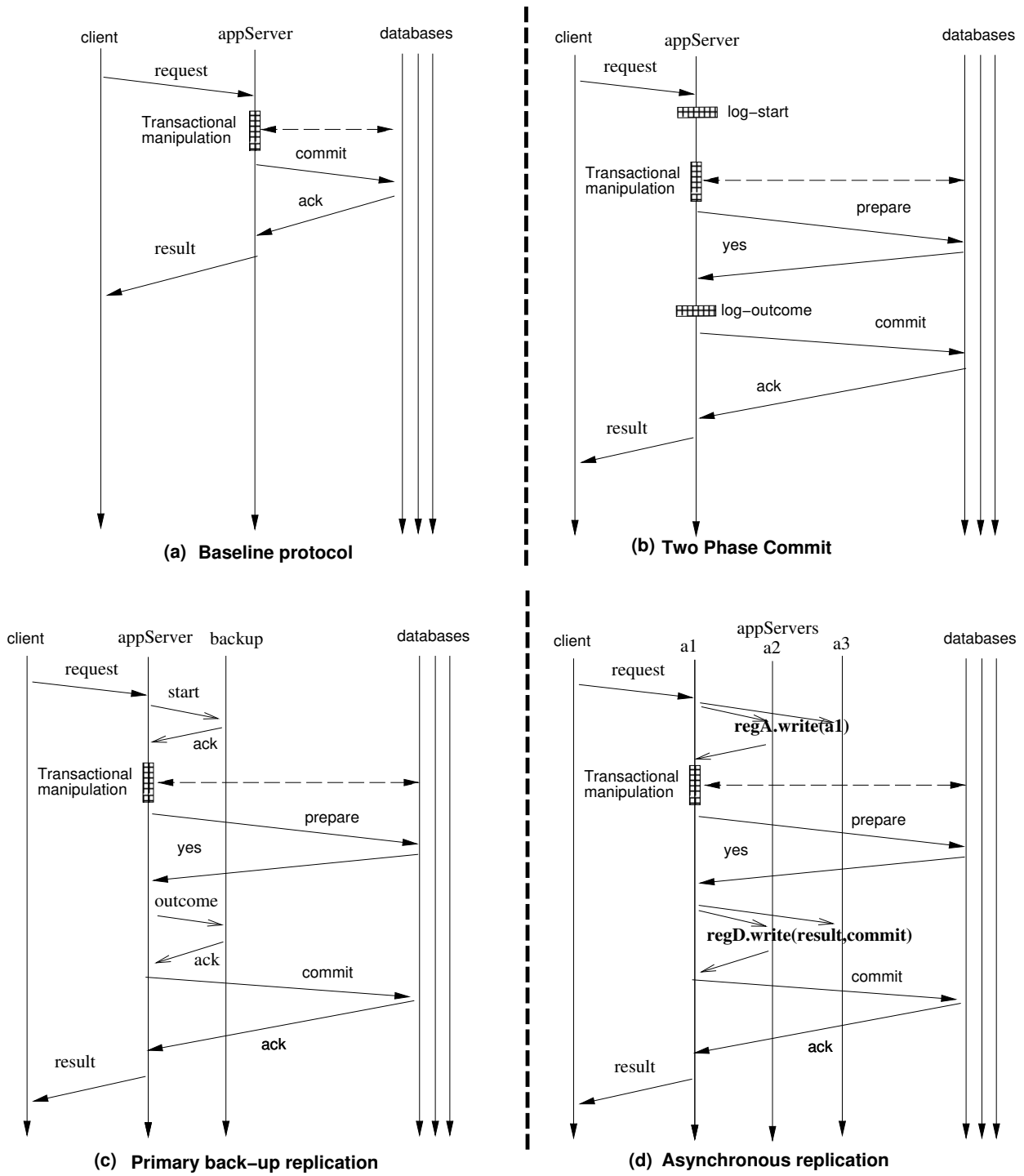

Figure 7: Communication steps in failure-free executions

processing (Figure 7 b), and a primary-backup replication scheme we adapted in [18] to implement *e-Transactions* (Figure 7 c). The primary-backup scheme requires a perfect failure detection mechanism among the application servers (a false suspicion might lead to an inconsistency).

We focus here simply on"nice" runs where no process crashes or is suspected to have crashed. In terms of latency, we show that our protocol introduces an overhead of about 16% over the baseline unreliable protocol (that does not offer any guarantee). This overhead is actually lower than 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 [3], and ensures only *at-most-once* request delivery). However, in contrast to 2PC, our protocol does not induce any forced disk IO. We use the same replication scheme to ensure client's outcome determination as we use to guarantee non-blocking.

Analytic measures

Figure 7 depicts the communication steps of the various protocols. Since our protocol requires a majority of correct application processes, we consider here the case where a single application server crash is tolerated. In that case, three application servers are required. In our primary-backup scheme, a single backup is enough.

We assume here an implementation of a *wo-register* using an optimized consensus protocol along the lines of [4]. Basically, in a "nice" run, it takes only a round trip message for the first primary to *write* into the register (the first consensus coordinator is the default primary application server).

In terms of the latency, as viewed by the client, our protocol introduces the same number of communication steps than a primary-backup scheme, but more than a 2PC protocol or an unreliable baseline protocol. The 2PC introduces however eager disk accesses.

Experimental measures

We quantify here the performance of our protocol 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].

The actual data manipulation by the application server is the same in all protocols: the application server executes some SQL statements to update a bank account on a single database, and ends the transaction. 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. production ethernet, but we obtained the measurements in the late evening when it is lightly loaded. We measured the end-to-end latency as seen by clients. For each protocol, we executed multiple identical transactions to quantify the variation in response time. We computed the 90% confidence interval for the mean response time. In all cases, the width of this interval was found to be less than 10 %. 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.

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 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.

protocol	baseline	AR	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 8: Comparing the latency of the protocols (milliseconds)

The resulting measurements are summarized in Figure 8. We measured the response time for three protocols. We did not measure the response time for the primary-backup scheme, (Figure 7 c) because the response-time components are the same as our protocol (Figure 7 a). In addition to the client-side elapsed time, we also allocated portions of this time to specific software components that service requests.

The "other" category in Figure 8 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.

The numbers in Figure 8 show that we save about 25 milliseconds by eliminating the forced-log IOs of a 2PC. In the 2PC, to maintain the log, the application server writes a start record before sending out prepare messages (this is based on a "presumed nothing" two-phase commit). When it knows the outcome, and the outcome is commit, it will write a commit record. Writing the start and commit records are eager IO operations.