# Pronto: High Availability for Standard Off-the-shelf Databases[*,†]

Fernando Pedone[⋆] Svend Frølund[†]

[⋆]University of Lugano (USI)
Switzerland

[†]Gatehouse A/S
Denmark

**Abstract**

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. Pronto is a novel replication protocol that handles non-determinism without relying on perfect failure detection, does not require any modifications in existing applications and databases, and allows databases from different providers to be part of the replicated compound.

**Index terms:** database replication, failover, heterogeneous databases, primary-backup, atomic broadcast

[*]A shorter version of this paper appeared previously in the Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS), October 2000.

[†]Part of the work done while the authors were at Hewlett-Packard Laboratories (HP Labs), USA.

# 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, which 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. In contrast, web servers are typically stateless. 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 a database from a failure. Thus, one way to reduce the impact of database failures on the overall system downtime and to increase the availability of the three-tier application is to use multiple replicated databases in the back-end.

This paper presents Pronto, a protocol that orchestrates the execution of replicated databases. Pronto uses standard, off-the-shelf, and possibly heterogeneous, database systems for the back-end transaction processing. Moreover, Pronto guarantees strong consistency (i.e., one-copy serializability [9]), providing to the clients the illusion that the database ensemble is a single, highly-available database system. Being highly available means that users of the ensemble never wait for database recovery, even if individual databases in the ensemble fail.

Pronto supports "interactive" transactions whose structure and SQL statements are not known up front. This allows Pronto to be deployed as a special implementation of a standard transaction interface, such as JDBC [54]. With JDBC, transactions may be constructed incrementally, passing only a single SQL statement through the interface at a time. Deploying Pronto as a special implementation of JDBC means that neither the databases nor the application require changes—we can force the application to load the Pronto JDBC driver instead of the standard JDBC driver by simply changing the CLASSPATH variable.

Pronto is a hybrid between primary-backup replication [12] and active replication [31, 51],[1] similarly to the leader-follower approach [7], developed in the context of real-time systems. Essentially, Pronto deals with database non-determinism by having a single (primary) database execute transactions in a non-deterministic manner. Rather than checkpoint the resulting state

[1]Active replication is a technique according to which replicas receive and execute all requests in the same total order. If the execution is deterministic, after having executed the same sequence of operations, every replica will reach the same local state [31, 51].

to backups, the primary sends 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, Pronto can support heterogeneous databases with different log formats, and prevent the contamination that may result if the data in one database becomes corrupt.

Primary-backup techniques usually rely on *failure detectors* (i.e., timeout mechanisms) to trigger the election of a new primary when the current one fails [9]. Setting the right timeout value is not straightforward, however. While fast failure detection is crucial for high availability—the longer it takes for the failure of the primary to be detected, the smaller the availability of the system—being too aggressive may result in false detections and multiple primaries executing simultaneously, which may compromise consistency. Pronto's solution to this problem is to allow the existence of several primaries during certain transitory conditions without leading to database inconsistencies. With Pronto, timeout mechanisms can be tuned to aggressively detect primary failures, without incurring inconsistencies due to false failures suspicions.

A remark is now in order considering Pronto's approach. Pronto was not designed to "compete" in performance with proprietary solutions. Even though the overhead introduced by Pronto is "acceptable"—experiments conducted with an ensemble of commercial databases, using a Java prototype interfaced through JDBC, have shown that when compared to a single-database configuration the overhead introduced by Pronto is below 20% in throughput and below 26% in response time—access to database internals allows optimizations not possible in a middleware protocol. On the other hand, Pronto is reasonably simple, can be readily put to use with off-the-shelf databases, and addresses the very practical problem of failure detection tuning.

The remainder of the paper is structured as follows. Section 2 describes the system model and some abstractions used in the paper, and formally states the problem we are concerned about. Section 3 presents the Pronto protocol in detail and discusses some additional aspects about Pronto. Section 4 describes our Pronto prototype, the experiments we conducted with it, and the results found. Section 6 concludes the paper. Proofs of correctness are presented in the Appendix.

# 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 (e.g., 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 (e.g., we do not consider Byzantine failures). For simplicity, database recovery is not introduced in the model, and treated later in the paper (see Section 3.4). We further assume that database clients and servers have access to failure detectors [15], 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)* [15].

Processes are all connected through reliable channels, which neither lose nor duplicate messages. 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 they do so in the same order *(total order)*.

## 2.2 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.

- $\texttt{exec}(t_a, \textit{sql-st})$ and $\texttt{response}(t_a, \textit{result})$. 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). The result can be the values returned from *sql-st* or an abort notification if $t_a$ was involved in a deadlock, for example.

- $\texttt{commit}(t_a)$ and $\texttt{response}(t_a, \textit{result})$. 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 or an abort notification.

- $\texttt{abort}(t_a)$. The abort primitive terminates a sequence of requests for transaction $t_a$ and requests $t_a$'s abort. We assume that when submitted to the database, 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 $\langle \texttt{begin}(t_a); \texttt{response}(t_a, -); \texttt{exec}(t_a, -); \texttt{response}(t_a, -); \ldots; [\,\texttt{commit}(t_a); \texttt{response}(t_a, -) / \texttt{abort}(t_a)\,]\,\rangle$, where $\texttt{commit}(t_a); \texttt{response}(t_a, -)$ and $\texttt{abort}(t_a)$ are mutually exclusive, although one must take place. Our definition of a transaction differs from traditional definitions (e.g., [9]) 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 a valid transaction is continuously re-submitted, it is eventually committed.

**DB-3** No transaction remains pending forever.

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 $\mathcal{E}$. Furthermore, strict 2PL schedulers have the following property, exploited by our protocol: if transaction $t_a$ has been committed before transaction $t_b$ by $s_i$ in some execution $\mathcal{E}$, and $t_a$ and $t_b$ conflict,[2] then in every serial execution $\mathcal{E}_s$ equivalent to $\mathcal{E}$, $t_a$ precedes $t_b$ [9].

[2]Transactions $t_a$ and $t_b$ conflict if they both access a common data item and at least one of them modifies the data item.

Property DB-2, although not explicitly stated as such, is often assumed to be satisfied by current database systems. A transaction is valid if it does not violate any integrity constraints (e.g., withdrawing money from an account without enough funds). DB-2 reflects the fact that even though 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), this does not mean that no liveness guarantee is provided.

Property DB-3 ensures that transactions terminate, either by committing, i.e., making their changes permanent to the database, or aborting, i.e., rolling them back. Thus, if a transaction cannot make progress because it is deadlocked or the client submitting the transaction's operations has crashed, the transaction will be eventually aborted by the database (and all its locks will be removed).

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

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.

Pronto uses the transactional-job abstraction to allow 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 the client will eventually receive a response for $j$.

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 against the database system.

# 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 job. The other database servers (the *backups*) interact only with the primary.

### 3.1.1 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 $s$ that $c$ believes to be, most likely, the current primary. After sending the begin transaction request, $c$ waits for the result or suspects $s$ to have crashed. The execution proceeds as follows (see Figure 1).

(a) If $s$ 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 $s$ is not the current primary, it returns an error message to $c$, which will choose another database server $s'$ and send to $s'$ the transaction request.

(c) The case where $c$ suspects $s$ to have crashed is treated later in the section.

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 (defined later in the section) to all backups.

Upon delivering a committing transaction, each database server executes a validation test to decide to commit or abort the transaction and sends the transaction's outcome to the client. The validation test depends on the delivery order of transactions. Since this order is the same for all servers and the validation test is deterministic, no two servers reach different outcomes (i.e., "commit" or "abort") for the same transaction.

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 or from the backup servers or from both. If the outcome is "commit" or "invalid" (i.e., the transaction should be aborted because it violates integrity constraints), the client issues a response for job $j$. Otherwise the client has to re-execute job $j$. A server may decide to abort a valid transaction during its execution if it finds out that the transaction is involved in a (local) deadlock or it suspects that the client submitting the transaction has crashed.

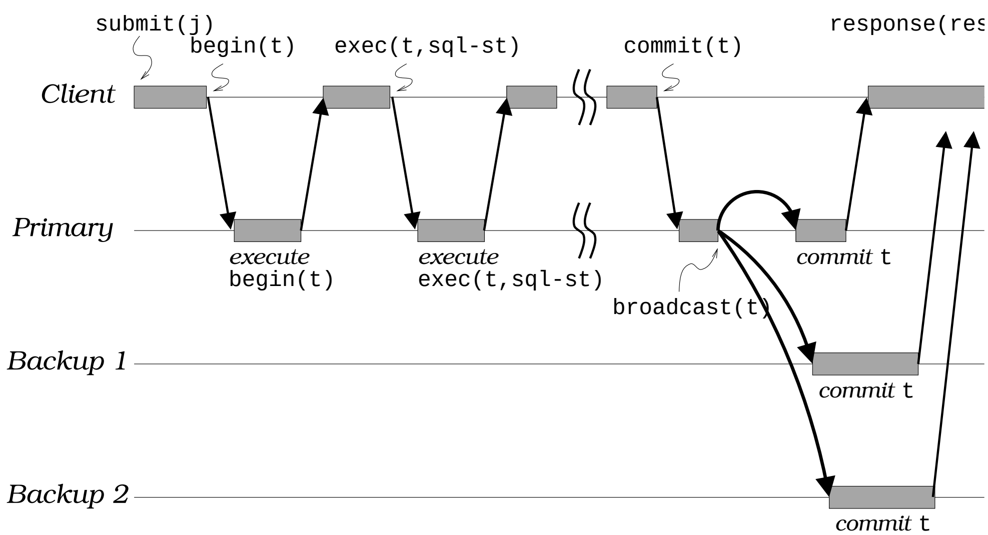



Figure 1: Pronto without crashes and suspicions

Transactions pass 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.

### 3.1.2 Failure/Suspicion Handling

After submitting a request, a client $c$ may suspect the primary $s$ to have crashed. This may happen because $s$ has crashed or because $c$ incorrectly suspects $s$. In either case, $c$ sends an abort transaction message to $s$, just in case it was a false suspicion, chooses another database server $s'$, and re-executes $j$, by sending the begin transaction request, using $s'$.

If the primary crashes or is suspected to have crashed, the execution evolves as a sequence of *epochs*[3]; in executions in which the primary does not crash nor is suspected to have crashed, however, there is only one epoch. 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 in which it executed. The validation test ensures that a transaction is only committed by some database server if the epoch in which the database server delivers the transaction and the epoch in which the transaction was executed are the same.

As a consequence, Pronto can tolerate inaccurate failure detection without hurting consistency. For performance reasons, however, failure detection should be as accurate as possible since it may lead to unnecessary aborts and resubmission of transactions. In any case, Pronto's modular design results in a primary-backup technique that does not have to care about how failure detection is implemented.

[3]The notion of epoch used in Pronto differs from the one in [21]. Epochs are used in [21] to synchronize the installation of log records coming from transactions that executed at primary sites.

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 delivered after the new epoch message, and so, it is aborted since $t$ did not execute in the epoch in which it is delivered. Transaction $t'$ is committed by every database server after it is delivered since it executed and is delivered in the same epoch. Although not shown in Figure 2, after validating a transaction, each database server sends the transaction outcome to the client.

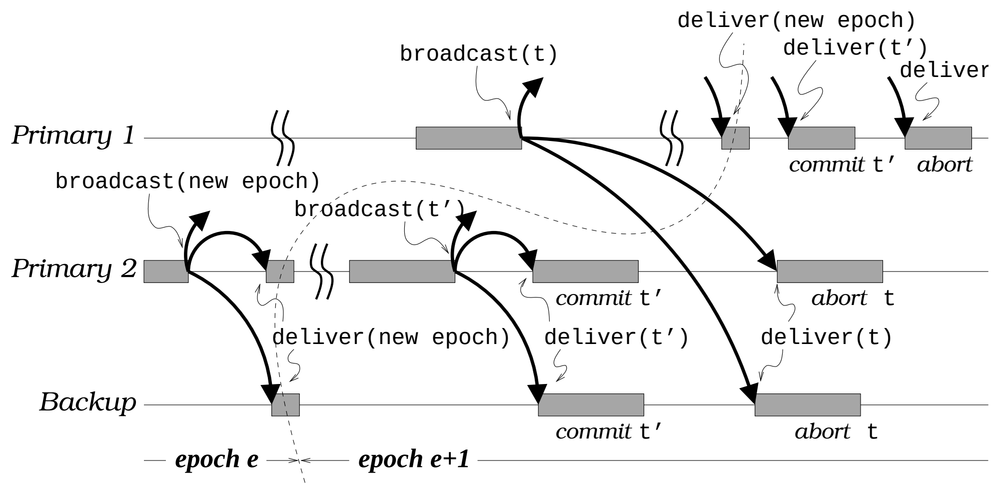



Figure 2: Two primaries scenario

Failures and failure suspicions may lead to situations in which a client should execute the same transactional job twice. Since communication is asynchronous, this may happen even in cases in which the first execution was successful, and the corresponding transaction committed (e.g., the primary fails after committing the transaction, its reply is lost, and it takes "too long" for the backups to reply to the client). As we discuss in the next sections, if required by the application, *exatly-once semantics*—i.e., only one transaction is committed despite multiple executions of the same transactional job—can be easily integrated in Pronto.

### 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 on this server. Clients (and servers) interact with their failure detection module by inspecting variable $\mathcal{D}$. Client $c$ suspects database server process $s$ if $s \in \mathcal{D}_c$.

- **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). In the latter case, the client sends an abort request to the primary and starts again, possibly with a different primary. The message sent by the client is an optimization, which in case of a wrong suspicion, will not leave the transaction hanging in the server and possibly blocking other transactions. 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. If the primary decides to abort a valid transaction (i.e., due to a local deadlock), the client re-executes the job using the same database server.

In both parts of the algorithm, when a client suspects a primary $p_c$ (lines 10, 18, 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 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). For brevity, we do not present the server code handling transactions aborted by a database during their execution (e.g., due to a local deadlock).

- **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 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 acknowledgment 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

**Algorithm 1** Database client $c$

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

  p_c ← 0
  findPrimary ← true
  repeat
    repeat
      request ← j_a(⊥)
      if findPrimary = true then
        p_c ← (p_c mod n) + 1
      send (t_a, request) to p_c
      wait until (receive (t_a, result) from p_c) or (p_c ∈ D_c)
      if not(received (t_a, result) from p_c) then
        send (t_a, abort(t_a)) to p_c
    until (received (t_a, result) from p_c) and (result ≠ "I'M NOT PRIMARY")
    findPrimary ← false

    repeat
      request ← j_a(result)
      send (t_a, request) to p_c
      wait until (receive (t_a, result) or (p_c ∈ D_c)
      if p_c ∈ D_c then
        send (t_a, abort(t_a)) to p_c
        result ← ABORTED
        findPrimary ← true
    until (result ∈ {COMMITTED, INVALID, ABORTED})
  until (request = ABORT) or (request = COMMIT and result ≠ ABORTED)
  response(result)
```

**Algorithm 2** Database server $s_i$

```
Initialization...

  e_i ← 1
  p_i ← 1

To execute a transaction...

when receive (request) from c
  case request = begin(t_a):
    if s_i ≠ p_i then
      send (e_i, t_a, "I'M NOT PRIMARY") to c
    else
      state(t_a) ← EXECUTING
      begin(t_a)
      wait for response (t_a, result)
      send (t_a, result) to c
  case (request = exec(t_a,sql-st)) and (state(t_a) = EXECUTING):
    exec task
      exec(t_a,sql-st)
      wait for response(t_a, result)
      if result ∈ {INVALID, ABORTED} then state(t_a) ← ABORTED
      send (t_a, result) to c
  case (request = commit(t_a)) and (state(t_a) = EXECUTING):
    state(t_a) ← COMMITTING
    sqlSeq(t_a) ← all exec(t_a,sql-st) in order
    broadcast(s_i, e_i, c, t_a, sqlSeq(t_a))
  case (request = abort(t_a)) and (state(t_a) = EXECUTING):
    state(t_a) ← ABORTED
    abort(t_a)

To commit a transaction...

when deliver(s_j, e_j, c, t_a, sqlSeq(t_a))
  if e_j < e_i then
    state(t_a) ← ABORTED
    if s_i = p_i then abort(t_a)
  else
    if s_i ≠ p_i then
      state(t_a) ← COMMITTING
      begin(t_a)
      for each exec(t_a,sql-st)∈sqlSeq(t_a) do
        exec(t_a,sql-st)
        wait for response(t_a, result)
    commit(t_a)
    wait for response(t_a, result)
    if result ≠ INVALID then state(t_a) ← COMMITTED else state(t_a) ← ABORTED
  send (t_a, result) to c
```

**Algorithm 2 (cont.)** Database server $s_i$

43: **To change an epoch...**

44: **when** $p_i \in \mathcal{D}_i$
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: $state(t_a) \leftarrow$ ABORTED
50: `abort`$(t_a)$
51: send $(t_a, \text{ABORTED})$ to $c$
52: $p_i \leftarrow (e_i \textbf{ mod } n) + 1$
53: $e_i \leftarrow e_i + 1$

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 `exec`$(t_a, -)$ statements are broadcast as a sequence. This allows the backups to execute $t_a$'s requests in the same order as they were executed by the primary. Notice that if read and write requests can be easily distinguished, then read statements do not have to be broadcast to the database servers.

- **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 in which $t_a$ executed 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 `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 $s$ suspects the current primary to have crashed, $s$ 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 transactions will pass the validation test: these transactions will not be delivered in the epoch in which they executed.

### 3.3 Exactly-Once Transactional Jobs

In some systems, it is crucial to ensure that a transactional job commits only a single transaction in each database. Consider the case of a client that detects the crash of the current primary server. To ensure progress, the client may have to re-execute its transactional job against the new primary. However, if the former primary has managed to execute the client's job to completion (e.g., it failed after committing the transaction but before notifying the client), in the end two transactions will be committed by the new primary as the result of a single transactional job execution. An exactly-once transactional job prevents such a case from happening.

There are several mechanisms to ensure the exactly-once property [20]. In principle, clients should assign unique identifiers to transactional jobs and servers should keep track of previously executed jobs, making sure that two transactions for the same job will not both commit. In Pronto, this check can be executed after a transaction commit request is delivered by the database. If a transaction for the same job has already been committed, the server aborts the second transaction and notifies the client. The client can also be notified before submitting the transaction, as soon as the server realizes that the job has already been dealt with.

Clients can obtain unique identifiers by combining a local monotonically increasing sequencer (e.g., the local clock) with their own unique identifiers (e.g., their IP addresses). Keeping track of previously executed transactional jobs can be done in many ways. They can be kept in memory, and asynchronously written to disk, if it is unlikely that all servers simultaneously fail. Alternatively, they can be stored in the database as part of the committing transaction.

### 3.4 Database Recovery

Databases interact with each other in Pronto using only the underlying Atomic Broadcast abstraction. The agreement and total order properties of Atomic Broadcast ensure a natural way to deal with database recovery: While agreement forces failing databases to deliver missed transactions upon recovery, total order prevents a failing database from delivering messages in the wrong order. This uniform treatment of failing and operational databases ensures that once a database recovers, the order in which it committed its transactions is consistent with the order other databases committed the same transactions.

Known Atomic Broadcast protocols implement uniform agreement and total order in the presence of crash and recovery in different ways [32, 49]. The protocol in [49], for example, allows a recovering process to retrieve the entire history of delivered messages. That is, the recovering process has access to all messages it delivered before it crashed and all messages that other processes delivered while it was down. In its simplest form, a recovering process in Pronto would have to replay the entire history of delivered messages. This would bring the epoch variable $e_i$ up to date, and it would execute all transactions that the process missed while it was down. Notice that the exactly-once mechanisms discussed in Section 3.3 ensure that transactions are not re-executed during recovery.

Even though such an approach hides the complexities related to recovery, it may not be efficient if the recovering database has missed "too many" transactions: the cost of catching up with the operational databases by processing missing transactions may become higher than simply copying the database from an up-to-date site. A more general discussion about how to efficiently integrate group communication primitives and database recovery is beyond the scope of this paper. For further reference see, e.g., [6, 26, 29].

# 4 Evaluation of the Protocol

## 4.1 Analytical Evaluation

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 Optimistic Atomic Broadcast algorithm [42] (OPT-broadcast) and the Paxos algorithm [33]. Both algorithms are non-blocking and tolerate an unbounded number of false failure suspicions.

The OPT-broadcast makes the optimistic assumption that in some networks there is a good probability that messages arrive at their destinations in a total order. When this assumption holds, messages can be delivered "fast"; when it does not, the algorithm incurs in further computation to ensure that total order is preserved [42]. If $\delta$ is the transmission delay along communication links, then the *latency*, that is, the time between the primary broadcasts and delivers a transaction, is $2\delta$. When communication between servers is point-to-point, OPT-broadcast injects $(n+1)(n-1)$ messages in the network, where $n$ is the number of servers. If a network-level broadcast primitive is available (e.g., IP-multicast), then $(n+1)$ messages are generated per broadcast.

The complete Paxos protocol has latency of $4\delta$. It injects $5(n-1)$ messages when communication is point-to-point, and $2(n+1)$ messages when communication uses network-level broadcast. If most messages are broadcast by the same sender, which is the expected behavior of Pronto, where most messages are update transactions broadcast by the primary, the protocol can be optimized leading to a latency of $2\delta$, and $3(n-1)$ and $n+1$ messages for point-to-point and low-level broadcast communication, respectively. Backups rely on the original protocol.

As a reference, we also consider the performance of 1-safe and 2-safe primary-backup replication [24]. Although [24] considers a single backup, we have specified the complexity for $n-1$ backups. Using the 1-safe configuration, the primary can commit a transaction before exchanging messages with the backups, however, the backups may miss some transactions if the primary crashes. This leads to zero latency, and $n-1$ point-to-point messages or 1 network-level broadcast. To commit a transaction using the 2-safe configuration, the primary has to wait for a round-trip message with each backup. If the primary crashes, 2-safe guarantees that the backups have all transactions committed by the primary. However, in order to ensure consistency (i.e., at most one primary at a time), 2-safe techniques rely on accurate failure detection. This approach has a latency of $2\delta$, and injects $2(n-1)$ and $n$ messages in the network for cases in which communication is by point-to-point and low-level broadcast, respectively.

Table 1 compares Pronto with OPT-broadcast and Paxos, 1-safe, and 2-safe. This comparison is only done for reference purposes since the 1-safe and 2-safe approaches make different assumptions about the underlying system and ensure different guarantees than Pronto with OPT-broadcast and Paxos. Pronto equipped with OPT-broadcast and optimized Paxos has the same latency as 2-safe. As for the number of messages, in point-to-point networks OPT-broadcast is $\mathcal{O}(n^2)$ while the other approaches are $\mathcal{O}(n)$. The message complexity of OPT-broadcast could be reduced to $\mathcal{O}(n)$ at the expense of an increase of one $\delta$ unit in the latency. If the network provides a low-level broadcast primitive, OPT-broadcast and optimized Paxos inject about the same number of messages as 2-safe. We notice however that false failure detection may lead to inconsistencies with 2-safe. With Pronto, false failure suspicions may prevent transactions from committing temporally, but consistency is always ensured.

| **Protocol / Approach** | **Latency** | **Point-to-point messages** | **Broadcast messages** | **Remarks** |
|---|---|---|---|---|
| OPT-broadcast | 2 $\delta$ | $(n+1)(n-1)$ | $n+1$ | optimistic order |
| Paxos | 4 $\delta$ | 5 $(n-1)$ | 2 $(n+1)$ | general case |
| Paxos (optimized) | 2 $\delta$ | 3 $(n-1)$ | $n+1$ | adapted to Pronto |
| 1-safe | 0 | $n-1$ | 1 | lost transactions |
| 2-safe | 2 $\delta$ | 2 $(n-1)$ | $n$ | accurate timeouts |

Table 1: Cost of terminating a transaction

## 4.2 Pronto Prototype

We built a simple prototype to assess the overhead introduced by Pronto relative to a non-replicated configuration. Figure 3 depicts the architecture of our Pronto prototype. Continuous lines between clients and the servers represent point-to-point communication; for simplicity we do not draw them for all clients. Dashed lines between servers represent broadcast communication.

In the prototype, every server has several execution threads and one communication thread. Each execution thread can execute one transaction at a time against the local database (i.e., the JDBC model). The communication thread implements atomic broadcast. At the backups, only one execution thread is active; the others simply wait for the server to become primary. Transactions are generated locally at the primary server.

Each execution thread has a JDBC connection to the local database, through which it sends SQL requests. After sending a request to the database, the thread waits for its response. If the request is a commit operation, before forwarding it to the database, the thread sends all SQL requests to the other servers, starting the broadcast execution for this transaction. The broadcast execution is then continued by the communication thread in the primary and in the backups.

Atomic broadcast is implemented based on the optimized Paxos protocol [33]. All messages are exchanged using TCP/IP. In the absence of failures and failure suspicions, a broadcast results in a message from the primary to all backups, an acknowledge message from each backup to the primary, and a final message from the primary to the backups. The primary can deliver the transaction after it receives an acknowledgment from a majority of servers (including itself). Backups deliver the transaction upon receipt of the second message from the primary. In case

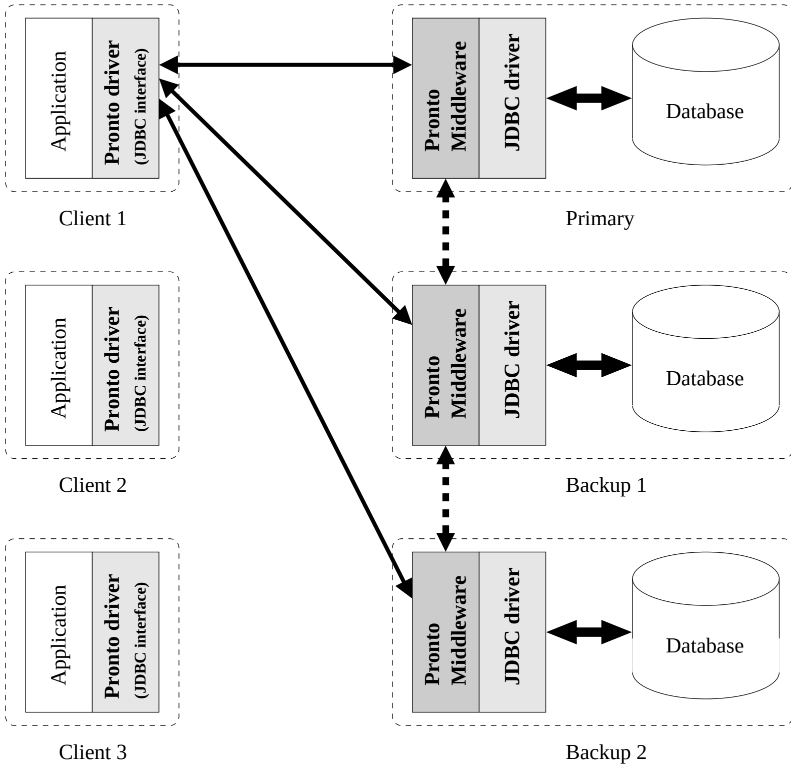



Figure 3: Pronto architecture

of failures and failure suspicions, more messages are exchanged to ensure that whatever delivery order is chosen by the primary is always respected by any future primary [33]. We assume that there is always a majority of servers up, and do not execute a synchronous logging as part of the broadcast [5].

After the execution thread broadcasts a transaction, it waits for a local event corresponding to its delivery. This event is raised by the communication thread once the transaction is delivered. The execution thread reacts to the delivery event by certifying the transaction. If the transaction passes the certification test, the execution thread sends a commit operation to the database; otherwise, it sends an abort operation.

Concurrent transactions never have their order reversed due to non-determinism of thread execution. To see why, consider threads $T_1$ and $T_2$ executing two transactions, $t_a$ and $t_b$, concurrently. If $t_a$ and $t_b$ conflict, there is some data item they both try to access. The first transaction that succeeds to hold a lock on the data, say $t_a$, proceeds but the second one, $t_b$, will be blocked by the database. Therefore, $T_2$ will not have a chance to broadcast $t_b$ until $t_b$ can access the requested data item, and $t_b$ will only access the data item after $t_a$ releases its locks. Transaction $t_a$ will release its locks after it is delivered, and thus, it cannot be that $t_b$ will be committed before $t_a$.

### 4.3 The Experiments' Setup

For the experiments, we used three database servers connected through a 10 MBit Ethernet network. The Primary executes in a 550 MHz Pentium III CPU with 256 MByte RAM; one of the backups (hereafter, Backup 1) executes in a 600 MHz Pentium III CPU with 128 MByte RAM; the second backup (hereafter, Backup 2) executes in a 200 MHz Pentium Pro CPU with 128 MByte RAM. The Primary and Backup 2 run Windows NT 4.0, and Backup 1 runs Windows 2000 Professional. All servers run a commercial database, installed without any tuning. Communication with the database is through JDBC 2.0.

The database has 6 similar tables, `account0`, `account1`, ..., `account5`, with 10000 records each. A record is composed of 6 fields: `acct_num` (10 bytes), `name` (10 bytes), `branch_id` (1 byte), `balance` (10 bytes), and `temp` (10 bytes). Each table is indexed by the `acct_num` field.

We have considered only update transactions in the experiments since these are the ones that generate interactions between servers, and can slow down the failover mechanism. JDBC has API's that allow applications to distinguish queries from update transactions. Thus, the primary can use these API's to process queries only locally. Each transaction has between 1 and 6 write operations. The number of operations in a transaction, and the tables and entries they update are randomly generated following a uniform distribution. A typical SQL operation looks like '`update account0 set balance = 1000.00 where acct_num = 1234`'.

### 4.4 The Performance of Pronto

In our experiments, we varied the multiprogramming level (MPL), that is, the number of execution threads in the primary and the think time (THT), that is, the time an execution thread in the primary waits after it finishes one transaction and before it starts another one. Once a threads starts a transaction, it executes the transaction as fast as it can. The think time controls the rate of transactions executed by a single thread.

To build the graphs presented next, we conducted a series of experiments, each one with about 40000 transactions. The executions were broken up into segments of about 20 seconds. At the end of each segment, a sample of values of interest was taken (e.g., throughput, response time) and at the end of the execution, we calculated their mean value. To eliminate initial transients, the values sampled in the first segment were discarded. All mean values are presented with a confidence interval of 95%.

In each experiment we measured two aspects of system performance:

- *Response time.* The time it takes to execute a single transaction end-to-end at the Primary. The start time is measured immediately before initiating the transaction; the end time is measured immediately after committing the transaction. The response time is then the difference between the end time and start time.

- *Throughput.* The transactions per second that a server process commits. As for response time, the throughput is measured by the Primary. As previously described, the execution is divided into segments. The throughput of the execution is calculated as the average of the throughput of each segment.

### 4.4.1 The Cost of Replication

The graphs in Figures 4–7 show the overhead of running Pronto as compared to a non-replicated, single machine system. We quantify the overhead in terms of both response time and throughput.

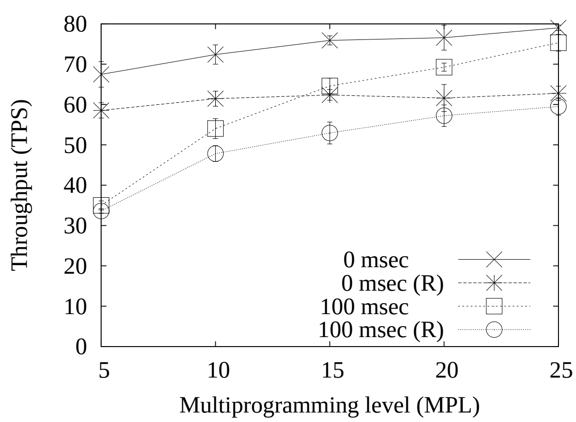



Figure 4: Ensemble throughput

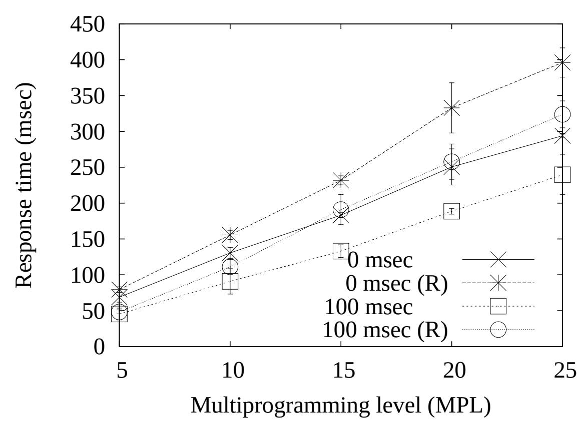
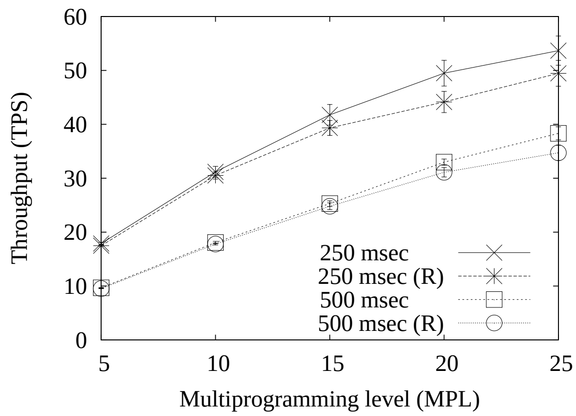



Figure 5: Ensemble response time

The graphs in Figures 4 and 6 show the relationship between throughput and multi-programming level for various think times. For each think time, we show a throughput graph for a single database system and a replicated database system. We can then determine the overhead



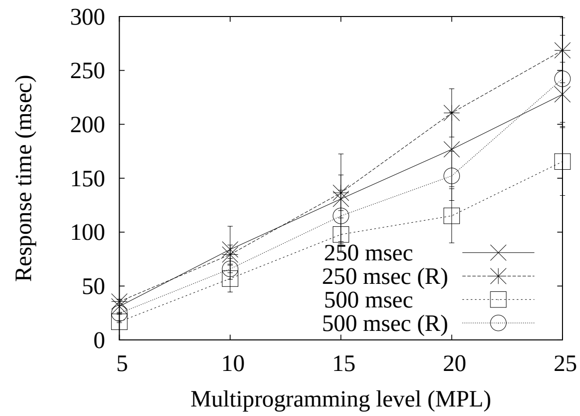




Figure 6: Ensemble throughput (cont.)

Figure 7: Ensemble response time (cont.)

of Pronto by comparing the graph for a single database system to the graph for a replicated database system for a given think time. For example, for a think time of 0 (see Figure 4) running Pronto results in a throughput decrease of about 13% when the multi-programming level is 5 and 20% when the multi-programming level is 25. Moreover, a think time of 0 illustrates the worst-case scenario for Pronto. In this scenario, there is maximal contention for network resources. If instead we consider a think time of 500 msec (Figure 6), the overhead for any think time is less than 10%.

The graphs in Figures 5 and 7 quantify the response-time overhead of running Pronto. As for the throughput measurements, a response time graph shows the relationship between response time and multi-programming level for a given think time. Again, the worst-case overhead occurs with a think time of 0 and the maximum multi-programming level considered (i.e., 25). In this particular case, the response time overhead is about 26%. As can be seen from the response-time graphs, the higher the multi-programming level, the bigger the overhead of running Pronto. As the multi-programming level increases, so does the contention for network resources.

#### 4.4.2 Response Time Break-Down

In Figures 8 and 9 we show a break-down of the response time measurements—we show where the elapsed time was actually spent. For the break-down, we use three categories: (a) *Exec time*, the elapsed time for executing SQL statements through the JDBC interface, (b) *Abcast time*, the elapsed time (as seen by the primary server) of executing Atomic Broadcast, and (c) *Commit* *time*, the elapsed time to execute the transaction commit operation by the primary server.

Figures 8 and 9 show a response-time break-down for a think time of 0 and 500 msec, respectively. Both figures show the break down as “stacked” graphs, with Abcast time stacked on top of Exec time, and Commit time stacked on top of Abcast time. This means that a data point on the graph labeled Exec time is the actual measured value for Exec time. In contrast, a data point on the graph labeled Abcast time is the sum of the measured Abcast time and the measured Exec time. Furthermore, a data point on a graph labeled Commit time is the sum of the measured Commit time, the measured Abcast time, and the measured Exec time. Thus, the commit-time graph actually shows the end-to-end response time. The distance between the graphs in a given figure then denotes the actual break-down into categories.

As can be seen from the graphs, the time spent executing Atomic Broadcast increases as the multi-programming level increases. As indicated above, this effect explains why the cost of Pronto increases as the multi-programming level increases. Moreover, the system spends more time executing Atomic Broadcast with a think time of 0 as it does with a think time of 500 msec: We have more contention for network resources with a think time of 0 than we do with a think time of 500 msec.

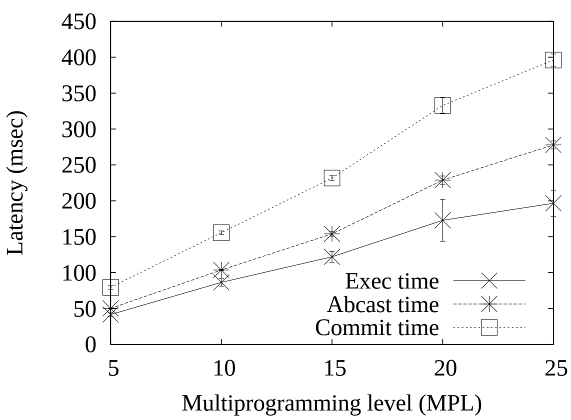



Figure 8: Resp.time break-down (THT=0)

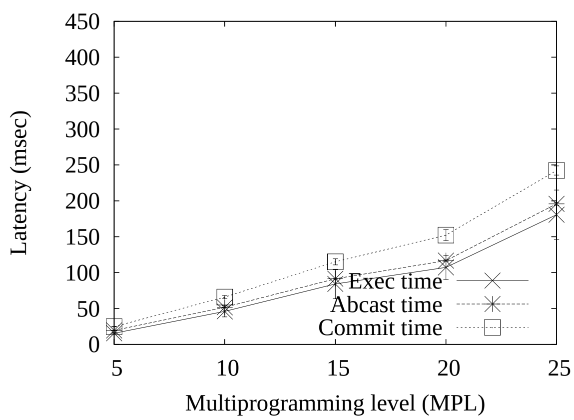



Figure 9: Resp.time break-down (THT=500)

#### 4.4.3 Keeping Up with the Primary

Figures 10 and 11 show the relationship between the throughput in the primary, under various multi-programming levels and think times, and the throughput in the backups, with multi-programming level 1 and no think time. A ratio smaller than or equal to 1 means that the backup can process transactions faster than or as fast as the primary; a ratio greater than 1 means that the backup is slower than the primary. If the backup cannot process transactions as fast as the primary, transactions will have to be queued. If the workload is constant, the ratio gives a measure of the growth rate of the backup queue.

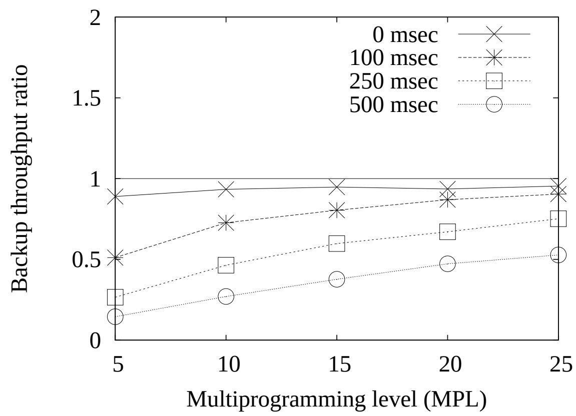



Figure 10: Throughput ratio (Backup 1)

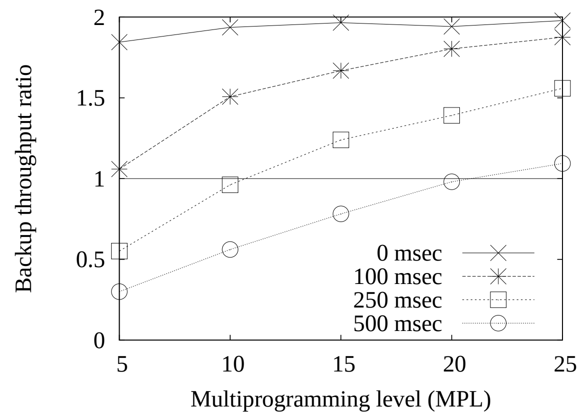



Figure 11: Throughput ratio (Backup 2)

From Figure 10, Backup 1 never lags behind the primary (i.e., for every multi-programming level and think time, the throughput ratio in Backup 1 is smaller than 1). The same does not hold for Backup 2. Figure 11 shows that Backup 2 cannot process transactions as fast as the primary when the think time is 0 or 100 msec. Nevertheless, Backup 2 can keep up with the primary when the think time is 250 msec and the multi-programming level below 10, and the think time is 500 msec and the multi-programming level below 20.

# 5 Related Work

Database replication has been the subject of much research in the past years and many protocols have been proposed. Gray et al. [23] have classified database replication into *eager* and *lazy*. Eager replication, the category Pronto falls into, ensures strong consistency. Lazy replication increases performance by allowing replicas to diverge, possibly exposing clients to inconsistent states of the database. Eager replication can be based on the *update everywhere* or on the *primary-backup* approach [23]. Update everywhere allows any copy of the database to be updated. Primary-backup assigns a single database, the primary, to process transactions and checkpoint its state to one or more backups. The backups will simply reproduce the primary's state by locally applying the shipped state. Usually, the actual information shipped to the backups is not the new database state but the transaction log [24].

Early work on eager update everywhere has been based on the read-one/write-all approach [9] and on quorum systems (e.g., [1, 2, 22, 35, 53]). More recently, several works have proposed implementing eager update everywhere replication using an underlying total order broadcast abstraction (e.g., [4, 14, 25, 41, 55]). In general, these protocols capture different trade-offs between scalability (in terms of throughput) of the replicated database system, underlying network assumptions, generality of the transaction model, and use of standard or custom databases. For example, some protocols have been optimized for wide-area networks [6, 48], while others exploit special local-area network properties [36]. The approach in [28] is to modify the underlying databases so that all write locks for a transaction can be acquired in a single atomic step. Compared to these existing approaches, Pronto uses standard databases and supports interactive transactions, whose structure is determined dynamically (i.e., the result of a previous query may determine which requests to execute subsequently).

In [40] the authors discuss *conflict classes* for synchronizing update transactions. A conflict class represents a partition of the database (e.g., one class per table). Replicas commit transactions in the same conflict class in the same order; total order broadcast is used to ensure that transactions belonging to more than one conflict class will execute in the same order. Transactions execute on one site and only their updates are processed by remote replicas, similarly to Pronto. Differently from [40], however, Pronto does not need any partitioning information.

Amza et al. [13] introduces a database replication protocol that guarantees strong consistency (one-copy serializability), typical of eager replication, and scaling properties typical of lazy replication. A scheduler is interposed between database clients and the replicas. Transactions are scheduled according to their access patterns. Multi-query transactions must declare what tables they read or write before they start the execution.

Pacitti et al. [38] present a replication protocol that supports update everywhere and partial replication. The consistency is guaranteed by annotating transactions with a chronological timestamp value and ensuring FIFO order among messages send by a replica. The transaction is allowed to execute on a database server only when the upper bound of the time needed to multicast a message has exceeded. Further optimizations of the protocol allow concurrent transactions execution at the replicas and reduction of delays.

Many works have considered alternative consistency criteria for database replication, such as Snapshot Isolation [8], and its variations for replicated settings [34, 50]. Kemme et al. discuss in [27] how to implement both serializability and snapshot isolation using group communication; all protocols presented require modifications to the database engine. Ganymed [43], Postgres-R(SI) [56], and Tashkent [18] are database replication protocols based on snapshot isolation.

Primary-backup replication can be configured as *1-safe* or *2-safe* [19, 24]. With a 1-safe configuration, the primary commits the transaction locally and then sends the transaction log to the backups. With a 2-safe configuration, the primary ships the transaction log to the backups and an atomic commit is used to ensure that either all databases will commit the transaction or none of them will. While 1-safe may result in lost transactions if the primary fails after committing the transaction locally and before sending the transaction log to the backups [24], 2-safe prevents the primary from committing a transaction before it is safely received at the backups [44] or the primary reliably detects the failure of the backups. Comparatively, Pronto prevents lost transactions and allows incorrect failure suspicions.

Disaster recovery in a system consisting of a group of primary sites executing distributed transactions is discussed in [21]. Backup sites duplicate the database stored in the primary sites. Each primary site sends its transaction log to its corresponding backup site (or sites, if more than one backup exists). For performance reasons, the algorithm used is 1-safe. Even though update transactions are required to execute at primary sites, overall system performance can be improved by executing read-only queries at backup sites [45]. The authors have concluded that although a 1-safe mechanism reduces processing and communication significantly, a 2-safe approach may be preferable if transactions cannot be lost or if communication and processing are fast and lock contention is not serious [44].

Lazy replication usually leads to weak consistency models [46, 30] and has to deal with issues not related to Pronto replication model, such as data *freshness*, i.e., how often the primary should propagate updates to the backups and how often the backups should process these updates [39]. For example, in [58] a real-time primary-backup replication scheme which enforces temporal consistency among replicated servers is considered. Although the work does not address databases explicitly, one could possibly apply the ideas in the context of databases. To detect failures, the primary and the backups permanently exchange ping messages. After the detection of a failure, the backup simply starts a backup version of the application and uploads the current state information. The authors have found that "in general, the total detection and recovery time is dominated by the detection time."

Database replication has also been implemented using epidemic techniques. The idea is that updates pass through the system like an infectious disease, from site to site [17]. Although the approach is particularly suited for weak consistency [47], some epidemic protocols have been augmented to ensure serializability [3]. Ensuring serializability with lazy propagation has also been achieved by restricting the placement of replicas [10, 16] and building global replication graphs [11].

Commercial products (e.g., [37, 57]) have traditionally favored failover based on storage systems that are shared among the cluster nodes (e.g., a disk array), whereas Pronto does not assume a shared storage system among the replicas. Besides, lazy replication has been provided by virtually every major commercial database, usually targeting at on-line analytical processing [52].

# 6 Conclusion

This paper presents 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 uninterrupted 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. Performance evaluation conducted with a prototype in an environment composed of off-the-shelf databases has shown that Pronto can synchronize the database ensemble with a reasonable performance overhead.

# Acknowledgments

The authors would like to thank the anonymous reviewers for their comments on this work.

## References


[1] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, 1989.

[2] D. Agrawal and A. El Abbadi. The tree quorum protocol: an efficient approach for managing replicated data. *16th Int. Conference on Very Large Databases*, pages 243–254, 1990.

[3] D. Agrawal, A. El Abbadi, and R. Steinke. Epidemic algorithms in replicated databases. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson (USA), May 1997.

[4] 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.

[5] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the International Symposium on Distributed Computing (DISC'98)*, pages 231–245, September 1998.

[6] Y. Amir and C. Tutu. From total order to database replication. In *International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

[7] P.A. Barrett, A.M. Hilborne, P.G. Bond, D.T. Seaton, P. Verissimo, L. Rodriguez, and N.A. Speirs. The Delta-4 extra performance architecture (XPA). In *Proceedings of 20th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1990.

[8] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2):1–10, June 1995.

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

[10] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data: SIGMOD '99*, volume 28(2), pages 97–108, 1999.

[11] Y. Breitbart and H. F. Korth. Replication and consistency: being lazy helps sometimes. In *Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 173–184, New York (USA), 1997.

[12] 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.

[13] W. Zwaenepoel C. Amza, A. Cox. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In M. Endler and D. C. Schmidt, editors, *Middleware 2003, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003, Proceedings*, volume 2672 of *Lecture Notes in Computer Science*, pages 282–304. Springer, 2003.

[14] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proceedings of USENIX Annual Technical Conference, Freenix track*, 2004.

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

[16] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proc. of the 12th Int. Conference on Data Engineering*, pages 469–476, February 1996.

[17] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of the 6th ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver (Canada), August 1987.

[18] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. In *Proceedings of EuroSys*, 2006.

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

[20] S. Frølund and R. Guerraoui. A pragmatic implementation of e-Transactions. In *Proceedings of the 21th IEEE Symposium on Reliable Distributed Systems (SRDS)*, October 2000.

[21] H. Garcia-Molina, C. A. Polyzois, and Robert B. Hagmann. Two epoch algorithms for disaster recovery. In *16th International Conference on Very Large Data Bases*, pages 222–230, Brisbane, Queensland, Australia, 13–16 August 1990. Morgan Kaufmann.

[22] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating System Principles SOSP 7*, pages 150–162, Pacific Grove (USA), December 1979.

[23] 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.

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

[25] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group multicast. In *Proceedings of International Symposium on Fault Tolerant Computing (FTCS29)*, pages 158–165. IEEE Computer Society, 1999.

[26] R. Jiménez-Peris, M. Pati no Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 150–159, 2002.

[27] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB'2000)*, pages 134–143, Cairo, Egypt, September 2000.

[28] B. Kemme and G. Alonso. A new approach ro developing and implementing eager database replication protocols. *ACM Transactions on Database Systems (TODS)*, 25(3), September 2000.

[29] B. Kemme, A. Bartoli, and Ö. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the Internationnal Conference on Dependable Systems and Networks (DSN2001)*, Göteborg, Sweden, June 2001.

[30] N. Krishnakumar and A. J. Bernstein. Bounded ignorance in replicated systems. In *Proc. of the Tenth ACM SIGACT-SIGMOD-SOGART Symposium on Principles of Database Systems*, volume 51(2), pages 63–74, 1991.

[31] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[32] L. Lamport. The part-time parliament. Technical Report 49, DEC Systems Research Center, 1989. Also published in ACM Transactions on Computer Systems (TOCS), Vol. 16, No. 2, 1998.

[33] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[34] Y. Lin, B. Kemme, M. Patino-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of data*, 2005.

[35] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[36] M. Pati no Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Disctributed Computing (DISC)*, 2000.

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

[38] E. Pacitti, C. Coulon, P. Valduriez, and M. Tamer Özsu. Preventive replication in a database cluster. *Distributed and Parallel Databases*, 18(3):223–251, nov 2005.

[39] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal: Very Large Data Bases*, 8(3–4):305–318, February 2000.

[40] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Middle-r: Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems*, 2005.

[41] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proc. of the 16th IEEE Symposium on Reliable Distributed Systems*, Durham (USA), October 1997.

[42] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

[43] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, 2004.

[44] C. A. Polyzois and H. Garcia-Molina. Evaluation of remote backup algorithms for transaction-processing systems. *ACM Transactions on Database Systems*, 19(3):423–449, September 1994.

[45] C. A. Polyzois and H. Garcia-Molina. Processing of read only queries at a remote backup. In *Symposium on Reliable Distributed Systems (SRDS '94)*, pages 192–201, October 1994.

[46] C. Pu and A. Leff. Replica control in distributed systems: an asynchronous approach. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(2):377–386, June 1991.

[47] M. Rabinovich, N. H. Gehani, and A. Kononov. Scalable update propagation in epidemic replicated databases. In *5th International Conference on Extending Database Technology (EDBT'96)*, volume 1057 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 1996.

[48] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P Vicente. Strong replication in the GLOBDATA middleware. In *Workshop on Dependable Middleware-Based Systems*, 2002.

[49] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems. In *20th Int. Conference on Distributed Computing Systems (ICDCS '00)*, pages 288–297, April 2000.

[50] F. Pedone S. Elnikety and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'2005)*, 2005.

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

[52] D. Stacey. Replication: DB2, Oracle, or Sybase? *SIGMOD Record*, 24(5):95–101, December 1995.

[53] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, 4(2):180–209, June 1979.

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

[55] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, April 2000.

[56] S. Wu and B. Kemme. Postgres-R(SI):combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the IEEE International Conference on Data Engineering*, 2005.

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

[58] H. Zou and F. Jahanian. A real-time primary-backup replication service. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):533–548, June 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 $\Sigma_i^e$ 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 $\Sigma_i^e$. 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** *For all epochs $e \geq 1$ and every two servers $s_i$ and $s_j$ that execute epoch $e$ to completion, $C(\mathcal{E}_i^e) = C(\mathcal{E}_j^e)$.*

PROOF (SKETCH): We show that database servers $s_i$ and $s_j$ commit the same transactions (i.e., $\Sigma_i^e = \Sigma_j^e$) in the same order (i.e., $<_i^e = <_j^e$). Let $s_{p(e)}$ be the primary in epoch $e$.

To prove that $\Sigma_i^e = \Sigma_j^e$, assume $s_i$ commits transaction $t$ in epoch $e$. Therefore, $t$ passed the validation test, and so, $s_i$ delivered message $(s_{p(e)}, e, -, t, \textit{\textbf{sqlSeq}}(t))$ before delivering any message of the type ($e$, "NEW EPOCH"). From the agreement and total order properties of the broadcast primitive, $s_j$ also delivers $(s_{p(e)}, e, -, t, \textit{\textbf{sqlSeq}}(t))$ before delivering any message of the type ($e$, "NEW EPOCH"). Thus, $t$ passes the validation test at $s_j$ and is committed by $s_j$.

We now show that $<_i^e = <_j^e$. Assume $t_a$ precedes $t_b$ at $s_i$. We prove that the same holds at $s_j$. (Notice that from the definition of serializability [9], one transaction can only precede another if they execute conflicting operations.) There are three cases to be considered:

- *Case 1. $s_i$ and $s_j$ are backups.* Then it follows directly from the total order property of the broadcast primitive and the fact that transactions are executed sequentially according to the delivery order that $s_i$ and $s_j$ will commit $t_a$ and $t_b$ in the same order.

- *Case 2. $s_i$ is the primary (i.e., $s_i = s_{p(e)}$) and $s_j$ is a backup.* Since $s_i$ executes transactions using a 2PL scheduler, if $t_a$ precedes $t_b$, then $t_a$ commits before $t_b$ at $s_i$. From the algorithm, $t_a$ is delivered before $t_b$. By total order of the broadcast primitive and the fact that backups commit transactions in the same order as they are delivered, $s_j$ delivers and commits $t_a$ before $t_b$. Thus, $t_a$ precedes $t_b$ at $s_j$.

- *Case 3. $s_i$ is a backup and $s_j$ is the primary.* Since $t_a$ precedes $t_b$ at $s_i$, (a) $t_a$ must have been delivered before $t_b$ at $s_i$ (from the way transactions are executed at the backups) and (b) both transactions execute conflicting operations (from the definition of serializability). From (a), it must be that $t_a$ is delivered before $t_b$ at $s_j$. Thus, $t_a$ was broadcast by $s_j$. Since both transactions execute conflicting operations (from (b)) and $s_j$ uses a 2PL scheduler, it follows from the fact that $t_b$ cannot reach the committing state until $t_a$ commits and releases all its locks. Therefore, $t_a$ precedes $t_b$ at $s_j$. $\square$

**Lemma 2** *For each epoch $e \geq 1$, if conflicting transactions $t_a$ and $t_b$ are committed by server $s_i$ in $e$, then their execution has been serialized in $e$.*

Proof (sketch): Since both $t_a$ and $t_b$ commit in $e$, from the validation test, they were both executed at $s_{p(e)}$, the primary in epoch $e$. Since the primary uses a local 2PL scheduler, it follows that $t_a$ and $t_b$ have been serialized in $e$. $\square$

**Property 1** ***(RDB-1)*** *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): By Lemma 1, for all $e \geq 1$ and every two servers $s_i$ and $s_j$ that execute $e$ to completion, $C(\mathcal{E}_i^e) = C(\mathcal{E}_j^e)$. From Lemma 2, we have that for every execution $\mathcal{E}_i^e$ produced by $s_i$ at epoch $e$, there is some sequential execution $\mathcal{E}_s^e$ such that $C(\mathcal{E}_i^e)$ 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 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 $C(\mathcal{E}_s)$. $\square$

**Property 2** ***(RDB-2)*** *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 10 and 18. 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 10, the client has requested a `begin`$(-)$ operation, and a database is always able to execute it; in the *wait* statement at line 18, 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, a client could not execute job $j$ if (a) it cannot find the primary, or (b) it 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 valid 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. $s_p$ eventually becomes primary and remains primary forever. So, the client will eventually contact $s_p$, start a transaction on $s_p$, and never suspect $s_p$. Finally, alternative (c) contradicts database property DB-2: since $s_p$ does not crash, $c$ does not suspect it and keeps sending transactions generated by executions of $j_a$. So, eventually, one transaction terminates and $j_a$ is successfully executed. We conclude that $c$ eventually contacts a primary that executes and commits a transaction generated by $j_a$. $\square$

## Biographies

Fernando Pedone received his MS degree in computer science from Universidade Federal do Rio Grande do Sul (UFRGS), Brazil, in 1995, and his Ph.D. degree in computer science from the Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland, in 1999. He worked as a researcher at the Hewlett-Packard Laboratories in Palo Alto, California, and as a senior researcher at EPFL. He is currently an Associate Professor at the Università della Svizzera italiana (USI), Switzerland. His professional interests include the theory and practice of distributed systems and distributed data management systems.

Svend Frølund received his MS degree in computer science from Århus University in Denmark in 1990, and his Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1994. He worked as a researcher, and as a senior researcher, at the Hewlett-Packard Laboratories in Palo Alto, California, for ten years, and currently works as a senior software architect at Gatehouse A/S in Denmark. His professional interests include object-oriented programming, distributed systems, and software architecture.