A Pragmatic Implementation of e-Transactions

Svend Frølund, Rachid Guerraoui[1]
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-97
July 25th, 2000*

reliability, fault-tolerance, transaction processing, exactly-once

Three-tier applications have nice properties, which make them scalable and manageable: clients are thin and servers are stateless. However, it is challenging to implement, or even define, end-to-end reliability for such applications. Furthermore, it is especially hard to make these applications reliable without violating their nice properties.

In our previous work, we have identified e-transactions as a desirable and practical end-to-end reliability guarantee for three-tier applications. Essentially, an e-transaction guarantees that the server-side transactional side-effect happens exactly-once, and that the client receives the result of the server-side computation. Thus, e-transactions mask server and database failures relative to the client. We present in this paper a pragmatic implementation of e-transactions that maintains the nice properties of three-tier applications in the special, but very common, case of a single backend database.

[1] Swiss Federal Institute of Technology, Lausanne, Switzerland CH 1015
* Internal Accession Date Only Approved for External Publication
© Copyright Hewlett-Packard Company 2000

A Pragmatic Implementation of e-Transactions*

Svend Frølund[1] Rachid Guerraoui[2]
[1] Hewlett-Packard Laboratories, Palo Alto, CA 94304
[2] Swiss Federal Institute of Technology, Lausanne, CH 1015

Abstract

Three-tier applications have nice properties, which make them scalable and manageable: clients are thin and servers are stateless. However, it is challenging to implement, or even define, end-to-end reliability for such applications. Furthermore, it is especially hard to make these applications reliable without violating their nice properties.

In our previous work, we have identified e-transactions as a desirable and practical end-to-end reliability guarantee for three-tier applications. Essentially, an e-transaction guarantees that the server-side transactional side-effect happens exactly-once, and that the client receives the result of the server-side computation. Thus, e-transactions mask server and database failures relative to the client. We present in this paper a pragmatic implementation of e-transactions that maintains the nice properties of three-tier applications in the special, but very common, case of a single back-end database.

1 Introduction

It is very common for modern applications to follow a pure three-tier structure: thin clients, stateless application servers, and a single back-end database. The database contains all the application's state. Application servers update this state within transactions. An application server starts a transaction in response to a client request, and returns the result of executing the transaction to the client. In other words, the client is not part of the transaction, it only initiates it (indirectly).

One reason for the popularity of the pure three-tier application structure is that it lends itself well to Internet-based computing. An applet running in a browser is a typical thin client (it is thin because it does not access a disk). Because the client is not part of server-side transactions, we get transactional semantics without having transactions that span the Internet. A web server is a typical example of a stateless application server. It is stateless because it considers individual http requests to be independent. In particular, it does not store "conversation" state.

Although popular, three-tier applications are challenging from an end-to-end reliability perspective. For example, an application server may crash after committing a transaction but before sending a result to the client. When the client detects the failure, it has no way to determine if the database was updated by the transaction, and it has no way to obtain the result. We refer to these issues as *outcome determination* and *result delivery*. We distinguish between the outcome of a transaction (commit or abort) and the result of a transaction (the value computed by the server-side business logic within the transaction).

Giving a precise specification of the desired end-to-end reliability guarantee is not obvious. For example, we cannot promise the client that it will always get a result because the client itself may crash. In [FG99], we define the concept of an e-transaction as a practical and desirable end-to-end reliability guarantee for three-tier applications. The "e" in e-transactions stands for exactly-once, and reflects the fact that clients want their requests to be processed with exactly-once semantics. Moreover, with e-transactions, clients get transactional guarantees, even though they are not part of the "real" server-side transaction. Roughly speaking, if a client submits a request "within" an e-transaction (and does not crash), the client will eventually receive a reply, and this reply is the result of a server-side transaction that has committed exactly-once.

In [FG00], we present a general algorithm to im-

*This is an extended and modified version of a paper, with the same title, that appears at the 2000 IEEE Symposium on Reliable Distributed Systems.

plement e-transactions in three-tier applications with *multiple* back-end databases. In the protocol, application servers depend on each other to provide termination guarantees: if an application server crashes, other application servers terminate its transactions. Ideally, we would like to implement e-transactions with independent application servers that do not communicate with each other. The middle tier would then be completely scalable and could dynamically scale the number of application servers up and down to meet the current demand.

In this paper, we present an e-transaction protocol that allows application servers to be completely independent. The protocol provides e-transaction semantics for the very common case where application servers only manipulate a single database. Because the system only contains a single database, we can use a one-phase commit protocol. In contrast, the protocol in [FG00] uses a two-phase commit protocol, even if there is only a single database. Thus, the protocol we present in this paper is not a special case of the protocol in [FG00]: it is a fundamentally different protocol that is inherently more scalable.

The key idea behind our protocol is for the server-side transactions to leave a persistent trace in the database. This trace contains enough information to reproduce lost replies and lost outcomes after crashes of the database or the middle-tier application servers. The client can access this recovery information through the application servers. We discuss how to garbage collect the trace information, and we assess the performance penalty incurred by creating the trace.

This paper proceeds as follows. We first define what we mean by a three-tier application in Section 2. We then define e-transactions as an end-to-end reliability guarantee in this application model. We give the definition of e-transactions in Section 3. We then introduce a protocol to implement e-transactions in the single database case in Section 4. We discuss the practicality of our protocol, including its performance in Section 5. Finally, we contrast our work with related approaches in Section 6.

2 A Three-Tier Model

We consider a distributed system as a finite set of processes that communicate by message passing. Processes fail by crashing. That is, they execute their prescribed algorithm until they crash—they do not behave maliciously when they crash.

Client processes have an operation, called issue, which they invoke with a request as parameter. A request is a value in the domain Request. Clients call the issue operation to invoke the transactional logic in application servers, and thereby update the database and compute a result. A result is a value in the domain Result. When a client invokes the issue operation, we say that it *issues* a request, and if issue returns a result, we say that the client *delivers* the returned result.

A request models information provided by the end user, such as travel destination and dates. The result represents information computed by the business logic, such as reservation number and hotel name. The result must typically be returned to the user.

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.

Application server processes interact with a single database process through transactions. We introduce a primitive called compute to model the database manipulations performed by the application server business logic. The compute primitive takes two parameters, a unique identifier and a request, and returns a result value. The compute primitive starts a database transaction, and uses the identifier as transaction identifier. The request contains the input values for the SQL statements executed by the compute primitive. The result value returned by compute captures the output from executing the transactional logic. The result value must be communicated back to the client. If an error condition occurs, the compute primitive returns a special result value, nil. For example, it may be impossible to start a transaction because the database is down, or the SQL statements may fail. In fact, the compute primitive starts at most one transaction. The compute primitive is non-deterministic: it may return different results if we invoke it multiple times with the same request value.

If no error conditions occur, the compute primitive will create a database transaction that will be pending when the primitive returns. That is, the compute primitive only creates transactions, it does not terminate them. In general, the database can terminate a transaction in one of two ways: abort or commit. If a transaction commits, its changes are made permanent. That is, the manipulations performed by the compute primitive on behalf of the transaction are made durable and can no longer be undone. If the transaction aborts, its changes are discarded, and the effects of executing compute are cancelled. If the database

terminates a transaction, we say that it *decides* on that transaction. The database can only decide once for each transaction.

To allow application servers to influence the database decision, we introduce a database primitive called decide. Application servers can then ask the database to invoke this primitive (using message passing). The decide primitive takes a decision (commit or abort) and a transaction identifier. The database *may* decide on the transaction when we invoke the decide primitive. However, there is no guarantee that the database is able to make the requested decision. For example, the database may already have decided on the transaction. If the decide primitive returns yes, the database made the requested decision; if the primitive returns no, the database did not make the requested decision (at least not during this invocation of the decide primitive).[1]

3 The Exactly-Once Transaction Problem

An e-transaction is a fault-tolerant activity that takes a request, computes a result using the server-side logic, updates the database, and returns the result to the client. As the name e-transaction indicates, the end-to-end activity appears to happen only once, even if we have to retry parts of it due to failures. For example, we may have to re-execute the server-side transaction a number of times, but only one of these transactions actually commits. Moreover, the result returned to the client is the result of the transaction that actually commits.

To precisely capture the properties of an e-transaction, we have formulated the e-transaction problem. We describe this problem in [FG99]. In that paper, we consider e-transactions that update multiple databases. Here, we give a single-database incarnation of the problem.

To specify the e-transaction problem, we introduce some terminology related to our model. Since the compute primitive starts a single transaction to compute a result, we say that a result has a *corresponding* transaction—the transaction within which the result was computed. Furthermore, a given request can give rise to one or more transactions (depending on how many times we call compute for that request). We say that the database commits a transaction *for* the request if it commits one of the transactions arisen from the 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 a client issues a request, then unless it crashes, it eventually delivers a result;

  (T.2) If an application server computes a result, then the database commits or aborts the corresponding transaction.

- **Agreement**

  (A.1) No result is delivered by the client unless the database commits the corresponding transaction;

  (A.2) The database does not commit more than a single transaction for each request.

- **Validity**

  (V) 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.

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 the database always decides on a transaction, and thereby releases resources held on behalf of that transaction (T.2). Agreement ensures the consistency of the result (A.1). It also guarantees *at most-once* request processing (A.2). Validity (V) excludes trivial solutions to the problem where the client *invents* a result, or delivers a result without having issued any request.

4 The Pragmatic Protocol

The basic idea behind our protocol is to provide server-side recovery logic that clients can invoke if they

[1] The behavior of decide closely matches the semantics of the XA [x/O91] interface. This is a standard interface defined by the X/open consortium, and is supported by most commercial database systems. To simplify the presentation, we do not model the "prepare" operation in XA. Since we use a one-phase commit protocol, we do not need this functionality. Moreover, we can specify the single-database incarnation of e-transactions without reference to the prepare operation.

suspect that a failure may have happened during request processing. The recovery logic allows clients to determine the outcome of *in-doubt* transactions. Furthermore, the recovery logic allows clients to retrieve the result computed by in-doubt transactions that have committed. Under normal circumstances, clients will not invoke the recovery logic, and the performance of transaction processing will be very close to an unreliable system. However, the performance will not be *identical* to an unreliable system. Since we don't know up front which transactions will fail, or be suspected to have failed, we need to store recovery information for every transaction.

To modularize our protocol, we introduce a server-side abstraction that captures *testable* transactions. A testable transaction is one whose outcome can be reliably determined and whose result is reliably stored. In addition to these recovery facets, we can also terminate (commit and abort) a testable transaction. Moreover, the recovery functionality of a testable transaction is highly available: one server can (start to) terminate a transaction, and another server can test the outcome and result of the transaction (independently of the first server). That is, if we suspect the crash of a server, we can fail over to another server and recover the in-progress transactions at the first server under exactly-once semantics.

With testable transactions, our e-transaction protocol then has two parts: a client-server retry protocol that uses testable transactions to guarantee exactly-once semantics, and a server-database protocol that implements testable transactions. In the following, we first outline the assumptions underlying our protocol in Section 4.1 (we argue for the practicality of these assumptions in Section 5.1). We then define precisely our notion of testable transactions in Section 4.2, and outline a retry protocol that uses testable transactions in Section 4.3. Finally, in Section 4.4, we give a protocol that implements testable transactions. We prove the protocols correct in Appendix A.

4.1 Assumptions

We assume an asynchronous, distributed system model, and make the following assumptions about the components of the system and their communication:

- The database will eventually make a decision for any transaction. Moreover, the database will only commit a transaction if we invoke the `decide` primitive with a commit decision. We assume that the `decide` primitive is non-blocking: if the database invokes it, and does not crash, `decide` will eventually return.
- At least one application server is correct, that is, it does not crash. The database always recovers after a crash. At any point in time, the database is either *up* or *down*. A crash causes a transition from up to down, and a recovery causes a transition from down to up. There is a time after which the database stops crashing and remains up.
- If an application server calls `compute`, and does not crash, then `compute` will eventually return. Moreover, there is a time after which the database will be able to commit any transaction started by `compute`.
- Channels are reliable. That is, we assume that a message is eventually delivered unless either the sender or receiver crashes during the transmission. We also assume that channels do not duplicate messages.
- The client has a failure detector that is eventually perfect in the sense of [CT96]. The failure detector gives hints to the clients about server crashes. Being eventually perfect, the failure detector will eventually discover a crashed server. Moreover, it will eventually stop falsely suspecting a non-crashed server.
- If the client sends a request to start a transaction and then sends a message to determine the outcome of that transaction, the request message will not start a transaction *after* the outcome message has tested the outcome of the transaction.

4.2 Testable Transactions

We define the interface of the testable transaction abstraction in Figure 1. The interface only contains methods to terminate and recover transactions. The `commit` method attempts to commit the transaction with identifier `id`. The method also takes a result, and, if the transaction commits, the result can later be retrieved through the `get-outcome` methods. The `get-outcome` method takes the identifier for a transaction. If the transaction committed, `get-outcome` returns the result given to the `commit` method for this transaction. If the transaction aborted, `get-outcome` returns `nil`. Finally, the `rollback` method attempts to abort a transaction.

Formally speaking, a correct implementation of the `Testable-transaction` abstraction must satisfy the following properties:

```
interface Testable-transaction {
  Status commit(Result res,UUID id);
  void rollback(UUID id);
  Result get-outcome(UUID id);
}
```

Figure 1: The interface of a transaction-processing system with outcome determination

- All methods are non-blocking. That is, if an application server invokes a method, and then does not crash, the method must eventually return.
- If `commit` or `rollback` returns, then the `decide` primitive has been executed at least once.
- If `commit` returns `yes` for a transaction, then the database has committed that transaction.
- If `get-outcome` returns `nil` for a transaction, then the database has not committed that transaction.[2] If `get-outcome` returns a non-`nil` result for a transaction, then (1) the database has committed that transaction, and (2) the result returned by `get-outcome` was passed to `commit` for that transaction.

In Section 4.4, we describe an implementation that satisfies these properties.

4.3 Retry Protocol

We describe the client-server retry protocol, which relies on the `testable-transaction` abstraction to ensure safety (at-most-once). The protocol has a client and a server part. We show the client part in Figure 2 and the server part in Figure 3.

We describe the client as an object with a single method `issue`. The code in Figure 2 describes the client-side "stub" that a client application would use to submit requests to application servers. We say that the client issues a request when the `issue` method is invoked with a request. The retry protocol is round-based. The client starts a round by sending a request to a server using a `Request` message. A round is over if the client receives a `Result` message from the server. If instead the client suspects the server to have crashed, the round is completed using a termination sub-protocol. The termination protocol is embodied in the `terminate` method. This method keeps sending `Terminate` messages to servers for the incomplete

[2] A return value of nil does not mean that the transaction has aborted, the transaction may still be in progress.

```
class Client {
  Process replicas[n];
  Int i := 0;

  Result issue(Request req) {
    Result res := nil;
    while(res == nil) {
      UUID id := new UUID; UUID id1;
      send [Request,req,id] to replicas[i];
      await (receive [Result,res,id1]
        where id1 == id) or suspect(replicas[i]);
      if(received [Result,res,id1] where res != nil)
        return res;
      res := terminate(id);
      i := (i +1) mod n;
      }
    return res;
    }

  Result terminate(UUID id) {
    Int k := i; Result res; UUID id1;
    while(true) {
      send [Terminate,id] to replicas[k];
      await (receive [Outcome,res,id1]
        where id1 == id) or suspect(replicas[k]);
      if(received [Outcome,res,id1]) then
        return res;
      k := (k + 1) mod n;
      }
    }
}
```

Figure 2: The client side of the retry protocol

```
class Server {
  Testable-transaction TT;

  while(true) {
    cobegin
      await receive [Request,req,id] from client;
      res := compute(req,id);
      if(TT.commit(res,id) == yes) then
        send [Result,res,id] to client;
      else
        send [Result,nil,id] to client;
    ||
      await receive [Terminate,id] from client;
      TT.rollback(id);
      res := TT.get-outcome(id);
      send [Outcome,res,id] to client;
    coend;
    }
}
```

Figure 3: The server side of the retry protocol

request. The server-side processing of Terminate messages is idempotent, so the client is free to concurrently invoke multiple servers. Eventually a server will respond to the Terminate messages sent, and the response contains the outcome of the in-doubt request. When the client receives an Outcome message, it knows that the database has decided on the server-side transaction that may have been started for the request. Moreover, if the database decided commit, the Outcome message contains the result of the committed transaction. In summary, a round completes when the client either receives a Result message or when it receives an Outcome message. In the latter case, the client may have to start a new round if the outcome is abort (the returned result is nil).

The server-side logic in Figure 3 relies on the testable-transaction abstraction. The server instantiates an object of this type, and uses the name TT for the object. The server accepts two kinds of messages: Request and Terminate. A Request message starts a new transaction by calling compute. If compute returns successfully (the return value is non-nil), the server tries to commit the transaction. If the server successfully commits the transaction it sends the result back to the client. If not, the server sends nil back to the client. A Terminate message causes execution of the server-side termination protocol, which ensures that the transaction in question is terminated (by calling rollback). The termination protocol then sends the outcome and result of the transaction back to the client.

4.4 A Protocol for Testable Transactions

We implement testable transactions using a one-phase commit protocol. Conventional one-phase commit protocols do not provide testability. We implement testability by explicitly storing a transaction's identifier and result in the database within the transaction itself. In this way, the identifier and result will be permanent parts of the database if and only if the transaction commits. We can then use the presence of the information, and the information itself if present, to perform outcome determination and result delivery.

To formalize this idea, we introduce two primitives, insert and lookup. The insert primitive takes a transaction identifier and a result. The lookup primitive takes a transaction identifier and returns a result. They satisfy the following properties:

- insert and lookup are non-blocking. If a server invokes one of these primitives, and then does not crash, then the primitive will eventually return.

```
class Testable-transaction {

  Status commit(Result res,UUID id) {
    Status stat; UUID id1; Decision val1;
    insert(res,id);
    repeat {
      send [Decide,commit,res,id] to db;
      set timer to database-timeout;
      await (receive [AckDecide,stat,val1,id1]
        where id1 == id and val1 == commit)
          or expire(timer)
      }
    until received [AckDecide,stat,val1,id1])
    return stat;
  }

  void rollback(UUID id) {
    Status stat; UUID id1; Decision val1;
    repeat {
      send [Decide,abort,id] to db;
      set timer to database-timeout;
      await (receive [AckDecide,stat,val1,id1]
        where id1 == id) or expire(timer)
      }
    until(received [AckDecide,stat,val1,id1])
  }

  Result get-outcome(UUID id) {
    return lookup(id);
  }
}
```

Figure 4: Implementation of the testable-transaction abstraction

- lookup returns non-nil for a transaction identifier, if and only if the database has committed a transaction with that identifier.

- If lookup returns a non-nil result for a transaction identifier, then a server has invoked insert with that result for that transaction identifier.

We have implemented these primitives using SQL statements. Essentially, the insert primitive inserts the pair of a transaction identifier and result into a dedicated, user-level table. The lookup primitive then performs a query against this table. We take these primitives as given, and do not further describe their implementation.

The logic of the commit method in Figure 4 invokes the insert primitive within a transaction. Moreover, in commit, the transaction identifier passed to insert is the identifier of the transaction within which the insertion is performed.

The rollback method tries to abort a given transaction. The rollback method ensures that the decide primitive is executed at-least-once, which again guar-

```
class Database {
  Decision val; UUID id;
  while(true) {
    Status stat;
    await receive [Decide,val,id] from server;
    stat := decide(val,id);
    send [AckDecide,stat,val,id] to server;
    }
}
```

Figure 5: The database part of the protocol

antees that the database has decided on the transaction. The `get-outcome` method simply returns the result of calling the lookup primitive for a given transaction.

Application servers invoke the decide primitive in a database by sending a `Decide` message to the database. We illustrate the pseudo-code for the database in Figure 5.

5 Discussion

We discuss various pragmatic aspects of our protocol. In Section 5.1, we argue that our protocol is based on practical assumptions. We discuss ways of garbage collecting recovery information in Section 5.2. Finally, we analyze the performance of our protocol in Section 5.3.

5.1 Practicality of our Assumptions

We assume that the database eventually makes a decision for any transaction. This assumption closely matches the behavior of commercial, off-the-shelf database systems. They time out transactions so that resources are not held forever by "orphan" transactions.

To guarantee termination of the protocol—the client will eventually get a result if it does not crash—we make liveness assumptions about the database and the transaction processing. We assume (1) that the database always recovers, that (2) it eventually stops crashing, and (3) that there is a time after which any transaction can be committed. Assumption (1) is typically satisfied in practice by running the database system in a high-availability cluster. Assumption (2) reflects the fact that we cannot guarantee termination for a given request unless the database eventually stays up long enough to allow an application server to process the request to completion. However, in an asynchronous model, we cannot capture "long enough" directly in the assumption because there is no notion of time. Assumption (3) reflects the following properties of real database systems:

- For any request, an application server will eventually be able to commit a transaction before the database unilaterally decides to abort the transaction. We can achieve this in practice by adjusting the database timeout period to be significantly longer than the time it normally takes to process a request.
- For any request, the business logic inside of the compute primitive will eventually start a transaction that runs to completion. Thus, if we resubmit a transaction a number of times, we will eventually avoid concurrency-control problems, such as deadlocks, in the database.

The assumption about clients having access to a failure detector with eventual accuracy may sound strong, especially since the client may communicate with servers across wide-area networks, or even the Internet. However, we only rely on the accuracy of the failure detector for liveness. That is, failure-detection mistakes will never cause transactions to be executed twice, it may only delay the completion of transactions because non-failed transactions are aborted by the protocol. In essence, what we assume is that the client will eventually give the servers enough time to execute a request.

The client may send a request message and a number of terminate messages for the same transaction. We assume that the request message will not arrive at a server after the terminate messages have been processed. In practice, we can implement this timing assumption by delaying the processing of terminate messages on the server side. Delayed processing of terminate messages is reasonable since they represent the uncommon case with failures or failure suspicions.

5.2 Garbage Collection of Recovery Information

Our protocol stores recovery information in the database in a dedicated, user-level table. Our protocol, as described in Section 4, only creates this information, it does not delete it when it is no longer necessary. However, as we describe in the following, we can add such garbage-collection capabilities to our protocol in various ways.

In essence, the recovery information is no longer necessary when the client knows the outcome and has the result of a server-side transaction. If the client

crashes, and does not recover, we may never be able to inform the client of the outcome and result. Thus, unless we assume that clients always recover, we cannot rely on the client to inform the servers when it is safe to delete recovery information. Assuming that clients always recover is not practical with Internet-based computing where clients may become disconnected for arbitrary periods of time. Thus, to be practical, a garbage-collection algorithm has to rely on some client-independent notion of expiration time for recovery information.

A simple garbage-collection algorithm is to rely exclusively on expiration times, and never involve the client in the garbage-collection algorithm. We can then have a special garbage-collection server that issues SQL queries against the recovery information to remove information that has expired. We can even refine this basic algorithm by having clients inform this garbage-collection service when they know the outcome and result. With these client-generated "hints" the service may be able to remove some recovery information before it expires.

5.3 Performance

To quantify the performance of our protocol, we built a test system that performs transaction processing in a three-tier setting. The test system consists of a synthetic client that generates workload for a simple application server. We implement the client-server communication with the Orbix CORBA Object Request Broker from IONA [ION97]. The server instantiates a single object that contains the business logic. In the test system, the business logic updates a bank account that resides in a back-end database. The database is a standard Oracle [Ora] database management system that runs in an HP ServiceGuard cluster [Wey96].

We measure the end-to-end latency for request processing as seen by the client. We obtain the measurements under contention-free conditions—the client submits request one at a time. Moreover, we obtain the measurements under failure-free conditions—we do not inject failures into the system and measure recovery times.

We perform the experiment for three transaction-processing protocols. The first is a baseline protocol, which does not provide exactly-once semantics. It simply uses a normal one-phase commit protocol. The second protocol is our one-phase commit protocol with outcome determination. Finally, we use a traditional two-phase commit protocol where the application server itself is the transaction coordinator.

protocol	baseline	Our protocol	2PC
start	3.4	3.5	3.5
end	3.4	3.5	3.4
commit	18.6	18.8	17.5
prepare	0	0	21.2
SQL	187.0	193.2	190.6
insert	0	9.4	0
log-start	0	0	12.5
log-outcome	0	0	12.7
other	5.0	5.1	5.1
total	217.4	233.4	266.5
reliability cost	0%	+7%	+23%

Figure 7: Comparing the latency of the protocols

We illustrate the three protocols, and the resulting interaction patterns, in Figure 6.

In Figure 7, we summarize our performance measurements. In addition to the client-side, end-to-end latency, we also measure how much time is spent performing the various tasks involved in transaction processing. For example, we measure the elapsed time for the various XA operations. All measurements are elapsed time in milli seconds. The numbers in Figure 7 are averages—we also computed the 90 % confidence interval for these average numbers. The width of the 90 % confidence interval is less than 10 % in all cases.[3] The "total" time is the end-to-end elapsed time. The "start," "end," "prepare," and "commit" times are the elapsed time for the XA operations. The "SQL" category is the time spent executing the business logic, in our case updating a bank account (a mix of select and update in SQL). The "insert" category is the time it takes to insert a transaction identifier into a dedicated user-level table in the database. In the test system, a transaction identifier is 32 byte text string. In the experiment, we only insert the transaction identifier, not the result. We expect that the cost of also inserting a result will be a linear function of the size of the result value (when marshaled into a byte stream). The "log-start" and "log-outcome" are the operations performed by the transaction coordinator to store its recovery information. We use a presumed nothing protocol for two-phase commit, and store the transaction identifier before the commit protocol is begun ("log-start"). With a presumed-abort protocol, we can avoid this operation, but still need to log the outcome. The "other" category includes

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

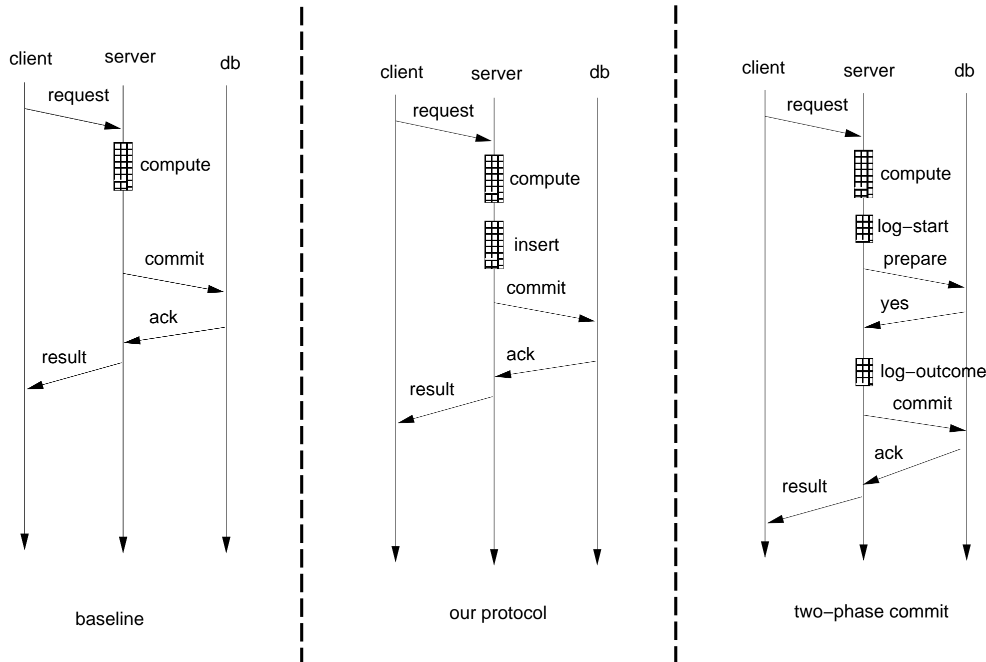

Figure 6: Communication steps in failure-free executions

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.

In summary, we can see that the cost of reliability with our protocol is less than 10 %. Moreover, we obtain exactly-once semantics without making application servers stateful or less scalable.

6 Concluding Remarks

It is challenging to implement end-to-end reliability solutions for three-tier applications. As we point out in the following, current technologies only solve part of the problem.

Typically, single-database applications use a one-phase commit protocol where the database unilaterally decides the transaction outcome. Although a conventional one-phase commit protocol guarantees all-or-nothing (atomicity), it does not guarantee that anyone but the database will know the actual decision, which makes it impossible to have middleware logic re-submit (exactly) the failed transactions. It is then left to the end-user to deal with errors fundamentally caused by failures in the back-end transaction-processing system.

With a two-phase commit protocol [GR93], a transaction coordinator computes the transaction outcome information. Compared with a one-phase commit protocol, a two-phase commit protocol involves an extra round-trip communication between the transaction coordinator and the database. It also requires the database to store a "prepare" record in its persistent transaction log, which is an eager (forced log) disk operation.

Most transaction processing monitors [BN97] employ a transaction coordinator that stores its recovery information, including transaction outcomes, in a disk file. Thus, using a transaction monitor typically makes the middle-tier stateful, violating the stateless nature of application servers. Moreover, transaction monitors usually do not provide any means for middleware retry logic to access transaction outcome information in a highly-available manner.

Besides providing highly-available access to transaction outcome information, we also need to (reliably) capture the result of transactions that actually commit. Such results cannot be re-generated without executing the same transaction twice. The approach in [LS98] is to store the result in the client's disk. The notion of resource proxy allows this result delivery to be part of the server-side transaction, but without making the client transactional. Although the client does not have to be transactional, the client still needs

access to stable storage (this may not be possible with thin clients). Moreover, the approach inherently requires a two-phase commit protocol because the result delivery is part of the two-phase commit protocol. The general technique of using message queues, as exemplified by [BHM90], is another approach to store the result in client-accessible storage (it is stored in a reply message queue). Message queues also suffers from the general issues with a server-side two-phase commit coordinator. The message queues are now transactional resources, and although we only have a single database, we have (at least) three transactional resources and need a two-phase commit protocol.

Some group-communication systems consider an open model and deals with the issue of client broadcast transparency, e.g., [BvR93, RSV94, KM99]. However, these systems only focus on message delivery in a client-server model. We consider end-to-end processing, and we assume a more general three-tier architecture where the servers not only respond to client requests; in our model, servers also invoke third party databases.

The protocol we present in this paper masks application server and database failures in three-tier applications. The key to failure masking is to transparently retry in-progress transactions without ever duplicating the side-effect of transactions. A retry mechanism requires recovery state, and we use the database itself as a container for this recovery state. Using the database allows us to use a one-phase commit protocol. In comparison, the protocol we introduced in [FG00] uses a two-phase commit protocol, even if there is only a single database, and links application servers, precisely because it does not store recovery state in the back-end database.

It may seem inelegant to store transaction recovery information in the database. However, it is important to notice that our choice is not *whether* to store this information but *where* to store it. Conventional transaction-processing systems do not solve this problem, they may at best allow the application to solve it. For example, with OTS [Obj97] the application can register an auxiliary resource with the transaction coordinator, and we can then encapsulate the transaction recovery information within this resource. However, recovery information will not be highly available (we will have to wait for server recovery to perform transaction recovery) and it will create recovery state in the middle tier, which violates the stateless nature of three-tier applications.

Acknowledgements

We thank Jerremy Holland, Fernando Pedone, Jim Pruyne, and Aad van Moorsel for feedback on earlier versions of this paper.

References

[BHM90] 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.

[BN97] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan-Kaufmann, 1997.

[BvR93] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1993.

[CT96] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[FG99] S. Frølund and R. Guerraoui. Exactly-once transactions. Technical Report HPL-1999-105, Hewlett-Packard Laboratories, September 1999.

[FG00] S. Frølund and R. Guerraoui. Implementing e-transactions with asynchronous replication. In *Proceedings of IEEE Conference on Distributed Systems and Networks (DSN)*, June 2000. An extended version of this paper is published as HP Labs Technical Report HPL-2000-46.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[ION97] IONA Technologies Ltd. *Orbix 2.2 Programming Guide*, 1997.

[KM99] C. Karamanolis and J. Magee. Client-access protocols for replicated services. *IEEE Transactions on Software Engineering*, 25(1), January 1999.

[LS98] 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.

[Obj97] Object Management Group. *CORBA Services—Transaction Service*, 1.1 edition, November 1997.

[Ora] Oracle Corporation. *Oracle8 Application Developer's Guide.*

[RSV94] L. Rodrigues, E. Siegel, and P. Veríssimo. A replication-transparent remote invocation protocol. In *Proceedings of the 13th Symposium on Reliable Distributed Systems.* IEEE, 1994.

[Wey96] P. S. Weygant. *Clusters for High-Availability: A Primer of HP-UX Solutions*. Prentice-Hall, Hewlett-Packard Professional Books., 1996.

[x/O91] x/Open Company Ltd. *Distributed Transaction Processing: The XA Specification*, 1991. XO/SNAP/91/050.

A Correctness Proofs

We first prove the end-to-end e-transaction protocol correct. The correctness proof in Section A.1 assumes the properties of testable transactions outlined in Section 4.2. Then, in Section A.2, we prove that our implementation of testable transactions satisfies those properties.

A.1 Correctness of the Retry Protocol

We prove that the protocols in Figure 2 and 3 correctly implement e-transactions.

Lemma 1 *Consider the client-side algorithm in Figure 2. The* `terminate` *method eventually returns and the client is not forever blocked in the wait statement in the* `issue` *method.*

PROOF: Consider first the wait statement inside the `issue` method in Figure 2. The client enters this statement after sending a request message to an application server a. There are two cases to consider: (1) a crashes or (2) a does not crash. If a crashes, then by the completeness property of the client's failure detector, the client will eventually suspect a and exit the wait statement. If a does not crash, then by the reliability of the communication channel, a will eventually receive the message. By the non-blocking property of compute and `commit`, a will eventually send back a result message to the client. Again by the reliability of the communication channels, the client will eventually receive this reply and exit the wait statement. Notice that the client may also (falsely) suspect a even if a does not crash. In that case, the client also exits the wait statement.

Consider the `terminate` method in Figure 2. Following the same reasoning as above, the client will not remain blocked in the wait statement inside the `terminate` method: the `rollback` and `get-outcome` methods in the server are non-blocking. Moreover, the client will not forever exit the loop after suspecting a server. By the eventual accuracy of the failure detector, there is a time after which the client will not falsely suspect a server that has not crashed. Furthermore, we assume that there is at least one correct server. Since the client keeps sending terminate messages to different servers, the client will eventually send a terminate message to a server that has not crashed or is suspected to have crashed. That server will send an outcome message that the client will eventually receive and exit the while loop. □

Lemma 2 *(T.1) If the client issues a request, and then does not crash, it eventually delivers a result.*

PROOF: Assume by contradiction that the client issues a request, remains up, and never delivers a result. Because of the non-blocking properties of Lemma 1, the client then executes the while loop in the `issue` method forever. Let t be a time after which the client does not falsely suspect application servers and all transactions started by compute can be committed. Such a time exists as a consequence of our assumptions in Section 4.1.

Let a be an application server that does not crash. The client will send a request message to a after t because it keeps sending messages to different application servers. The server a will eventually receive the request message, and compute a transaction that can be committed. It will then call `commit`, which will eventually return because it is non-blocking. Moreover, we know that decide has been executed at least once. Since the database is able to commit all transactions after t, one of these invocations will have committed the transaction started by compute. If `commit` returns yes, a sends back the result to the client, who will then deliver this result—a contradiction. If `commit` returns no, then the client will invoke send a terminate message to a. Since the transaction was committed, the `get-outcome` method will return a non-nil result, which is then passed back to the client and delivered—also a contradiction. □

Lemma 3 *(A.1) No result is delivered by the client unless the database commits the corresponding transaction.*

PROOF: Assume by contradiction that the client delivers a result, but the database has not committed the corresponding transaction. Since the client delivers the result, the client must have received a non-nil result from an application server, say a. We have to consider two cases: (1) a has sent the result as part of a result message or (2) a has sent the result as part of an outcome message.

Consider (1). A server only sends a result message with a non-nil result if `commit` returns yes, which only

happens if the corresponding transaction was committed. This is a contradiction.

Consider (2). A server only sends a non-nil result as part of an outcome message if the `get-outcome` method returns this result. By the properties of the `get-outcome` method, the server only returns a non-nil result if a server has invoked `commit` with that result and if the corresponding transaction has committed. □

Lemma 4 *(A.2) The database does not commit more than one transaction for each issued request.*

PROOF: The client sends a, possibly empty, sequence of request messages—one per iteration of the while loop in the `issue` method. We prove the following about this sequence:

1. Each request message starts at most one transaction
2. Message number $n+1$ is not sent if the database commits the transaction of message number n.

Consider (1). Each request message causes compute to be invoked at most once—the channels do not duplicate messages. Furthermore, an invocation of compute starts at most one transaction.

Consider (2). After sending request number n to a server a, the client waits to either suspect a or receive a reply from a. If the client receives a non-nil reply, it returns from the loop, and does not send request $n+1$, in which case (2) is satisfied. If the client either receives a nil result or suspects a, it calls the `terminate` method. The client only sends request $n+1$ if the `terminate` method returns nil. This method only returns nil for a given transaction identifier if the database does not commit that transaction. To see this, assume that `terminate` returns nil. Then the client has received an outcome message from some application server a'. But a' has called `rollback` before sending the outcome message. Since `rollback` has returned, we know that decide has been executed at least once. This ensures that the transaction is no longer pending in the database. Moreover, since the result contained in the outcome message was nil, a' must have gotten nil back from the `get-outcome` method. This again means that the database has not committed the transaction. Thus, the transaction is neither pending nor committed. Furthermore, the transaction will not be committed in the future because the request message will not arrive after we have execute the `terminate` method. This proves (2).

We can now use induction to prove that only the last request message sent by a client for a given request can cause a transaction for that request to commit. Moreover, since each message starts at most one transaction, we have proved the lemma. □

Proposition 1 *The algorithms in Figure 2 and 3 is a correct implementation of e-transactions.*

PROOF: Lemma 2 proves the termination property (T.1). The other termination property (T.2) follows directly from our assumption that databases eventually decide for all transactions. We prove the agreement property (A.1) in Lemma 3, and the agreement property (A.2) in Lemma 4. Finally, the validity (V) follows immediately from the structure of our algorithm: neither the server nor the client invents a result. □

A.2 Correctness of the Testable Transaction Protocol

We prove that the algorithms in Figure 4 correctly implements the properties of testable transactions.

Lemma 5 *All methods are non-blocking. If an application server invokes a method, and does not crash, the method eventually returns.*

PROOF: Consider the `commit` method in Figure 4. The insert primitive eventually returns. This is one of our assumptions about the primitive. The database eventually stops crashing. Thus, the database will eventually be able to receive and respond to one of the decide messages sent in the repeat loop of the `commit` method. When the server receives a response to one of these messages, it exits the repeat loop, and the method returns.

Consider the `rollback` method in Figure 4. This method returns when the server exits the repeat loop, which happens when the server receives an ack message for a decide message for the same transaction identifier. Assume by contradiction that the server never receives an ack message. Let t be the time after which the database is up and stops crashing. Since the server never receives an ack message, it will send a decide message after t. This message will reach the database due to channel completeness. Since the database does not crash, it will execute the decide primitive to completion (it is non-blocking). The database will send an ack message back to the server, which will receive the ack message due to completion

of the communication channels. This is a contradiction.

Consider the `get-outcome` method in Figure 4. This method is non-blocking because the lookup primitive is non-blocking. □

Lemma 6 *If* `commit` *or* `rollback` *returns, then the* decide *primitive has been executed at least once.*

PROOF: If these methods return, they have received an ack message from the database in response to a decide message. Receiving an ack message means that the database has executed the decide primitive at least once. □

Lemma 7 *If* `commit` *returns* `yes` *for a transaction, then the database has committed that transaction.*

PROOF: The `commit` method only returns yes for a transaction if it has received an ack message from the database. Moreover, it only returns yes if the ack message was sent by the database after it decided commit for the transaction. □

Lemma 8 *If* `get-outcome` *returns* nil *for a transaction, then the database has not committed that transaction.*

PROOF: The `get-outcome` method only returns nil for a transaction if lookup returns nil for that transaction. The lookup primitive only returns nil if the database has not committed a transaction with the given identifier. □

Lemma 9 *If* `get-outcome` *returns a non-*nil *result for a transaction, then (1) the database has committed that transaction, and (2) the result returned by* `get-outcome` *was passed to* `commit` *for that transaction.*

PROOF: Property (1) follows directly from the property of lookup: it returns a non-nil result for a transaction identifier if and only if the database has committed a transaction with that identifier.

Consider property (2). Assume by contradiction that lookup returns a non-nil result that was not passed to commit for the transaction. The definition of lookup guarantees that it only returns a non-nil result if a server has invoked insert with that result. But insert is only called in the `commit` method, which means that the result passed to insert was also passed to `commit`. This is a contradiction. □

Proposition 2 *The algorithms in Figure 4 and 5 correctly implement testable transactions*

PROOF: The proof follows directly from Lemma 5–9. □