Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases

Fernando Pedone, Svend Frølund
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-96
July 25th, 2000*

E-mail: {Fernando_Pedone, Svend_Frolund}@hp.com

high availability, database failover, database replication, fault tolerance, transaction processing, three-tier architectures

Enterprise applications typically store their state in databases. If a database fails, the application is unavailable while the database recovers. Database recovery is time consuming because it involves replaying the persistent transaction log. To isolate end-users from database failures, we introduce Pronto, a protocol to orchestrate the transaction processing by multiple, standard databases so that they collectively implement the illusion of a single, highly-available database. The key challenge in implementing this illusion is to enable fast failover from one database to another so that database failures do not interrupt the transaction processing. We solve this problem with a novel replication protocol that handles non-determinism without relying on perfect failure detection.

* Internal Accession Date Only Approved for External Publication

1 Introduction

High availability is essential for mission-critical computing systems. This is especially true for internet-based e-commerce applications: if the application is unavailable, the business is closed. Such applications commonly follow a three-tier structure, where front-end web browsers send http requests to middle-tier web servers who perform transactions against a back-end database. As for most online transaction processing systems, the database is the availability bottleneck in three-tier applications. This is not surprising because the database usually contains the entire application state. Database failures normally result in log based recovery: to restart, the database has to establish a consistent pre-failure state from a persistent transaction log. In contrast, web servers are typically stateless, and if a web server fails, browsers can failover to another web server and immediately continue their processing. Rebinding to another web server is usually orders of magnitude faster than recovering against a transaction log.

To construct applications that are continuously available, we have to reduce the impact of database failures on the overall system downtime. At the same time, we want to use standard, off-the-shelf database systems for the back-end transaction processing. Since standard database systems use log-based recovery, we want to use multiple databases in the back-end to somehow "mask" the log-based recovery of one database and thus be able to continue processing transactions against other databases.

Our goal is to build a highly-available transaction-processing system out of standard, and possibly heterogeneous, databases. Moreover, we want the multiplicity of the databases to be transparent to clients of the transaction-processing system. The database ensemble should appear as if it were a single, highly-available database. We formally define what it means for the ensemble to be highly available. Intuitively, it means that users of the ensemble never wait for log-based recovery, even if individual databases in the ensemble fail.

We present a protocol to orchestrate the execution of a database ensemble, providing the illusion of a single, highly-available database system. Orchestrating the execution of standard database systems in a highly-available ensemble is not straightforward. To illustrate why, let us consider using conventional replication techniques, such as active replication and primary-backup, to this problem. With active replication, we would keep the ensemble synchronized through some broadcasting protocol that delivers messages in the same order to all databases. The problem with this approach is that databases are inhererently non-deterministic in the way they process transactions. Even if we deliver the same messages in the same order at all databases, their states may diverge due to their individual non-determinism. In principle, one could build a deterministic execution by submitting transaction requests sequentially to the database, but such a solution would not have good performance. With primary-backup, a single

database would be the primary. The primary processes transactions, and checkpoints its state to one or more backups. Since only the primary actually executes transactions, we can deal with non-determinism. However, making a standard database checkpoint its state against other databases introduces a number of issues. The checkpointing mechanism typically supported by standard databases is to ship the transaction log to the backups. However, log shipping is either asynchronous, and we may lose transactions in a failover to a backup, or based on atomic commit (to preserve consistency), which may block transactions if the primary fails.

The main idea behind our protocol is to use a hybrid approach that has elements of both active replication and primary-backup. Essentially, we deal with database non-determinism by having a single (primary) database execute transactions in a non-deterministic manner. But rather than checkpoint the resulting state to backups, we send the transaction itself to the backups along with ordering information that allows the backups to make the same non-deterministic choices as the primary. Like active replication, every database processes all transactions. Unlike traditional active replication, the backups process transactions after the primary, which allows the primary to make non-deterministic choices and export those choices to the backups. By shipping transactions instead of transaction logs, we can have heterogeneous databases with different log formats, and prevent the contamination that may result if the data in one database becomes corrupt. Briefly, our protocol can cope with the existence of several primaries, a situation that can happen in certain transitory conditions, without leading to database inconsistencies. Furthermore, our protocol does not block nor lose transactions if the primary or the backups crash.

This paper is structured as follows. Section 2 describes the system model, some abstractions used in the paper and formally states the problem we are concerned about. Section 3 presents the Pronto protocol in detail and sketches its proof of correctness. Section 4 discusses some aspects about the impact of the Pronto protocol on the transaction response time. Section 5 reviews some related works on commercial databases and research projects, and Section 6 concludes the paper.

2 Model, Definitions and Problem

2.1 Processes, Communication and Failures

We assume an asynchronous system composed of two disjoint sets of processes: a set $C = \{c_1, c_2, ...\}$ of database client processes (i.e., middle-tier web servers), and a set $S = \{s_1, s_2, ..., s_n\}$ of database server processes (i.e., back-end databases). Every database server has a copy of all data items. Processes can only fail by crashing (i.e., we do not consider Byzantine failures).

We further assume that database clients and servers have access to failure detectors [4], used to monitor database server processes. The class of failure detectors we consider guarantees that every database server process that crashes is eventually suspected to have crashed *(completeness)*, and there is a time after which some database server process that does not crash is never suspected *(accuracy)* [4].

Processes are all connected through reliable channels. We 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. Reliable channels are defined by the primitives *send*(m) and *receive*(m). Database server processes can also exchange messages using a broadcast abstraction built on top of reliable channels (also known as Atomic Broadcast or Total Order Broadcast). Broadcast communication is defined by the primitives *broadcast*(m) and *deliver*(m), and guarantees that if a database server process delivers a message m, then all database server processes that do not crash eventually deliver m *(agreement)*, and if two database servers, s_i and s_j, both deliver messages m_1 and m_2, then s_i delivers m_1 before m_2 if and only if s_j delivers m_1 before m_2 *(total order)*. We discuss in Section 4 implementations for the broadcast primitives.

2.2 Databases and Transactions

In this section, we precisely define the notions of databases and transactions. Databases are specified by a set of primitives and their associated behavior. The syntax and semantics of our database primitives are those of standard off-the-shelf relational database systems. We define transactions as sequences of database requests.

A database is an abstraction implemented by database server processes. The database abstraction is defined by the primitives presented next.

- `begin`(t_a) and `response`$(t_a, -)$. The `begin` primitive starts a transaction t_a in the database. It has to precede any other operations requested on behalf of t_a. The database issues a response once it is ready to process t_a's requests.

- `exec`$(t_a,$ *`sql-st`*$)$ and `response`$(t_a, -)$. The `exec` primitive requests the execution of an SQL statement to the database. *`sql-st`* can be any SQL statement, except for commit and abort (e.g., select or update). When executing *`sql-st`*, the database may decide to abort t_a, if t_a is involved in a deadlock, for example.

- `commit`(t_a) and `response`$(t_a, -)$. The `commit` primitive terminates a sequence of `exec`'s for transaction t_a and requests t_a's commit. The response returns a confirmation that t_a

has been committed. If t_a cannot be committed in response to a commit request (e.g., no disk space), we assume that the database server process crashes.[1]

- `abort`(t_a). The `abort` primitive terminates a sequence of requests for transaction t_a and requests t_a's abort. We assume that an abort request can always be executed, and so, there is no need for a confirmation response from the server.

We define a transaction t_a as a finite sequence $\langle$ `begin`(t_a); `response`$(t_a, -)$; `exec`$(t_a, -)$; `response`$(t_a, -)$; ... ; [`commit`(t_a); `response`$(t_a, -)$ / `abort`(t_a)] $\rangle$, where `commit`(t_a); `response`$(t_a, -)$ and `abort`(t_a) are mutually exclusive, although one must take place. Our definition of a transaction differs from traditional definitions (e.g., [2]) in that we focus on the requests submitted to the database and not on the operations performed by the database on the data items. For example, in some cases, even if a commit is requested for some transaction t_a, the database may decide to abort t_a.

A database server process s_i guarantees the following properties:

DB-1 Transactions are serialized by s_i using strict two-phase locking (strict 2PL).

DB-2 If s_i receives an infinite number of transactions to execute and does not crash, then s_i commits an infinite number of transactions.

Property DB-1 ensures *serializability*, that is, any concurrent transaction execution $\mathcal{E}$ has the same effect on the database as some serial execution $\mathcal{E}_s$ of the same transactions in $\mathcal{E}$ [2]. Furthermore, strict 2PL schedulers have an interesting property that will be exploited by our protocol: if transaction t_a has been committed before transaction t_b by s_i in some execution $\mathcal{E}$, then in any serial execution $\mathcal{E}_s$ equivalent to $\mathcal{E}$, t_a precedes t_b (notice that a serial execution establishes a total order between transactions).

Property DB-2, although not explictly stated as such, is often assumed to be satisfied by current database systems. DB-2 reflects the fact that in general, databases do not guarantee that a submitted transaction will commit (e.g., the transaction may get involved in a deadlock and have to be aborted) but the chances that a database aborts all transactions that it processes is very low.

2.3 Clients and Transactional-Jobs

The transactional-job abstraction models the business logic that runs in middle-tier web servers. The execution of a job generates a transaction that will be executed by database server processes.

[1] This is not as restrictive as it may seem because if a client requests a commit, all transaction operations have been executed successfully (e.g., the transaction is not involved in a deadlock).

A job j terminates sucessfully if the transaction it generates requests a commit and is committed, or requests an abort.

Database client processes use a *transactional-job* abstraction to generate database requests. The execution of a transactional job (or simply job, for short) is defined by the primitives `submit`(j) and `response`(*result*). The `submit`(j) primitive requests the execution of a job, and the `response`(*result*) primitive returns the results of the job.

The transactional-job abstraction allows clients to access a replicated database as if they were accessing a highly-available single-copy database. It satisfies the following properties.

RDB-1 If $\mathcal{E}$ is an execution of the replicated database system, then there exists some serial execution $\mathcal{E}_s$ that is equivalent to $\mathcal{E}$.

RDB-2 If a client submits a transactional job j and does not crash, then it will eventually issue a response with the results of j.

RDB-1 is the serializability property in the context of replicated databases (also called one-copy-serializability). RDB-2 states that the submission of a job should always result in its execution.

2.4 Pragmatic Issues About the Model

Our database model captures the behavior of standard, off-the-shelf database systems. Since our model is based on the notion of SQL statements, and not read and write operations, it closely matches the semantics of standard database interfaces, such as JDBC and ODBC [16]. Furthermore, we allow the database to exhibit non-deterministic behavior in how it chooses to order transactions.

Another aspect of the database model is the interface used by clients to access the database. We want the client access method to capture a standard database interface as well. This will allow us to implement a replicated database ensemble and give clients the illusion of accessing a single, highly-available database system. In our model, clients access the database by submitting entire transactional jobs. A transactional job models a piece of business logic that contains a number of SQL statements. Using jobs instead of SQL statements at the client side interface allows us to provide transparency and mask failures relative to the client. If the user instead submits SQL statements, and a failure occurs, we cannot simply re-submit the same statements: the business logic that submits the statements may select subsequent statements based on the non-deterministic results returned by previous statements. The notion of transactional job is supported by the JDBC interface in the form of *prepare statements* [16].

A consequence of modeling the client-side behavior in terms of transactional jobs is that we assume that the business logic does not receive input from users or other entities during its execution. This assumption is satisfied by most interactive systems, and is certainly true for three-tier applications where the business logic resides in the middle-tier. Using transactional jobs to capture the client access method does not mean that our algorithm cannot be used if the client submits individual SQL statements through a JDBC interface. It means that we only can provide transparency if the client allows us to re-execute the business logic. If the client submits individual SQL statements, it must be prepared to re-submit these statements if a failure occurs.

3 The Pronto Failover Protocol

3.1 The Protocol at a Glance

The protocol is based on the primary-backup replication model, where one database server, the *primary*, is assigned a special role. The primary is the only server supposed to interact with the clients, who submit transaction requests resulting from the execution of a transactional job. The other database servers (the *backups*) interact only with the primary.

Failure/Suspicion Free Execution. To execute a transactional job j, a client c takes the first transaction request originated from the execution of j (i.e., begin transaction) and sends this request to the database server s_i that c believes to be, most likely, the current primary. After sending the begin transaction request, c waits for the result or suspects s_i to have crashed. The execution proceeds as follows.

(a) If s_i is the current primary, it executes the request and sends the result to c. In this case, c continues the execution of j, by submitting other transaction requests to the primary on behalf of j. If the primary does not crash and is not suspected by c, the execution proceeds until c requests the transaction termination (see below).

(b) If s_i is not the current primary, it returns an error message to c, which will choose another database server s_j and send to s_j the transaction request.

(c) The case where c suspects s_i to have crashed is treated in the next paragraph.

If the primary does not crash and is not suspected, the client eventually issues a request to terminate the transaction (i.e., commit or abort). The primary executes a commit request from the client by broadcasting the transaction unique identification, the SQL statements associated with the transaction, and some control information to all backups.

Upon delivering a committing transaction, each database server decides to commit or abort the transaction, and sends the transaction's outcome to the client. The protocol ensures that all database servers reach the same result (i.e., commit or abort) individually, after executing the validation test. If a database server decides to commit the transaction, it executes the SQL statements associated with the transaction against the local database, making sure that if two transactions t_1 and t_2 have to be committed, and t_1 is delivered before t_2, t_1's SQL statements are executed before t_2's SQL statements.

As a response to the commit request, a client will eventually receive the transaction's outcome from the primary and/or from the backup servers. If the outcome is abort, the client re-executes job j. Otherwise the client issues a response for job j. Figure 1 shows the protocol in an execution without failures and failure suspicions. In such a case, the protocol comes down to a typical primary-backup replication model.

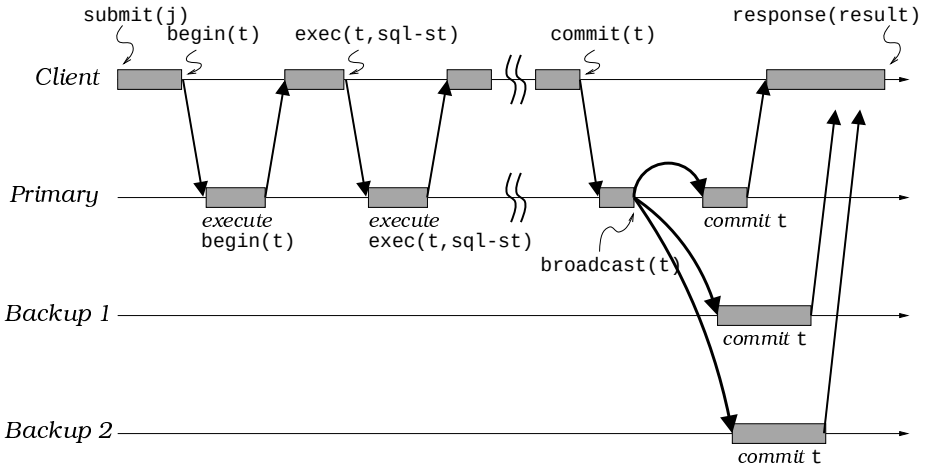

Figure 1: Pronto without crashes and suspicions

Failure/Suspicion Handling. After submitting a request, a client c may suspect the primary s_i to have crashed. This may happen because s_i has crashed or because c incorrectly suspects s_i. In any case, c sends an abort transaction message to s_i (just in case it was a false suspicion), chooses another database server s_j, and re-executes j (by sending the begin transaction request) using s_j.

If the primary crashes or is suspected by another database server, the execution evolves as a sequence of epochs. During an epoch, there can only exist one primary, which is deterministically computed from the epoch number. The epoch change mechanism works as follows. When a backup suspects the primary to have crashed, it broadcasts a message to all database servers

to change the current epoch (which will result in another database server as the primary). A backup may suspect the current primary incorrectly. In such a case, the primary also delivers the change epoch message, and will abort all transactions in execution and inform the application servers corresponding to these transactions that a new epoch has started with a new primary. Clients have to re-start the execution of their jobs in the new primary, as described before.

Due to the unreliable nature of the failure detection mechanism used by database servers and the time it takes for messages to reach their destinations, it is possible that at a given time during the execution, database servers disagree on the current epoch, and so, multiple primaries may actually be able to process transactions simultaneously.

To prevent database inconsistencies (i.e., non-serializable executions) that may arise from transactions executing concurrently on different primaries, a transaction passes a validation test before committing. In order to do that, every transaction is broadcast together with the epoch when it executed. The validation test ensures that a transaction is only committed by some database server if the epoch when the database server delivers the transaction and the epoch when the transaction was executed are the same.

Figure 2 depicts a scenario where two primaries try to commit transactions t and t'. When Primary 1 broadcasts transaction t, it has not delivered the new epoch message. Transaction t' is committed by every database server after it is delivered since t' executed in the same epoch as it is delivered. When transaction t is delivered by some database server, t is aborted since t did not execute in the epoch when it is delivered. Although not shown in Figure 2, after validating a transaction, each database server sends the transaction outcome to the client.

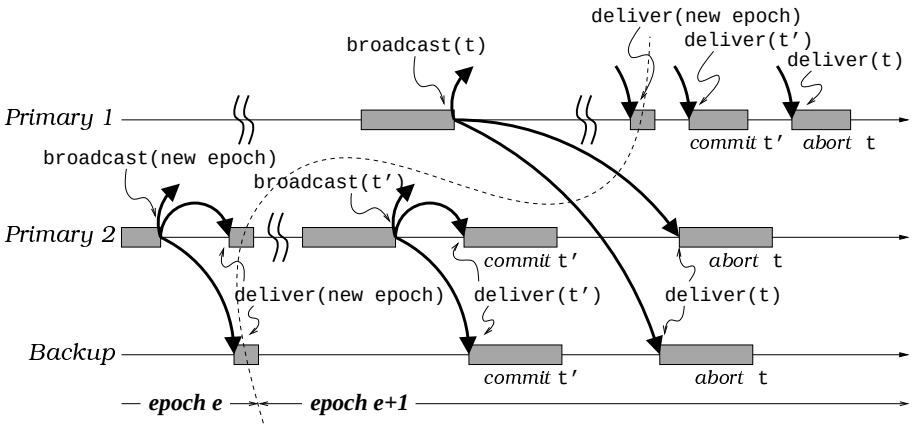

Figure 2: Two primaries scenario

According to the protocol, a transaction passes through some well-defined states. A transaction starts in the executing state and remains in this state until a commit or abort is requested.

If the client requests to commit the transaction, the transaction passes to the committing state and is broadcast to all database servers by the primary. A transaction delivered by a database server is in the committing state, and it remains in the committing state until its fate is known by the database server (i.e., commit or abort). The executing and committing states are transitory states, whereas the committed and aborted states are final states.

3.2 The Detailed Algorithm

Algorithm 1 is the client side of the Pronto protocol. The algorithm is divided into two parts. In the first part, the client locates the primary server, and in the second part the client executes the transactional job. A transactional job is modeled as a function that receives the results from the previous request to generate the next request. The first request generated by a job j_a is $\texttt{begin}(t_a)$ (i.e., $j_a(\perp) = \texttt{begin}(t_a)$).

- **Locating the primary server (lines 2-14).** Initially, the client chooses one database server (lines 7-8), tries to start the transaction on this server (line 9), and waits for an answer or suspects the server to have crashed (line 10). If the client suspects the server to have crashed (line 11), the client chooses another server and, as before, tries to start the transaction in this server. Notice that client c suspects database server process s_i if $s_i \in \mathcal{D}_c$.

- **Executing a transactional job (lines 15-25).** The client sends the transaction requests, resulting from the execution of the transactional job, to the primary. After sending a request (line 17), the client waits until it receives an answer or it suspects the primary to have crashed (line 18). If the primary decides to abort the transaction, the client re-executes the job using the same database server. A suspicion leads the client to loop back to the first part of the algorithm, and re-execute the job on a new primary, if the previous has crashed.

In both parts of the algorithm, when a client suspects a primary p_c (lines 11 and 19), before trying to execute the transaction in another server, the client sends an abort transaction message to p_c (lines 12 and 20). This is done because the client may falsely suspect the primary, and in this case, before starting a new transaction, the client terminates the previous one.

Algorithm 2 (pages 12 and 13) is the server side of the Pronto protocol. Except for lines 15-19, the algorithm is executed as a single task with several entry points (lines 5, 28, 44, and 46).

- **Executing transactions (lines 5-26).** This is the primary's main procedure. If a client sends a $\texttt{begin}(t_a)$ request to a backup, the backup refuses to process the request (lines

Algorithm 1 Database client c

```
1:  To execute submit(j_a)...

2:    p_c ← 0
3:    findPrimary ← true
4:    repeat
5:      repeat
6:        request ← j_a(⊥)
7:        if findPrimary = true then
8:          p_c ← (p_c mod n) + 1                          {choose some server and...}
9:        send (t_a, request) to p_c                       {...try to begin t_a on this server}
10:       wait until (receive (t_a, result) from p_c) or (p_c ∈ D_c)   {wait response or suspect server}
11:       if not(received (t_a, result) from p_c) then     {if suspect the server...}
12:         send (t_a, abort(t_a)) to p_c                  {...abort t_a (in case of false suspicion)}
13:     until (received (t_a, result) from p_c) and (result ≠ "I'M NOT PRIMARY")
14:     findPrimary ← false

15:     repeat
16:       request ← j_a(result)
17:       send (t_a, request) to p_c                       {send request to primary, and...}
18:       wait until (receive (t_a, result) or (p_c ∈ D_c)   {...wait result or suspect primary}
19:       if not(received (t_a, result) from p_c) then     {if suspect primary...}
20:         send (t_a, abort(t_a)) to p_c                  {...abort t_a}
21:         result ← ABORT
22:         findPrimary ← true
23:     until (result = COMMIT) or (result = ABORT)
24:   until (request = ABORT) or (request = COMMIT and result ≠ ABORT)
25:   response(result)
```

Algorithm 2 Database server s_i

1: **Initialization...**

2: $e_i \leftarrow 1$ {*current epoch at* s_i}
3: $p_i \leftarrow 1$ {*primary in epoch* e_i}

4: **To execute a transaction...**

5: **when** receive (`request`) from c
6: **case** `request` = `begin`(t_a):
7: **if** $s_i \neq p_i$ **then** {*if not the primary...*}
8: send (e_i, t_a, "I'M NOT PRIMARY") to c {*...refuse to process* t_a}
9: **else**
10: $state(t_a) \leftarrow$ EXECUTING {*set* t_a*'s current state and...*}
11: `begin`(t_a) {*...start transaction*}
12: **wait for** response ($t_a, result$)
13: send ($t_a, result$) to c
14: **case** (`request` = `exec`(t_a,`sql-st`)) **and** ($state(t_a)$ = EXECUTING):
15: **exec task** {*execute SQL statement...*}
16: `exec`(t_a,`sql-st`) {*... concurrently with other requests*}
17: **wait for** `response`($t_a, result$)
18: **if** $result$ = ABORTED **then** $state(t_a) \leftarrow$ ABORTED
19: send ($t_a, result$) to c
20: **case** (`request` = `commit`(t_a)) **and** ($state(t_a)$ = EXECUTING):
21: $state(t_a) \leftarrow$ COMMITTING {*update* t_a*'s current state, ...*}
22: `sqlSeq`(t_a) $\leftarrow$ all `exec`(t_a,`sql-st`) statements in order {*gather SQL statements for* t_a*, and...*}
23: broadcast(s_i, e_i, c, t_a, `sqlSeq`(t_a)) {*...broadcast everything to everybody*}
24: **case** (`request` = `abort`(t_a)) **and** ($state(t_a)$ = EXECUTING):
25: $state(t_a) \leftarrow$ ABORTED {*update* t_a*'s current state, and...*}
26: `abort`(t_a) {*...tell the database to abort* t_a}

27: **To commit a transaction...**

28: **when** deliver(s_j, e_j, c, t_a, `sqlSeq`(t_a))
29: **if** $e_j < e_i$ **then** {*if* t_a *didn't execute in the current epoch...*}
30: $state(t_a) \leftarrow$ ABORTED {*...*t_a *has to be aborted*}
31: `abort`(t_a) {*...ditto*}
32: **else**
33: **if** $s_i \neq p_i$ **then** {*if* t_a *didn't execute on* s_i*...*}
34: $state(t_a) \leftarrow$ COMMITTING {*...update current status, and...*}
35: `begin`(t_a) {*...send whole transaction to the database*}
36: **for every** `exec`(t_a,`sql-st`) $\in$ `sqlSeq`(t_a) **do**
37: `exec`(t_a,`sql-st`)
38: **wait for** `response`($t_a, result$)
39: `commit`(t_a) {t_a *can be committed now...*}
40: **wait for** `response`($t_a, result$)
41: $state(t_a) \leftarrow$ COMMITTED {*...done!*}
42: send ($t_a, state(t_a)$) to c {*in any case, send results to the client*}

Algorithm 2 (cont.) Database server s_i

```
43: To change an epoch...

44: when p_i ∈ D_i
45:   broadcast(e_i, "NEW EPOCH")

46: when deliver(e_j, "NEW EPOCH") and (e_j = e_i)
47:   if p_i = s_i then
48:     for every t_a such that state(t_a) = EXECUTING do
49:       state(t_a) ← ABORTED
50:       abort(t_a)
51:       send (t_a, ABORTED) to c
52:   p_i ← (e_i mod n) + 1
53:   e_i ← e_i + 1
```

7-8), but if the client sends the $\texttt{begin}(t_a)$ request to the primary, the primary initializes the transaction's state (lines 10-12) and returns an acknowledgement that the database is ready to process $\texttt{exec}(t_a, -)$ requests. If the request is an $\texttt{exec}(t_a, \textit{sql-st})$, the server executes it as an independent task (lines 15-19). This is done to prevent the server from executing requests sequentially against the database. If the primary receives a commit request for transaction t_a, it updates t_a's current state to committing (line 21), and broadcasts t_a's identifier, the epoch when t_a executed, the client identifier associated with t_a, and t_a's operations to all database servers (line 23). The $\texttt{exec}(t_a, -)$ statements are broadcast as a sequence. This allows the backups to execute t_a's requests in the same order as the primary.

- **Committing transactions (lines 28-42).** Committing transactions are delivered by the *when* statement at line 28. After delivering some transaction t_a, a server first validates t_a (line 29). The validation consists in checking whether the epoch t_a executed in is the current epoch. If it is not, t_a is aborted by the server (lines 30-31). If t_a executed in the current epoch, t_a passes the validation test and is locally committed (lines 33-41). At the primary, committing t_a consists of issuing a database commit request and waiting for the response (lines 39-40). At the backups, committing t_a consists of executing all t_a's $\texttt{exec}(t_a, -)$ operations (lines 34-38), issuing a database commit operation, and waiting for the response.

- **Changing epochs (lines 44-53).** The *when* statements at lines 44 and 46 handle epoch changes. When a server s_i suspects the current primary to have crashed, s_i broadcasts a 'new epoch' message (lines 44-45). Upon delivering the 'new epoch' message, a server determines the new primary from the epoch number (line 52) and updates the current

epoch (line 53). If the primary has not crashed (i.e., in case of false suspicion), it also delivers the 'new epoch' message, and before passing to the next epoch, it aborts all local transactions in execution (lines 48-51). Aborting local transactions in execution before passing to the next epoch is an optimization as no such transaction will pass the validation test: these transactions will not be delivered in the epoch in which they executed.

3.3 Algorithm Correctness

Correctness proofs are given in the Appendix. We present here the intuitive ideas behind the correctness of the Pronto protocol. In order for the protocol to be correct, properties RDB-1 (safety) and RDB-2 (liveness) have to be guaranteed.

The protocol is safe. Property RDB-1 follows from the fact that (a) at any epoch e, the primary and the backups that do not crash commit the same transactions in the same order, and (b) an execution involving several epochs (or primaries) is equivalent to an execution with only one epoch (or primary).

(a) To commit a transaction in e, the primary first has to deliver this transaction. From the properties of the broadcast primitive, the backups that do not crash also deliver this transaction. Since the transaction commits in the primary, and the validation test is deterministic, the transaction is committed in the backups. Commit order is preserved since transactions are delivered in an order that satisfies the execution order generated by the 2PL scheduler on the primary, and backups commit transactions in the order they are delivered.

(b) Committed transactions that execute in some epoch e are serialized by the primary, and committed transactions that execute in different epochs, say e and $e+1$, are guaranteed to execute sequentially: no transaction that executes in epoch e and is not delivered and committed before the epoch changes to $e+1$ will pass the validation test in any server. Therefore, every operation performed by a committed transaction that executed in epoch e happens before any operation performed by every transaction that executes in epoch $e+1$.

The protocol is live. It can be proved that any client that submits a job j_a and does not crash eventually contacts a primary that executes and commits a transaction t_a generated by j_a. The argument is that, firstly, no client blocks forever in the *wait* statements at lines 9 and 16. In both cases, if the primary crashes, the client suspects it, and tries to execute the job

in another server. If the primary does not crash, it will send a response to the client. Thus, the only reasons why a client would not be able to execute job j_a are (a) the client cannot find the primary, or (b) the client finds the primary and starts executing the job in the primary but before terminating the job, the epoch changes and another server becomes primary (i.e., the transaction generated by the job is aborted due to an epoch change), or (c) the primary does not change but keeps aborting the transactions generated by j_a. Cases (a) and (b) never happen because there is a time when some server p that does not crash is not suspected by any other process to have crashed. This server eventually becomes primary and remains primary forever. So, the client will eventually contact p, start a transaction on p, and never suspect p. The only alternative left is (c), where the primary keeps aborting transactions generated by j_a. However, this cannot happen because it would contradict database property DB-2. Therefore, every client that does not crash eventually contacts a primary that executes and commits a transaction generated by j_a.

4 Evaluation of the Protocol

4.1 The Cost of Broadcasting Transactions

In a system with a single database server, when a client requests the commit of some transaction, the server tries to execute the commit operation against the database, and returns the answer to the client. In the Pronto protocol, before a server tries to execute the commit operation, it has to deliver and validate the transaction. Since the validation test is very simple (actually an integer comparison), the overhead introduced in the transaction response time by the Pronto protocol is expected to be mostly due to the broadcast primitive.

To discuss the cost of broadcasting transactions, we consider the broadcast algorithm presented by Chandra and Toueg [4] (CT-broadcast), and the Optimistic Atomic Broadcast algorithm presented in [12] (OPT-broadcast). Both algorithms are non-blocking and tolerate an infinite number of false failure suspicions. Briefly, with the CT-broadcast algorithm, broadcast messages are first sent to the servers, and then the servers decide on a common delivery order for the messages. The OPT-broadcast makes some optimistic assumptions about the system (e.g., by taking into account the hardware characteristics of the network) to deliver messages fast. The key observation is that in some cases, there is a good probability that messages arrive at their destinations in a total order, and so, servers do not have to decide on a common delivery order. Servers have to check whether the order is the same, and if this is the case, OPT-broadcast is "cheaper" (i.e., requires fewer messages) than CT-broadcast. Otherwise, OPT-broadcast is "more expensive" than CT-broadcast.

Table 1 compares CT-broadcast and OPT-broadcast to atomic commitment protocols. This comparison is only done for reference purposes, as atomic commitment protocols have traditionally been used by databases to terminate transactions. Table 1 shows best case scenarios for each algorithm.[2] When comparing protocols with the same resilience, both CT-broadcast and OPT-broadcast outperform atomic commitment protocols. 2PC is as expensive as OPT-broadcast, however, OPT-broadcast does not block the system in the event of crashes. Therefore, the overhead introduced by broadcasting transactions in Pronto is smaller than the cost of a non blocking atomic commitment protocol.

Protocol	**Resilience**	**Number of steps**	**Number of messages**
CT-broadcast	non-blocking	4	$4(n-1)$
OPT-broadcast	non-blocking	3	$3(n-1)$
2PC	blocking	3	$3(n-1)$
3PC	non-blocking	5	$5(n-1)$

Table 1: Cost of broadcasting a transaction

4.2 Transaction Operations and the Validation Test

Pronto uses a very simple validation test. It only requires to know whether the epoch when a transaction executed is the same as the current epoch. This actually comes down to comparing two integers. We can compare this validation test to more detailed validation tests used in optimistic concurrency control protocols [2, 9]. On one side, the Pronto validation test needs little information about transactions (only the epoch number), does not require much storage space, and allows for a fast validation mechanism. On the other side, by using limited information about transaction operations, the Pronto validation test aborts more transactions than detailed validation tests.

For example, consider that in epoch e, database servers validate a transaction t that has executed in some epoch $(e-1)$. According to the Pronto validation test, t has to be aborted. However, if t does not access any data item accessed by any transaction committed in epoch e, t could actually be committed in epoch e and database consistency would still be preserved. This example shows a tradeoff between the amount of information the validation test has about transactions and the number of transactions that pass the test. We have opted for a simple

[2] A best case scenario for 2PC and 3PC is an execution with no crashes. The best case for CT-broadcast and OPT-broadcast is an execution with no crashes and no failure suspicions. For OPT-broadcast, it is also the case that the optimistic assumption is verified.

validation test not only for the reasons already mentioned, but also because a more complex test would require parsing SQL statements to know which data items are accessed by the transactions.

5 Related Work

In the following, we review commercial database products and research projects that address similar topics as the Pronto protocol. Failover mechanisms with strong consistency constraints (i.e., serializability) have often been related to parallel database systems by the main commercial database players. A parallel database system has a cluster of database processes that access the same data. Such systems typically provide a notion of failover. If a client accesses the database through one process, it can switch to another process if the first one fails. Besides failover for availability, a parallel database system also provides concurrent processing of SQL queries. There are two main ways to build a parallel database system: shared disk and shared nothing.

The Oracle Parallel Server (OPS) [11] is an example of a shared disk approach. All processes access the same disk, but they have separate transaction logs, and they coordinate their updates through a distributed lock manager. The Informix Extended Parallel Server (XPS) [17] uses the shared nothing approach. Each process has its own disk, but if a process fails, another process mounts the failed process' disk. Compared to traditional single-process database systems, a parallel database system provides faster recovery because clients can fail over to another process. However, a fail-over still requires log-based recovery: if one process takes over for a failed process, it must reconcile its own state with the log of the failed process. Moreover, to use a parallel database system as a highly available transaction processing system, we need database processes on different machines to access the same disks. This requires special hardware/software, such as high-availability clusters. In contrast, our approach does not inject log-based recovery into the fail-over process, and we rely on standard hardware and software. Further, we do not use our database ensemble as a parallel-processing platform — we focus exclusively on high availability.

A number of commercial database systems support asynchronous data replication. A primary database processes transactions, and asynchronously sends data updates to a number of standby databases. For example, the Tandem Remote Data Facility (RDF) [14] ships the transaction log to remote sites, and the Microsoft SQL Server [10] ships data operations to remote sites for committed transactions. Since these systems propagate data updates in an asynchronous manner, data updates can be lost in a fail over. Moreover, these systems can typically not provide consistency in the presence of multiple primaries. In contrast, our protocol provides consistency in the multi-primary case. Furthermore, we do not lose transactions — the client "stub" transparently replays in-progress transactions after the fail-over is complete.

Several academic and industrial research projects have addressed database failover mostly based on some form of database replication. Database replication techniques can be divided in two groups: active replication and primary backup [6, 8].

With active replication [15], all database servers, one way or another, execute the clients' requests, and return the results to the clients. Database server communication is usually based on a broadcast primitive with similar guarantees to the one described in this paper (i.e., Atomic Broadcast). Database replication techniques based on active replication can provide fast failover and, to a certain extent, failure transparency to the clients. Active replication, however, requires strong deterministic assumptions on the way databases process transactions [1]. In general, database replication techniques based on active replication require stronger assumptions about individual databases than the assumptions made by the Pronto protocol (e.g., having access to database internal modules), and/or assume a certain knowledge about the semantics of the operations. An exception is [13], which presents a weaker form of active replication to solve non-determinism due to preemption in replicated real-time applications.

Primary-backup replication techniques have been studied and applied from both a theoretical and practical viewpoint [2, 3, 5, 7]. Recent work [18] has considered using virtual memory mapped communication to achieve fast failover by mirroring the primary's memory on the backups. A common assumption made by most replication techniques based on primary-backup is to assume a perfect failure-detector mechanism.

6 Conclusion

This paper introduces Pronto, a protocol that allows clients to access an ensemble of databases as if it were a single database. Clients access the ensemble through a standard interface, such as JDBC. The benefit of using an ensemble instead of a single database is that the ensemble can provide un-interrupted service in the presence of database failures. This capability provides a highly-available transaction-processing system, and allows enterprise applications to be continuously available.

Pronto relies on practical assumptions: clients access databases through standard interfaces, databases from different vendors can be part of the ensemble (the only requirement is to support the standard interfaces), and the correctness of the system does not need perfect failure detection. Although Pronto is not fully operational yet, we expect that its synchronization mechanism will orchestrate the database ensemble with a reasonable performance overhead.

There are multiple directions for future work. One interesting avenue of research we are presently looking at is to define alternative data consistency criteria besides one-copy-serializability.

In practice, single databases typically support a range of isolation levels, but it is not clear how to "translate" these per-database isolation levels into a global, ensemble level, data consistency notion.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, Passau (Germany), September 1997.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Optimal primary-backup protocols. In *Distributed Algorithms, 6th International Workshop, WDAG '92*, volume 647 of *Lecture Notes in Computer Science*, pages 362–378, Haifa, Israel, 2–4 November 1992.

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

[5] L. Frank. Evaluation of the basic remote backup and replication methods for high availability databases. *Software Practice and Experience*, 29:1339–1353, 1999.

[6] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal (Canada), June 1996.

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

[8] R. Guerraoui and A. Schiper. Software based replication for fault tolerance. *IEEE Computer*, 30(4), April 1997.

[9] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[10] Replication for Microsoft SQL server 7.0, 1998. Microsoft Part Number 098-80829.

[11] Oracle Parallel Server for Windows NT clusters. Online White Paper.

[12] F. Pedone. *The Database State Machine and Group Communication Issues.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, December 1999. Number 2090.

[13] D. Powell, M. Chéréque, and D. Drackley. Fault-tolerance in Delta-4. *ACM Operating Systems Review, SIGOPS*, 25(2):122–125, April 1991.

[14] Compaq remote database facility product family. Online white-paper.

[15] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[16] S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner. *JDBC API Tutorial and Reference, 2nd edition.* Addison-Wesley, Menlo Park, California, 1994.

[17] Informix extended parallel server 8.3. Online White-Paper.

[18] Y. Zhou, P. Chen, and K. Li. Fast cluster failover using virtual memory-mapped communication. Technical Report TR-591-99, Department of Computer Science, Princeton University, January 1999.

Appendix

For the following proofs, $C(\mathcal{E}_i^e)$ is the committed projection of the execution $\mathcal{E}_i^e$, created by the Pronto protocol during epoch e at database server process s_i. $C(\mathcal{E}_i^e)$ is a partial order, that is, $C(\mathcal{E}_i^e) = (\Sigma_i^e, <_i^e)$, where Σ_i^e is the set of committed transactions in $\mathcal{E}_i^e$, and $<_i^e$ is a set defining a transitive binary relation between transactions in Σ_i^e. We define $C(\mathcal{E}^e) = C(\mathcal{E}_i^e) \cup C(\mathcal{E}_j^e)$ such that $\Sigma^e = \Sigma_i^e \cup \Sigma_j^e$ and $<^e = <_i^e \cup <_j^e$.

Lemma 1 *Let $s_{p(e)}$ be the primary database server at epoch e. For every backup server $s_{b(e)}$ that does not crash in epoch e, and all $e \geq 1$, $C(\mathcal{E}_{p(e)}^e) = C(\mathcal{E}_{b(e)}^e)$.*

PROOF (SKETCH): We have to show that database servers $s_{p(e)}$ and $s_{b(e)}$ commit the same transactions (i.e., $\Sigma_{p(e)}^e = \Sigma_{b(e)}^e$) in the same order (i.e., $<_{p(e)}^e = <_{b(e)}^e$). Assume that $s_{p(e)}$ commits transaction t during epoch e. Therefore, t passed the validation test, and so, $s_{p(e)}$ has executed t during epoch e, and has not delivered any message of the type (e, "NEW EPOCH") before delivering message $(s_{p(e)}, e, -, t, \texttt{sqlSeq}(t))$. From the agreement and total order properties of the broadcast primitive, server $s_{b(e)}$ also delivers message $(s_{p(e)}, e, -, t, \texttt{sqlSeq}(t))$ before delivering any message of the type (e, "NEW EPOCH"). Thus, t passes the validation test at $s_{b(e)}$ and is committed by $s_{b(e)}$.

We now show that $<_{p(e)}^e = <_{b(e)}^e$, that is, if a transaction t_a precedes another transaction t_b in $C(\mathcal{E}_{p(e)}^e)$, then t_a also precedes t_b in $C(\mathcal{E}_{b(e)}^e)$. Since server $s_{p(e)}$ executes transactions using a 2PL scheduler, if t_a precedes t_b, then t_a commits before t_b. From the algorihm, it follows that t_a is delivered before t_b. By the total order property of the broadcast primitive and the fact that backups commit transactions in the same order as they are delivered, server $s_{b(e)}$ delivers and commits t_a before t_b. Therefore, t_a precedes t_b in database server $s_{b(e)}$. □

Lemma 2 *For any execution $\mathcal{E}$ of the Pronto protocol, there exists a serial execution $\mathcal{E}_s$ involving the committed transactions in $\mathcal{E}$, such that $C(\mathcal{E})$ is equivalent to $C(\mathcal{E}_s)$.*

PROOF (SKETCH): From property DB-1, for any execution $\mathcal{E}_{p(e)}^e$, at epoch e, there exists a serial execution $\mathcal{E}_s^e$, involving the committed transactions in $\mathcal{E}_{p(e)}^e$, such that $C(\mathcal{E}_{p(e)}^e)$ is equivalent to $C(\mathcal{E}_s^e)$. By Lemma 1, for every backup $s_{b(e)}$ that does not crash in epoch e, and all $e > 1$, $C(\mathcal{E}_{p(e)}^e) = C(\mathcal{E}_{b(e)}^e)$. Thus, from the definition of committed projection, $C(\mathcal{E}_{p(e)}^e) \cup C(\mathcal{E}_{b_1(e)}^e) \cup ... \cup C(\mathcal{E}_{b_{m(e)-1}(e)}^e) = C(\mathcal{E}_{p(e)}^e)$, where $m(e)$ is the number of database servers that do not crash in epoch e. Thus, $C(\mathcal{E}_{p(e)}^e) \cup C(\mathcal{E}_{b_1(e)}^e) \cup ... \cup C(\mathcal{E}_{b_{m(e)-1}(e)}^e)$ is equivalent to $C(\mathcal{E}_s^e)$.

We claim that $\cup_{e=1} C(\mathcal{E}_s^e)$ is equivalent to $C(\mathcal{E}_s)$. The proof for the claim follows from the fact that for all $e \geq 1$, the Pronto protocol ensures that executions $\mathcal{E}_s^e$ and $\mathcal{E}_s^{e+1}$ are executed

sequentially. That is, every transaction that executes in epoch $e+1$ starts after all transactions that commit in epoch e have been committed. We conclude that for any execution $\mathcal{E}$, there exists an execution $\mathcal{E}_s$, such that $C(\mathcal{E})$ is equivalent to $C(\mathcal{E}_s)$. □

Lemma 3 *If a client c executes* ***submit***(j) *and does not crash, then c eventually executes* ***response***$(result)$.

PROOF (SKETCH): We show that client c eventually contacts a primary that executes and commits a transaction t_a generated by j_a. The argument is that, firstly, no client blocks forever in the *wait* statements at lines 9 and 16. In both cases, if the primary crashes, the client suspects it, and tries to execute the job in another server. If the primary does not crash, it will send a response to the client: in the *wait* statement at line 9, the client has requested a `begin`$(-)$ operation, and a database is always able to execute it; in the *wait* statement at line 16, if the request cannot be processed by the primary (e.g., the transaction is deadlocked), the primary aborts the transaction and replies to the client.

Thus, the only reasons why a client would not be able to execute job j are (a) the client cannot find the primary, or (b) the client finds the primary and starts executing the job in the primary but before terminating the job, the epoch changes and another server becomes primary (i.e., the transaction generated by the job is aborted due to an epoch change), or (c) the primary does not change but keeps aborting the transactions generated by j_a. Cases (a) and (b) never happen because there is a time when some server s_p that does not crash is not suspected by any other process to have crashed. This server eventually becomes primary and remains primary forever. So, the client will eventually contact s_p, start a transaction on s_p, and never suspect s_p. The only alternative left is (c), where the primary keeps aborting transactions generated by j_a. However, this contradicts database property DB-2 because s_p does not crash, c does not suspect it and keeps sending transactions generated by executions of $j)a$, so, eventually, one transaction terminates and job j_a is successfully executed. We conclude that any client eventually contacts a primary that executes and commits a transaction generated by j_a. □