Continental Pronto

Svend Frolund, Fernando Pedone
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-166 (R.1)
July 19th , 2001*

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

data replication, disaster recovery, high availability

Continental Pronto unifies high availability and disaster resilience at the specification and implementation levels. At the specification level, Continental Pronto formalizes the client's view of a system addressing local-area and wide-area data replication within a single framework. At the implementation level, Continental Pronto makes data highly available and disaster resilient. The algorithm provides disaster resilience with a cost similar to traditional 1-safe and 2-safe algorithms and provides highly-available data with a cost similar to algorithms tailored for that purpose.

To be published in the 20th IEEE Symposium on Reliable Distributed Systems, 2001.

Continental Pronto[*]

Svend Frølund Fernando Pedone
Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA
{Svend_Frolund, Fernando_Pedone}@hp.com

Abstract

Continental Pronto unifies high availability and disaster resilience at the specification and implementation levels. At the specification level, Continental Pronto formalizes the client's view of a system addressing local-area and wide-area data replication within a single framework. At the implementation level, Continental Pronto makes data highly available and disaster resilient. The algorithm provides disaster resilience with a cost similar to traditional 1-safe and 2-safe algorithms and provides highly-available data with a cost similar to algorithms tailored for that purpose.

1 Introduction

Increasingly, online databases must be continuously available. To remain available in the presence of disasters, such as earthquakes and floods, critical online databases typically run in multiple, geographically dispersed data centers. Each data center contains a complete copy of the database, and these copies operate in a primary-backup manner: clients are connected to a single primary data center, and the other data centers are in stand-by mode waiting to take over if the primary suffers a disaster. In some cases, clients can also connect to backup data centers, but only to request queries. Data centers are connected via wide-area networks, and because of severe consequences (e.g., client re-connections), the take-over process—when a backup data center becomes the new primary data center—usually involves human operators. The backup data centers may use timeouts to detect disasters in the primary, but the actual fail-over requires operator approval.

Current data centers consist of clusters of servers, with servers within a cluster connected through a local-area network. This local-area replication (within a data center) aims to increase both the scalability and availability of the database. In terms of availability, the local-area replication enables the database to survive non-disaster failures without activating a backup data center: another replica within the primary data center can take over in case of non-disaster failures, such as process crashes, disk crashes, machine crashes, and so on. Thus database availability involves both local-area replication within a single data center, for non-disaster failures, and wide-area replication across data centers, for disaster recovery.

[*]© IEEE. This is an extended version of a paper, with the same title, that appears in the proceedings of the 20th (2001) IEEE Symposium on Reliable Distributed Systems.

Local-area replication and wide-area replication have traditionally been addressed by separate mechanisms. Local-area replication is typically achieved through a parallel database system, such as Oracle OPS [17], that runs in a high-availability cluster. In contrast, wide-area replication can be classified as either 1-safe or 2-safe [10], and typically involves shipping the transaction log from a primary site to a backup site. However, combining conventional local-area with wide-area mechanisms is not trivial. For example, assume that a local-area replication mechanism replicates a data item x in two databases d and d', both in the same data center. We do not want both d and d' to propagate the same updates to x to the backup data center. On the other hand, we want to propagate each update to x in a fault-tolerant manner—we do not want an update to be lost if either d or d' goes down.

Continental Pronto provides both local-area and wide-area replication in an integrated manner. Continental Pronto combines the classical notions of 1-safe and 2-safe with a local-area replication protocol based on transaction shipping. While running in a single data center only, Continental Pronto provides local-area replication with a cost similar to protocols that provide local-area replication only [18]. While running in a configuration with 2 data centers (or more), each with a single database, Continental Pronto behaves like either a classical 1-safe or a classical 2-safe disaster-recovery protocol—the choice of 1-safe and 2-safe is configurable.

One of the features of Continental Pronto is that we use transaction shipping for both local-area and wide-area replication. Using the same basic mechanism for both types of replication allows us to provide a combined approach. For example, we use the replication of databases within a data center to ensure that the disaster-recovery protocol itself can tolerate local failures. Residing above the database also allows us to cope with heterogeneous systems because we rely on a standard interface only (e.g., JDBC). Finally, by using transactions as the unit of replication and shipping, we have access to a higher level of semantic information about data updates as compared to transaction log mechanisms.

To reason about the correctness of Continental Pronto, we have formalized the notion of disaster recovery. Our formalism captures the client's view of the system, including both local-area and wide-area concerns in a single framework. Local-area and wide-area replication fundamentally solve the same problem: making data, and thereby transaction processing, highly available. Rather than treating local-area and wide-area replication as separate issues, with separate correctness conditions, we define a single correctness specification that captures highly-available transaction processing. This single specification reflects correctness according to external entities (clients) that access and manipulate highly-available data.

The rest of the paper is structured as follows. Section 2 introduces the system model and some terminology. Section 3 discusses Continental Pronto properties. Section 4 presents Continental Pronto in detail. Section 5 assess the performance of Continental Pronto, Section 7 discusses related work, and Section 6 concludes the paper. Correctness proofs are in the Appendix.

2 System Model and Definitions

2.1 Processes and Groups

We consider a system composed of two disjoint sets of processes, the *Clients* set and the *Databases* set. To capture the notion of data center, we sub-divide the *Databases* set into subsets, $G_1, \ldots, G_n$. We refer to each subset G_x as a group of databases (or simply a group), $G_x = \{d_1, d_2, \ldots, d_{n_x}\}$. We assume that there are "logical" communication links connecting clients and databases, and databases among themselves. In practice, several logical links can be multiplexed over the same physical link. We assume that links connecting databases within the same group transmit messages more efficiently than links connecting databases across groups. Figure 1 depicts a typical system that exemplifies our model. Each group is internally connected through a local-area network. Different groups communicate via wide-area networks. These wide-area network links can either be leased lines or part of the public Internet. Clients connect to the groups via the public Internet.

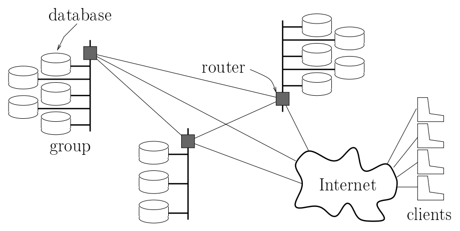

Figure 1: System model

Clients communicate with databases by message passing and can establish a connection with any database. Databases communicate with each other using the Hierarchical Atomic Broadcast abstraction defined in Section 2.3. We do not make assumptions about message-delivery times nor the time it takes for a process to execute individual steps of its local algorithm.

Processes (clients and databases) can fail by crashing, that is, when a process fails, it permanently stops executing its algorithm—we do not consider Byzantine failures. A *correct* process is one that does not fail. Database processes may also recover after a crash, but for simplicity we do not introduce database recovery explicitly in the model—we address database recovery in Section 4.3. A disaster is an event that makes a group permanently unable to perform its intended function. We define the notion of disaster based on the aggregate failure of databases: a group G_x suffers a disaster if a certain number $k_x, 0 < k_x \leq n_x$, of databases in G_x have failed.[1] We say that a group is *operational* if it does not suffer a disaster; a group that suffers a disaster is *in-operational.* In this paper, we assume that at least one group is operational.

[1]Parameter k_x is not necessarily equal to n_x; in some cases, the failure of a majority of databases in a group may prevent the remaining ones in the same group from performing their intended function—in such a case, $k_x = \lceil (n_x + 1)/2 \rceil$.

2.2 Failure and Disaster Detectors

We equip the system with failure detectors and disaster detectors. A failure detector $\mathcal{D}_x$ gives information about the possible crash of databases in a group G_x to databases in G_x and to the clients. In general, if the failure detector of process p returns a set that includes a database d, we say that p *suspects* d. We assume that eventually, every database in G_x that crashes is permanently suspected by every correct database in G_x and by every correct client; and that if group G_x contains a correct database, then there is a time after which this database is never suspected by any correct database in G_x or by any correct client (i.e., $\mathcal{D}_x$ belongs to the class of eventually strong failure detectors [5]).

Disaster detectors are defined using similar machinery as failure detectors. If a disaster detector $\mathcal{DD}$ returns a set of groups including group G_x to a process p, we say that p suspects G_x to be in-operational. We assume that every group G_x that contains fewer than k_x correct databases is eventually permanently suspected by every correct process. Moreover, no group G_x is suspected by any database if it contains k_x or more correct databases. That is, each database has access to a disaster detector that is perfect in the sense of [5]. In contrast, clients have access to a weaker disaster detector $\mathcal{DD}_c$: eventually, no group G_x is suspected by any client if it contains k_x or more correct databases. That is, clients have access to a disaster detector that is eventually perfect in the sense of [5].

Our decision to give each database access to a perfect disaster detector reflects our assumptions about Continental Pronto's execution environment. Continental Pronto executes in a number of data centers—modeled as groups—and these data centers are configured in a primary-backup fashion. Appointing a new data center as the primary data center is an expensive operation that usually is only initiated in response to disasters. Because data center fail-over is expensive, the decision to initiate such an operation is usually made by a human operator. We refer to this kind of fail-over as "push-button" fail-over since there is a human in the loop. With push-button fail-over, the detection of disasters is more accurate since operators in different data centers can potentially verify disaster suspicions. Moreover, it is possible to enforce the appearance of a disaster, for example by shutting down computers manually, before initiating a fail-over operation.

Clients have access to an eventually perfect disaster detector. This reflects our assumption that the network connection between clients and the primary data center may undergo instability periods, during which clients may incorrectly suspect the primary data center to be in-operational, but will eventually stabilize for long enough to ensure that "useful" computation can be done.

2.3 Hierarchical Atomic Broadcast

In the following, we define Hierarchical Atomic Broadcast (HABcast), the communication abstraction used by databases to communicate in Continental Pronto. HABcast uses the notions of failure detectors and disaster detectors and trades communication within groups for communication across groups.

HABcast is defined by the primitives Broadcast(m), Deliver(1-SAFE, m), and Deliver(2-SAFE, m). If a correct database in an operational group executes Broadcast(m), it eventually executes Deliver(1-SAFE, m) and Deliver(2-SAFE, m). HABcast also ensures the following properties—we assume next that *sender*(m) is the database that executes Broadcast(m), and *group*(m) is *sender*(m)'s group.

- HB-1: If a database in an operational group executes Deliver(1-SAFE, m), then every correct database in each operational group eventually executes Deliver(1-SAFE, m).
- HB-2: If a database in *group*(m) executes Deliver(2-SAFE, m), then every correct database in each operational group eventually executes Deliver(2-SAFE, m).[2]

In the absence of disasters, both properties HB-1 and HB-2 ensure that messages are delivered by every correct database in each operational group. In the presence of disasters the properties differ. If a database executes Deliver(1-SAFE, m) and *group*(m) suffers a disaster, there is no guarantee that correct databases in other operational groups will also execute Deliver(1-SAFE, m). However, that is not the case if a database in *group*(m) executes Deliver(2-SAFE, m) and then *group*(m) suffers a disaster: every correct database in each operational group will also execute Deliver(2-SAFE, m). To ensure property HB-2, databases have to exchange messages across groups, and, as a result, Deliver(1-SAFE, m) can be implemented more efficiently than Deliver(2-SAFE, m)—we revisit this issue in Section 5.

Moreover, HABcast guarantees that:

- HB-3: If two databases execute Deliver(1-SAFE, m) and Deliver(1-SAFE, *m'*), then they do so in the same order.
- HB-4: No database executes Deliver(2-SAFE, m) before executing Deliver(1-SAFE, m).

Property HB-3 states that messages of the type Deliver(1-SAFE, –) are globally ordered, and property HB-4 specifies a constraint on the order in which messages are locally delivered.

HABcast properties define the permissible event sequence for broadcast and delivery events. To illustrate the semantics of HABcast, Figure 2 shows an event sequence that satisfies the HABcast properties, and Figure 3 shows an event sequence that does not. In Figure 2, the database d_1^x broadcasts a message m, and every correct database in groups G_x and G_y delivers (1-SAFE,m) and (2-SAFE,m). Furthermore, database d_1^x broadcasts a message m' and delivers (1-SAFE,m'), but since G_x suffers a disaster, databases in G_y do not necessarily have to deliver (1-SAFE,m'). Figure 3 depicts an execution that does not satisfy the HABcast specification since database d_3^x is in *group*(m), delivers message (2-SAFE,m), and no database in G_y delivers (2-SAFE,m).

2.4 Databases and Transactions

Database processes implement a number of primitive operations. These primitives capture the behavior of commercial database systems, as accessed through standard APIs, such as JDBC. A transaction is started with the `begin` primitive, and terminated with either the `commit` or the `abort` primitives. While a transaction is active, we can use the `exec` primitive to execute SQL statements within the transaction. We assume that all the primitives are non-blocking: if we call a primitive on a database and the database does not crash, the primitive will eventually return.

Besides the primitives, we also make the following assumptions about every database d_i:

[2]Notice that HB-2 is not the uniform counterpart of HB-1, in the sense of [12]—which is "If a database executes Deliver(2-SAFE, m), then every correct database in each operational group eventually executes Deliver(2-SAFE, m)." HABcast takes advantage of the "asymmetry" in HB-2 to reduce the number of messages exchanged between groups.

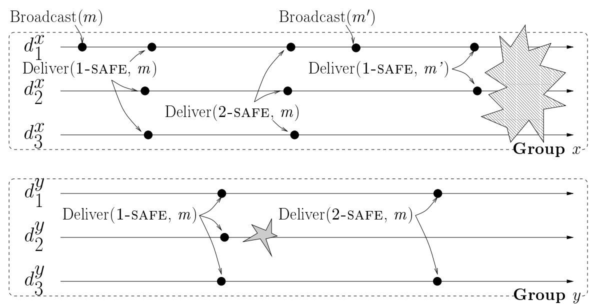

Figure 2: Correct execution of HABcast

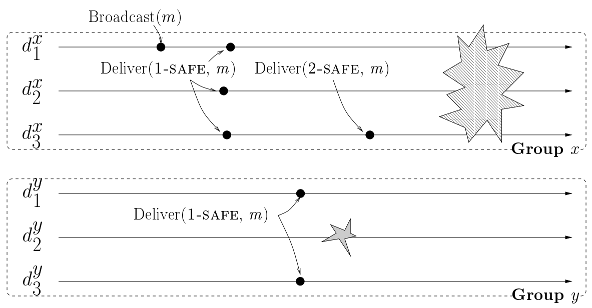

Figure 3: Incorrect execution of HABcast

- DB-1: Transactions are serialized by d_i using strict two-phase locking (strict 2PL).
- DB-2: If d_i is correct, there is a time after which d_i commits every transaction that it executes.

The first property, as known as serializability, ensures that 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 $\mathcal{E}$ [3]. Furthermore, strict-2PL schedulers ensure that if transactions t_a and t_b *conflict*[3] and t_a has been committed before t_b by d_i in some execution $\mathcal{E}$, then t_a precedes t_b in any serial execution $\mathcal{E}_s$ that is equivalent to $\mathcal{E}$ [3].

The second property, although not explicitly stated as such, is often assumed to be satisfied by database systems. The property 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—of course, this property assumes that transactions do not request an abort operation.

3 Problem Specification

Continental Pronto provides data replication across both local-area and wide-area networks. Here, we outline properties that characterize what it means for a data replication protocol such as Continental Pronto to be correct.

The first property we consider is concerned with data consistency in a replicated database system. For data consistency, we use the conventional notion of 1-copy serializability [3]:

- CP-1: Every execution consisting of all committed transactions in a group is 1-copy serializable.

Property CP-1 is a safety condition. It requires the replication algorithm to provide the illusion that there is only a single copy of each data item. One can trivially satisfy the 1-copy serializability requirement with a system that never commits any transactions, or with a system that only has a single copy of each data item. Even though these solutions guarantee 1-copy serializability, the former does not do any useful computation, and the latter does not provide high availability or disaster resilience. To rule out unsatisfactory solutions, we need additional properties.

[3]Two transactions conflict if they both access the same data item and at least one of the operations modifies it.

To capture data availability and prevent trivial solutions, we introduce a liveness property that Continental Pronto should satisfy. The liveness property is concerned with clients, and represents the idea that a client should eventually receive a reply from the replicated database system, even if individual databases fail or groups of databases become in-operational.

To state the liveness property, we first introduce the notion of a *job.* A job is the transactional logic that a client executes to manipulate the replicated data in the system. To handle failures and disasters, this logic may execute multiple physical transactions. For example, the logic may start a transaction against one database, and, if that database fails, retry the same logic against another database, giving rise to another physical transaction. We say that a client *submits* a job when it starts to execute the job's logic, which may encompass generic retry logic and transaction-specific SQL statements. We say that a client *delivers* a job when the job execution is complete. Delivering a job captures successful completion: a physical transaction has committed in some database, and the client has the result of the transaction (e.g., a confirmation number for a hotel reservation). With these concepts in place, we can now state the liveness property for our protocol:

- CP-2: If a client submits a job j, and does not crash, then it will eventually deliver j.

Of course, we cannot require a system to unconditionally provide such a liveness guarantee: if all databases crash, the system can obviously not provide an answer to the client. We state the property as an absolute requirement. Particular implementations will then provide this guarantee under certain assumptions. For example, our protocol will provide the guarantee if a single group of databases remains operational.

Property CP-2 ensures that a transaction will eventually be committed in some database. However, if that database fails, the transaction may be lost. If the client delivers one job and then submits another job, property CP-2 does not ensure that the updates performed by the first job are visible to the second job. The issue of lost transactions is concerned with the durability of transactional updates. To consider it correct, we need the system to satisfy some level of durability that ensures visibility of transactional updates across database crashes and group disasters. Informally, what we want to ensure is that if a client successfully updates the state of a particular database (i.e., delivers a job), then those updates are visible in all databases that the client may subsequently connect or fail over to. Based on the conventional concepts of 1-safe and 2-safe disaster recovery [10], we identify two levels of durability: 1-safe durability and 2-safe durability. These durability levels generalize the conventional 1-safe and 2-safe characterization to a system where we rely on local replication for high availability and wide-area replication for disaster recovery.

We formulate 1-safe durability as follows:

- CP-3: If a database in an operational group commits a transaction t, then all correct databases in all operational groups commit t.

The conventional flavor of 1-safe disaster recovery is to asynchronously propagate updates to a backup site—the primary site commits transactions without waiting for the backup site to do the same. Rather than capturing the propagation scheme, we capture the consequence of using the scheme as seen by clients. With a 1-safe scheme, transactions can be lost: the primary may commit a transaction and then fail before sending it to the backup. Our 1-safe durability has the same spirit: it only ensures global transaction commitment if an operational group commits the transaction (an operational group in our scheme corresponds to a correct primary in conventional 1-safe protocols).

With 1-safe durability, delivery of a job by a client does not imply global commitment of the transactional update. Delivery of the job implies that some database committed the transaction, but if that database is in a group that suffers a disaster, the 1-safe durability property imposes no obligation on other database to also commit the transactional update. In contrast, 2-safe durability imposes such an obligation:

- CP-4: If a client delivers a job j then all correct databases in all operational groups commit the transactional updates performed by j.

With 2-safe durability, there are no lost transactions. If a client delivers a job against one database, and then fails over to another database in a different group, the job's updates are also present in the second database. This is similar in spirit to conventional 2-safe protocols where the primary site does not respond to the client unless it has received acknowledgement from the backup site that the updates are committed. Continental Pronto is configurable to provide either 1-safe durability or 2-safe durability.

Finally, to provide global consistency, we require that databases in different groups commit conflicting transactions in the same order. Property CP-1 makes sure that this holds for databases in the same group, but it does not prevent the case where a database d_i in a group commits a transaction t before a conflicting transaction t' and another database d_j, in a distinct group, commits t' before t, as long as both d_i and d_j are consistent. Property CP-5 handles this case.

- CP-5: If two databases commit conflicting transactions t and t', they do so in the same order.

4 Continental Pronto

4.1 Overview of the Protocol

Continental Pronto is based on the primary-backup replication model: a single database is appointed as global primary, and all other databases are backups. Clients connect to the primary database to submit update transactions; read-only transactions, or queries, can be executed against any database, be it in the same group as the primary or not.

Normal operation. In the absence of failures, disasters, and suspicions, the protocol works as follows—we consider next only update transactions; queries are simply executed locally at any database and do not require any distributed synchronization among databases. To submit transactions, clients first have to find the current primary database, which they do by polling databases. If the first database contacted turns out to be the current primary, it establishes a connection with the client; otherwise it returns to the client the identity (i.e., group id and database id) of the database it believes to be the current primary.

Assuming that the system eventually stabilizes, that is, failures, disasters, and suspicions do not keep happening indefinitely, clients eventually find the current primary database.

The primary receives SQL requests from the clients and executes them concurrently, but under the constraints of (local) strict two-phase locking. Because the execution at the primary is serializable, it is equivalent to some serial execution of the same transactions, and the order in which commit requests are submitted to the primary defines one such serial execution order. To see why, consider two concurrent transactions t_1 and t_2. If t_1 has its commit requested before t_2, either (a) t_1 and t_2 do not conflict, and their relative order does not matter (i.e., t_1 can be ordered before t_2), or (b) t_1 and t_2 conflict, and the locking mechanism built-in in the primary will block t_2 until after t_1 releases its locks, and so, t_2 will have its commit requested after t_1, in which case t_2 is ordered after t_1 in any serial order equivalent to the real execution.

When the primary receives a commit request for a transaction from some client, the primary uses HABcast to broadcast the SQL statements for the transaction to all the backups. Upon 1-SAFE delivering such a message, each backup executes the transaction against its database, following the delivery order. Update transactions at the backups are executed sequentially, according to the order they are delivered. Since this delivered order corresponds to the serializable order in the primary, all databases order conflicting update transactions in the same order. Thus, the primary database can be non-deterministic and execute transactions concurrently since we can repeat the same non-deterministic choices at the backups. Even though backups have to process update transactions sequentially, they can do it concurrently with the execution of local read-only transactions. Notice that performance at the primary is not hurt by the backups because the primary can reply back to the client right after delivering a 1-SAFE or 2-SAFE message (depending on the level of durability required by the transaction) and receiving a reply from its local database acknowledging the commit of the transaction.

Failures, disasters, and suspicions. The Continental Pronto model of primary-backup replication allows us to use standard, off-the-shelf, databases without modification. However, this replication model introduces the challenge of how to maintain consistency without assuming a perfect failure-detection scheme. Remember that we assume a failure-detection scheme where databases within a group have access to a very weak failure detector which is allowed to make an arbitrary number of mistakes. Across groups, we have a disaster-detection scheme that is perfect (i.e., does not suspect disasters unless they have actually happened). The disaster-detection scheme reflects the push-button fail-over scheme between data centers.

Continental Pronto implements primary-backup on top of these failure and disaster detection mechanisms. We use failure detection to elect a new primary within the same group when the current primary is suspected, and we use disaster detection to elect a new primary in a different group when the current primary's group has suffered a disaster. In both cases, we use the HABcast abstraction, introduced in Section 2.3, to ensure that all databases agree on the sequence of primaries; but the databases only agree on the sequence of primaries, not on the actual real time at which a database is appointed primary. This looser notion of agreement allows us to implement a primary-backup mechanism without assuming a synchronous model and without making timing assumptions within groups.

Due to the asynchrony of message transmissions, however, more than one primary in the same group may co-exist during certain periods of time. To handle situations of multiple primaries executing transactions concurrently, we rely on a certification scheme similar to the one used in the Pronto protocol [18]. With such a scheme, the execution evolves as a sequence of *epochs*. All databases start their execution in the first epoch, and for any given epoch, there exists a pre-assigned primary database. Whenever a database suspects the current primary to have crashed, it uses HABcast to request an epoch change, and, consequently, a change in the primary. Every message broadcast carrying a transaction, a failure suspicion, or a disaster suspicion also contains the epoch in which the message was broadcast. Upon delivering a message, the action taken depends on its epoch.

- A transaction delivered in the epoch in which it was broadcast (and thus, executed) is committed; a transaction delivered in a different epoch than the one in which it was broadcast is aborted.
- A suspicion delivered in the same epoch in which it was broadcast makes the database pass to the next epoch; a suspicion delivered in a later epoch than the one in which it was broadcast is ignored.

Since all databases deliver messages in the same order, they all agree on which transactions should be committed and which ones should be aborted. A client that has its transaction aborted because it used an outdated primary re-executes its transactional using the current primary.

4.2 The Protocol in Detail

Algorithm 1 depicts the database side of Continental Pronto. We have omitted pseudo code for the client-side algorithm in Continental Pronto, as it is quite similar to the client-side algorithm in Pronto [18].

The first part of the algorithm (lines 1–28) shows the transaction processing by the primary database. The start of a transaction is embodied in a `begin` request. In response to such a request, the primary initializes some control variable for the transaction. The primary tracks the durability level of the transaction (the variable `level`), a unique identification of the job from which the transaction originates (the variable `jobid`), and the state of the transaction (the variable `state`). 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 databases by the primary. A transaction delivered by a database is in the committing state, and it remains in the committing state until its fate is known by the database (i.e., commit or abort). The executing and committing states are transitory states, whereas the committed and aborted states are final states.

The second part of the algorithm (lines 29–50) shows the delivery of transactions. Upon delivering a committing transaction, each database server certifies the transaction using the control information provided by the primary. The certification test is deterministic, ensuring that all database servers reach the same result (i.e., commit or abort) individually. If a backup 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. If the primary does not crash and its group is operational, clients eventually receive the transaction's outcome. If the outcome is abort, the client re-executes the transactional job.

Finally, the third part of the algorithm (line 51–66) shows the fail-over logic—the behavior that triggers and controls the election of primaries. The execution evolves as a sequence of epochs: whenever a backup suspects the primary or its group, it starts a new epoch, by broadcasting an epoch change message. If a backup process in the primary group suspects the primary process, some other database in the primary group becomes the next primary. If a backup process suspects the primary group, another group assumes the role of primary group, and some database in this group becomes primary. Primaries are deterministically computed from the epoch number. If a backup suspects the primary incorrectly, the primary also delivers the change epoch message and aborts all transactions in execution. In such a case, clients have to re-start the execution of their jobs in the new primary, as described before. Notice that given the properties of disaster detectors, backups never incorrectly suspect the primary group.

Algorithm 1 Database d_i^x in group G_x

```
1:  **Initialization...**

2:    $e_i \leftarrow 1$                                   {$e_i$ is the current epoch at $d_i^x$}
3:    $prmy\_grp_i \leftarrow 1$                           {$prmy\_grp_i$ is the current primary group and...}
4:    $prmy\_db_i \leftarrow 1$                            {...$prmy\_db_i$ the current primary according to $d_i^x$}

5:  **To execute a transaction...**

6:  **when** receive ($t_a$, request) from $c$ **do**
7:    **case** request = begin($job\_id$, $durability\_level$, $t_a$):
8:      **if** $d_i^x \neq prmy\_db_i$ **or** $G_x \neq prmy\_grp_i$ **then**      {if not the primary...}
9:        send ($t_a$, "I'M NOT PRIMARY") to $client(t_a)$          {...refuse to process the request, ...}
10:     **else**                                                   {...else...}
11:       $client(t_a) \leftarrow c$                    {...keep the id of the client associated with $t_a$, ...}
12:       $job(t_a) \leftarrow job\_id$                            {...$t_a$'s job identification, and...}
13:       $level(t_a) \leftarrow durability\_level$                {...its durability level}
14:       $state(t_a) \leftarrow$ EXECUTING                        {update $t_a$'s state, and...}
15:       begin($t_a$)                                   {...start $t_a$'s execution against the database}
16:       **wait for** response($t_a$, $result$)                {wait for a reply from the database, and...}
17:       send ($t_a$, $result$) to $client(t_a)$               {...send the results to the client}
18:   **case** (request = exec($t_a$, sql-req) **or** request = abort($t_a$)) **and**...
          ...$state(t_a)$ = EXECUTING:
19:     **exec task**                                              {execute SQL statements...}
20:       exec($t_a$, sql-req)                                     {...concurrently with other requests}
21:       **wait for** response($t_a$, $result$)                 {wait for a reply from the database}
22:       **if** $result$ = ABORTED **then**                       {if the database decided to abort $t_a$...}
23:         $state(t_a) \leftarrow$ ABORTED                        {...update $t_a$'s state}
24:       send ($t_a$, $result$) to $client(t_a)$              {in any case, send results to the client}
25:   **case** request = commit($t_a$) **and** $state(t_a)$ = EXECUTING:
26:     $state(t_a) \leftarrow$ COMMITTING                         {update $t_a$'s state...}
27:     sql-seq $\leftarrow$ all exec($t_a$, sql-req) statements in order     {...gather SQL statements for $t_a$, and...}
28:     Broadcast($d_i^x$, $G_x$, $e_i$, $t_a$, $client(t_a)$, $job(t_a)$, $level(t_a)$, sql-seq)     {...broadcast $t_a$ to all}
```

Due to the unreliable nature of the failure detectors used by the backups within the primary group, it is possible that at a given time during the execution, database processes execute in different epochs, and so, multiple primaries may be able to execute transactions simultaneously. To prevent database inconsistencies (i.e., non-serializable executions) that may arise from such situations, transactions have to pass a validation test before committing. 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 if the epoch in which the database delivers the transaction and the epoch in which the transaction was executed are the same.

4.3 Dealing with Database Recovery

Continental Pronto was formally designed in the crash-stop model—in which databases never recover after a crash. Even though not realistic in many practical circumstances, we decided to consider this model to simplify the formal treatment of Continental Pronto, and then extend it in an *ad hoc* manner to re-integrate recovering databases. Two aspects have to be addressed to allow databases to recover: HABcast and Continental Pronto, the protocol itself. HABcast can be given a crash-recover semantics along the lines of [2, 22], which extend Consensus and Atomic Broadcast to the crash-recover model.

In the case of Continental Pronto, besides announcing its presence to other databases, a recovering database also has to catch up with those other databases. One way to do it is for a recovering database d to deliver all messages that have been delivered by other databases while d was down. Notice that this is already captured by the HABcast properties which require correct databases to deliver the same transactions in the same order—correct in the crash-recover model does not rule out databases that crash and recover, but those that crash and never recover, or keep crashing and recovering without performing any useful computation.

5 Performance Assessment

In Continental Pronto, all database communication relies on HABcast (defined in Section 2.3). To reply to a client's commit request, the primary first has to broadcast the commit request and deliver it. Thus, the performance of the broadcast abstraction has a major impact on the performance of Continental Pronto. In the following, we discuss how to implement HABcast. Then we compare the performance of Continental Pronto to the performance of traditional 1-safe and 2-safe protocols. Our comparison is done analytically (i.e., based on the number of messages and the latency) as well as by simulation.

5.1 Implementing HABcast

There are many ways to implement HABcast. Our discussion here is based on the implementation in [7]. Thus, HABcast is implemented as a composition of uniform atomic broadcast protocols running independently of each other in each data center. As for Continental Pronto, there is a primary process, and the group to which the primary belongs is denoted the primary group. During times when processes do not crash, groups do not suffer disasters, and there are no suspicions, the protocol works as follows (see Figure 4). To broadcast a message m, a process in the primary group first executes a local atomic broadcast within its group. When a process in the primary group local delivers m, it delivers message

Algorithm 1 (cont.) Database d_i^x in group G_x

```
29: To commit a transaction...

30: when Deliver(1-SAFE, $d_j^y, G_y, e_j, t_a, client(t_a), job(t_a), level(t_a)$, sql-seq) do
31:   if $\exists t_b \neq t_a$, s.t. $state(t_b) =$ COMMITTED and $job(t_b) = job(t_a)$ then   {if committed a txn for the same job...}
32:     if $level(t_a) =$ 1-SAFE then send (–, COMMITTED) to $client(t_a)$   {...send reply to the client}
33:   else
34:     if $e_j < e_i$ then   {if $t_a$ didn't execute in the current epoch...}
35:       execute abort($t_a$)   {...$t_a$ has to be aborted}
36:       wait for response($t_a, result$)   {...ditto}
37:       $state(t_a) \leftarrow$ ABORTED   {update $t_a$'s state}
38:     else
39:       if $d_i^x \neq prmy\_db_i$ or $G_x \neq prmy\_grp_i$ then   {if $t_a$ didn't execute at $d_i^x$ in $G_x$...}
40:         for each ($t_a$, sql-req) in sql-seq do   {...submit the whole txn to the database}
41:           execute sql-req   {$t_a$ can be committed now...}
42:           wait for response($t_a, result$)   {...done!}
43:       execute commit($t_a$)   {request $t_a$'s commit, and...}
44:       wait for response($t_a, result$)   {...wait for confirmation}
45:       $state(t_a) \leftarrow$ COMMITTED   {update $t_a$'s state}
46:     if ($state(t_a) =$ ABORTED or $level(t_a) =$ 1-SAFE) and...
          ...$d_i^x = d_j^y$ and $G_x = G_y$ then   {if aborted or not a 2-SAFE txn...}
47:       send ($t_a, state(t_a)$) to $client(t_a)$   {...send status to the client}

48: when Deliver(2-SAFE, $d_j^y, G_y, e_j, t_a, client(t_a), job(t_a), level(t_a)$, sql-seq) do
49:   if $\exists t_b$ s.t. ($state(t_b) =$ COMMITTED and $job(t_b) = job(t_a)$) and...
        ...$level(t_a) =$ 2-SAFE and $d_i^x = d_j^y$ and $G_x = G_y$ then   {if $t_a$ or other txn for the same job committed...}
50:     send ($t_a$, COMMITTED) to $client(t_a)$   {...send reply to the client}

51: To request a primary server/group change...

52: when $prmy\_db_i \in \mathcal{D}_i$ and $G_x = prmy\_grp_i$ do   {if suspect primary and belong to its group...}
53:   Broadcast($e_i$, "CHANGE SERVER")   {...request a primary change}

54: when $prmy\_grp_i \in \mathcal{DD}$ do   {if suspect the primary group...}
55:   Broadcast($prmy\_grp_i$, "CHANGE GROUP")   {...request a primary group change}

56: To change a primary server/group...

57: when (Deliver(1-SAFE, $e_j$, "CHANGE SERVER") and $e_j = e_i$) or...
      ...(Deliver(1-SAFE, $prmy\_grp_j$, "CHANGE GROUP") and $prmy\_grp_j = prmy\_grp_i$) do
58:   if $prmy\_db_i = d_i^x$ and $G_x = prmy\_grp_i$ then   {if incorrect primary suspicion...}
59:     for each $t_a$ s.t. $state(t_a) =$ EXECUTING do   {...abort all txns in execution}
60:       execute abort($t_a$)   {request $t_a$'s abort, and...}
61:       wait for response($t_a, result$)   {...wait for confirmation}
62:       $state(t_a) \leftarrow$ ABORTED   {update $t_a$'s state}
63:       send ($t_a$, ABORTED) to $client(t_a)$   {notify $t_a$'s client}
64:   $e_i \leftarrow e_i + 1$   {change the current epoch}
65:   if Delivered (1-SAFE, "CHANGE GROUP") then $prmy\_grp_i \leftarrow prmy\_grp_i + 1$   {determine new primary group}
66:   $prmy\_db_i \leftarrow e_i$ mod sizeof(group $prmy\_grp_i$)   {determine the current primary}
```

(1-SAFE, m). Upon locally delivering m, the primary process p_i also sends m to a single process in each other group. When a process p_j in group G_y receives m, p_j executes a local atomic broadcast with m. Upon locally delivering m, p_j sends a reply to p_i and delivers messages (1-SAFE, m) and (2-SAFE, m). After p_i receives a reply for m from some process in each operational group it does not suspect to have suffered a disaster, p_i delivers message (2-SAFE, m) and sends a message to the other members of the primary group so that they can also deliver message (2-SAFE, m).

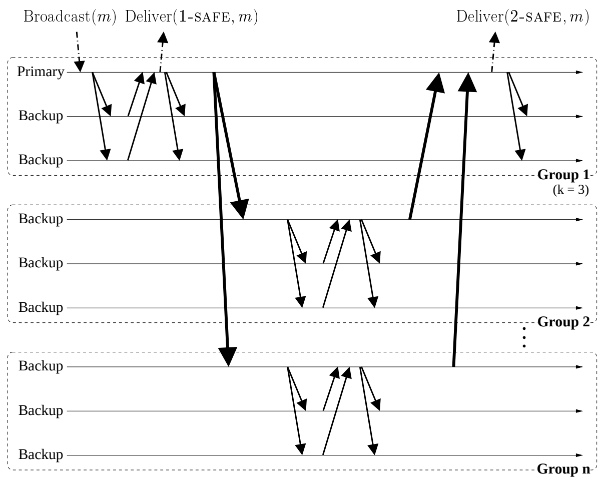

Figure 4: Failure-free execution of the HABcast protocol

Dealing with failures and suspicions in HABcast boils down to determining another primary process whenever the current crashes or is suspected, and taking over the execution from where the previous one left it. This is a particularly complicated operation in HABcast since a process taking over the role of primary has to take actions that are not "incompatible" with the ones taken by the previous primary, such as determining the delivery order of messages. A detailed discussion about how HABcast handles failures, suspicions, and disasters is out of the scope of this paper, and can be found in [7].

5.2 Analytical Evaluation

We compare Continental Pronto to two algorithms that inherently deal with data center disasters: 1-safe and 2-safe [10]. These algorithms are the conventional means to achieve database disaster recovery. The 1-safe and 2-safe algorithms deal with a system where each data center has a single database only, and although [10] 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 is similar to Continental Pronto's 1-safe durability in case of data center disasters. 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. This is what Continental Pronto guarantees with 2-safe durability in case of data center disasters.

Our comparison assumes best case scenarios, without failures and suspicions. This means, for example, that when considering Continental Pronto, we assume that the primary process in HABcast coincides with the primary database process in Continental Pronto. We use number of messages and latency as metrics for our comparison. For the latency analysis, we distinguish between δ_l, the transmission delay along local-area links, and δ_w, the transmission delay along wide-area network links; we assume $\delta_w > \delta_l$. We also specify the number of messages injected into the network per message broadcast, distinguishing between messages injected into a local-area network and messages injected into a wide-area network. If a process in some data center sends a message to a process in another data center, we count that communication as a single wide-area message and no local-area messages. If a process sends a message to another process in the same data center, we count a single local-area message only. We assume, as a simplification, that all n groups have the same number k of processes.

Table 1 presents the results of our comparison. The 1-safe protocol only involves a single wide-area message to each backup and no latency because the primary does not wait for the backups to commit. The latency for Continental Pronto is based on the latency of HABcast when delivering a message of the type (1-SAFE, $-$), which is determined by the latency of the local Atomic Broadcast within the primary group. Using the Atomic Broadcast algorithm presented in [5] with some optimizations [23], this latency is $2\,\delta_l$. Although the protocol does not wait for the backups to deliver messages, the primary data center still communicates with all the backup data centers (asynchronously). The primary in the primary data center sends a round-trip wide-area message to a single process in each backup data center. This communication pattern amounts to $2(n-1)$ wide-area messages. Each data center executes a local Atomic Broadcast protocol, which requires $3(k-1)$ local-area messages. Moreover, there are n such executions, giving a total of $3n(k-1)$ local-area messages. In addition, the primary in the primary data center executes a reliable broadcast, which amounts to $k-1$ local-area messages. All in all, running Continental Pronto in 1-safe configuration gives rise to $(k-1)(3n+1)$ local-area messages.

The latency for a 2-safe protocol is $2\,\delta_w$ because the primary synchronously communicates with the backups. A conventional 2-safe protocol gives rise to wide-area messages only—there is no notion of local-area replication in a conventional 2-safe protocol. If we run Continental Pronto in 2-safe mode, its latency is based on the latency of HABcast when delivering a message of the type (2-SAFE, $-$). As illustrated in Figure 4, this latency is composed of a local Atomic Broadcast in the Primary group, a local Atomic Broadcast in each backup group (these occur concurrently), and a round-trip communication with each backup group (these are also concurrent). Thus, the total latency for Continental Pronto in 2-safe mode is $4\,\delta_l + 2\,\delta_w$. The number of messages is the same for Continental Pronto in 1-safe and 2-safe mode—only the latency is different.

Protocol	**Latency**	**Wide-area msgs**	**Local-area msgs**
1-safe	0	$(n-1)$	0
C Pronto 1-safe	$2\,\delta_l$	$2(n-1)$	$(k-1)(3n+1)$
2-safe	$2\,\delta_w$	$2(n-1)$	0
C Pronto 2-safe	$4\,\delta_l + 2\,\delta_w$	$2(n-1)$	$(k-1)(3n+1)$

Table 1: Cost of protocols

5.3 Simulation-Based Evaluation

Our analytical evaluation of HABcast does not consider local messages used within the groups by the failure detection mechanism, and the impact of having to share common resources, such as communication links, on the latency of the protocol. In order to take these factors into account, we have built a simulation model and conducted several experiments. Our simulation model considers n groups of processes, and each group has its own local-area network. Groups communicate with each other using dedicated links, however only one link is used between any two groups. Transmission of wide-area messages also impacts the transmission of a local-area message in the sender's group and in the receiver's group, to model the local communication with the routers in each group. For local-area messages, we assume a transmission latency randomly generated between 2 and 3 milliseconds, and for wide-area messages between 100 and 150 milliseconds. Messages are all broadcast by the same process at maximum rate, that is, some process in the primary group broadcasts a message right after delivering $(2\text{-SAFE}, -)$ for the previous message.

Figures 5 and 6 depict some of the results of our experiments. In both cases, enough experiments were conducted to build confidence intervals of 98%. The confidence intervals are not shown in the graphs since they never overlap. Figure 5 compares the times to deliver messages of the type $(1\text{-SAFE}, -)$ and $(2\text{-SAFE}, -)$ in a system with 3 groups. Not surprisingly, most of the overhead to deliver a message of type $(2\text{-SAFE}, -)$ is related to wide-area messages.

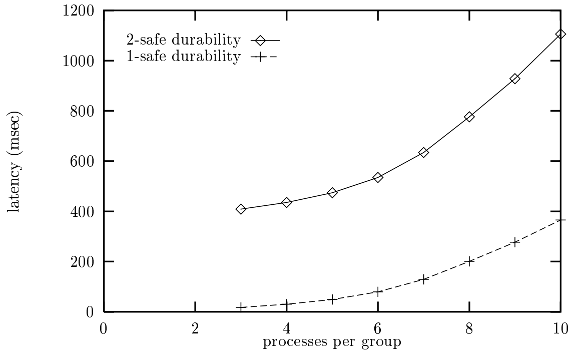

Figure 5: 1-safe and 2-safe durability

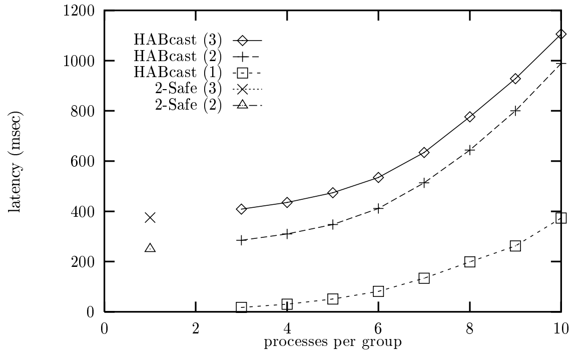

Figure 6: Comparing HABcast to 2-safe

Figure 6 compares the time to deliver messages of the type $(2\text{-SAFE}, -)$ with the time of the 2-safe algorithm, for configurations with 2 and 3 groups. The first observation is that for groups with 3 processes, for systems with 2 and 3 groups, there is no big difference between Continental Pronto and the 2-safe algorithm. The wide-area latency for Continental Pronto in 2-safe mode and for a traditional 2-safe protocol is the same, and this wide-area latency is the main component of the total latency. In terms of resilience, however, one process crash in the 2-safe algorithm requires a data center failover, while in Continental Pronto this requires a local reconfiguration.

6 Related Work

The basic issues in disaster recovery and the notion of 1-safe and 2-safe protocols are not new [15, 8, 10]. A large body of research and several commercial systems, such as [21, 25], address these issues. We identify next some of the main trends distinguishing wide-area replication from local-area replication.

6.1 Wide-Area Replication

Most works consider disaster recovery in the context of a primary site and a number of backup sites. Activating a backup site to be a new primary site is usually done with human intervention (i.e., push-button fail over). A key feature distinguishing these works is whether a site contains a single database or whether it contains multiple databases that each contain a partition of the data.

Single-database sites. The algorithms in [16] are based on log shipping: the transaction log at the primary is sent to the backups. The algorithms implement the basic 1-safe semantics, where the main idea is to parallelize the processing of log entries at backup sites to reduce the resource consumption at those sites. The notion of 0-safe introduced in [6] also addresses the scenario with a single database per site. Unlike traditional 1-safe algorithms, the 0-safe approach allows multiple copies of the database to be updated simultaneously (i.e., the system can have multiple primaries). As in 1-safe log-propagation schemes, when a database is updated, the log entries are asynchronously sent to all other copies. The basic assumption behind the 0-safe scheme is that all transactions commute. Although transactions may be lost as in the traditional 1-safe case, the 0-safe algorithm allows a recovered databases to later on inform the other databases of transactions that may have been lost (without violating consistency).

Multiple-database sites. In [19], multiple primaries databases in the primary site, each with its own disjoint partition of the database, separately send their logs to a backup site. Each primary database has a backup peer database, and the issue is to provide "consistent" 1-safe semantics in the context of independent streams of log entries without violating atomicity: a disaster at the primary could result in a transaction that is only partially reflected at the backup. The paper introduces two algorithms to prevent such atomicity violations, which tag information onto the log entries being sent to the backups. The backup site only commits a transaction if it has all its pieces, and if it has all the preceding transactions. The algorithms in [14] address the same scenario, but as an extension to [19], they allow the same site to contain both primary and backup partitions. The challenge with such mixed sites is to ensure that any given site can recover to a consistent state after possibly aborting some in-progress transactions. The basic idea in [14] is extend the locking period at a primary database so that locks are not released until a transaction is stable in the corresponding backup database. Thus, the system has 1-safe characteristics from the client perspective, but the throughput may be more like a 2-safe approach. Finally, [13] considers the case of a partitioned database, but from a 2-safe viewpoint. The main idea is to exploit the lazy commit optimization described in [10] to release the locks held at the primary site before sending the log records to the backup site. Thus, unlike traditional 2-safe protocols, locks will not be held at the primary while the transaction logs are sent to the backup. This allows for increased transaction throughput at the primary. However, in terms of end-to-end response time (as seen by the client), the performance will be similar to traditional 2-safe protocols: to avoid lost transactions, the primary cannot reply to the client until it receives an acknowledgement message from the backup site that the transaction is durable.

There are several differences between these existing approaches and Continental Pronto. First, Continental Pronto relies on transaction shipping rather than log shipping. This means that we can support heterogeneous databases as long as they support a standard interface, such as JDBC—of course, the price for this flexibility is a degradation in the performance relative to log shipment. Second, we can deploy Continental Pronto without modification of the database internals—we only rely on the standard database semantics. Third, Continental Pronto provides disaster resilience for systems where the primary and the backup sites contain multiple copies of the same data item. That is, the failure of a single database can be handled locally, within a single data center. Only when a data center actually suffers a disaster is it necessary to bring another data center online. More importantly, we provide this localized handling of non-disaster failures without increasing the cost of the disaster-recovery protocol. In other words, the local replication of data within a data center does not give rise to "replication" of the transaction sent across the wire to backup sites: each transaction is only sent once.

6.2 Local-Area Replication

In terms of commercial systems, the traditional way to provide local-area replication is through a parallel database system, such as OPS [17] or XPS [27]. These systems run in clusters [26], where each cluster node contains an instance of the database. With a shared-disk approach, used by OPS, the instances access the same disk (each cluster node mounts the disk). With a shared-nothing approach, used by XPS, the instances access their own disk, but can mount each others disk if an instance fails. In both cases, the disk sharing requires special hardware.

A number of systems provide local-area replication without special hardware. These systems can be divided in two groups: active replication and primary backup [9, 11]. With active replication [24], 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 total-order guarantees (i.e., Atomic Broadcast). Database replication techniques usually assume a certain knowledge about the semantics of the operations, or require stronger deterministic assumptions on the way databases process transactions [1] than the assumptions made by Continental Pronto (e.g., having access to database internal modules), or both. An exception is [20], which presents a weaker form of active replication to solve non-determinism due to preemption in replicated real-time applications. With primary-backup [4], the idea is to have a single primary database that processes transactions and sends the state updates to one or more backups. For example, [28] uses virtual-memory mapped communication to achieve fast failover by mirroring the primary's memory on the backups. Commonly, most replication techniques based on primary-backup assume a perfect failure-detector mechanism. Pronto [18] is a primary-backup protocol whose idea is to allow for incorrect suspicions by enforcing agreement on failure suspicions among the replicas through a totally-ordered broadcast abstraction.

In terms of functionality, it is possible to run the local-area replication algorithms in an environment with multiple data centers. Although functionally correct, the cost of such an approach would be prohibitive. Algorithms for local-area replication do not distinguish between communication within a data center and communication between data centers. For example, if we ran the Pronto protocol [18] in the multi-data center setting, the number of wide-area messages would be proportional to the total number of databases in the system whereas with Continental Pronto, the number of wide-area messages is proportional to the number of data centers.

7 Conclusion

Traditional approaches to data replication typically provide *either* local-area replication *or* wide-area replication, but usually not both. Moreover, combining these point solutions into a replication mechanism that provides both local-area and wide-area replication is not straightforward. The difficulty of performing this combination is evidenced by the complexity of the protocols that provide disaster-recovery for partitioned databases. Having multiple partitions introduce some, but not all, of the problems with combining local-area and wide-area replication.

Continental Pronto provides a unified approach to wide-area and local-area data replication. One of the keys to cover this space with a relatively simple protocol is the formulation of an underlying communication abstraction, called HABcast. The agreement properties of HABcast give a nice foundation for programming the various durability levels (1-safe and 2-safe) for transactions. Furthermore, the ordering guarantees of HABcast allows us to factor out the complex ordering and dependency issues for transactions that result from combining local-area and wide-area replication.

The price for the relative simplicity of Continental Pronto is the increased "cost" of performing data replication. Where traditional disaster-recovery protocols rely on low-level log shipping, Continental Pronto uses higher-level, and less efficient, transaction shipping. We believe that the resulting flexibility and simplicity is worth the extra price for many online e-commerce systems.

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] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proc. of the 12th Intl. Symposium on Distributed Computing (DISC-12)*, Andros, Greece, September 1998.

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

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

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

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

[7] S. Frølund and F. Pedone. Dealing efficiently with data center disasters. Technical Report HPL-TR 167, Hewlett-Packard Labs, 2000.

[8] H. Garcia-Molina and C. A. Polyzois. Issues in disaster recovery. In *IEEE CompCon*, 1990.

[9] 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.

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

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

[12] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.

[13] K. Hu, S. Mehrotra, and S. Kaplan. An optimized two-safe approach to maintaining remote backup systems. Technical report, University of Illinois at Urbana-Champaign, 1997.

[14] R. Humborstad, M. Sabaratnam, and Ø. Torbjørnsen. 1-safe algorithms for symmetric site configurations. In *Proceedings of the VLDB conference*, 1997.

[15] J. Lyon. Design considerations in replicated database systems for disaster protection. *IEEE CompCon*, 1988.

[16] C. Mohan, K. Treiber, and R. Obermarck. Algorithms for the management of remote backup data bases for disaster recovery. In *Proceedings of the IEEE conference on Data Engineering (ICDE)*, 1993.

[17] Oracle parallel server for windows NT clusters. Online White Paper.

[18] F. Pedone and S. Frølund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, October 2000.

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

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

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

[22] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous systems where processes can crash and recover. In *Proc. of the 20th Intl. Conference on Distributed Computing Systems (ICDCS-20)*, Taipei, Taiwan, April 2000.

[23] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

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

[25] Symmetrix remote data facility. Online data sheet.

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

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

[28] 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: Proofs

To prove CP-1, we define $C(\mathcal{E}_i^{x,e})$ as the committed projection of the execution $\mathcal{E}_i^{x,e}$, involving all transactions committed during epoch e at database d_i^x in group G_x. $C(\mathcal{E}_i^{x,e})$ is a partial order, that is, $C(\mathcal{E}_i^{x,e}) = (\Sigma_i^{x,e}, <_i^{x,e})$, where $\Sigma_i^{x,e}$ is the set of committed transactions in $\mathcal{E}_i^{x,e}$, and $<_i^{x,e}$ is a set defining a transitive binary relation between transactions in $\Sigma_i^{x,e}$. We define $C(\mathcal{E}^{x,e}) = C(\mathcal{E}_i^{x,e}) \cup C(\mathcal{E}_j^{x,e})$ such that $\Sigma^{x,e} = \Sigma_i^{x,e} \cup \Sigma_j^{x,e}$ and $<^{x,e} = <_i^{x,e} \cup <_j^{x,e}$.

Lemma 1 *If $d_i^{x,e}$ and $d_j^{x,e}$ are databases in G_x that do not crash in epoch $e \geq 1$, then $C(\mathcal{E}_i^{x,e}) = C(\mathcal{E}_j^{x,e})$.*

PROOF (SKETCH): We have to show that databases $d_i^{x,e}$ and $d_j^{x,e}$ commit the same transactions (i.e., $\Sigma_i^{x,e} = \Sigma_{j,x}^{x,e}$) in the same order (i.e., $<_i^{x,e} = <_{j,x}^{x,e}$). Assume that d_i^x commits transaction t during epoch e. Therefore, t passed the validation test, and so, d_i^x has executed t during epoch e, and has not executed Deliver(1-SAFE, e, "CHANGE SERVER") or Deliver(1-SAFE, $-$, "CHANGE GROUP") before executing Deliver(1-SAFE, $-, -, e, t, -, -, \mathit{sqlSeq}(t)$). From HB-1 and HB-3 of HABcast, database d_j^x also executes Deliver(1-SAFE, $-, -, e, t, -, -, \mathit{sqlSeq}(t)$) before executing Deliver(1-SAFE, e, "CHANGE SERVER") or Deliver(1-SAFE, $-$, "CHANGE GROUP"). So, t passes the validation test at d_j^x and is committed by d_j^x.

We now show that $<_i^{x,e} = <_j^{x,e}$, that is, if a transaction t_a precedes another transaction t_b in $C(\mathcal{E}_i^{x,e})$, then t_a also precedes t_b in $C(\mathcal{E}_j^{x,e})$. Since database d_i^x executes transactions using a 2PL scheduler, if t_a precedes t_b, then t_a commits before t_b. From the algorithm, it follows that t_a is delivered before t_b. By the HB-3 property of HABcast and the fact that databases commit transactions in the order they are delivered, database d_j^x delivers and commits t_a before t_b. Therefore, t_a precedes t_b in database d_j^x. □

Proposition 1 (CP-1) *Every execution with all committed transactions in a group is 1-copy serializable.*

PROOF (SKETCH): From property DB-1, for any execution $\mathcal{E}_i^{x,e}$, at epoch e, there exists a serial execution $\mathcal{E}_s^e$, involving the committed transactions in $\mathcal{E}_i^{x,e}$, such that $C(\mathcal{E}_i^{x,e})$ is equivalent to $C(\mathcal{E}_s^e)$. By Lemma 1, for every database d_j^x that does not crash in epoch e, $e > 1$, $C(\mathcal{E}_i^{x,e}) = C(\mathcal{E}_j^{x,e})$. Thus, from the definition of committed projection, $C(\mathcal{E}_{l_1}^{x,e}) \cup C(\mathcal{E}_{l_2}^{x,e}) \cup ... \cup C(\mathcal{E}_{l_k}^{x,e}) = C(\mathcal{E}_i^{x,e})$, where d_{l_k} is a database that does not crash in epoch e. Thus, $C(\mathcal{E}_{l_1}^{x,e}) \cup C(\mathcal{E}_{l_2}^{x,e}) \cup ... \cup C(\mathcal{E}_{l_k}^{x,e})$ 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 protocol ensures that $\mathcal{E}_s^e$ 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 is an execution $\mathcal{E}_s$, such that $C(\mathcal{E})$ is equivalent to $C(\mathcal{E}_s)$. □

Proposition 2 (CP-2) *If a client submits a job j, and does not crash, then it will eventually deliver j.*

PROOF (SKETCH): For a contradiction, assume that c submits j, does not crash, but never delivers j. Let τ_1 be a time after which no database aborts transactions; let τ_2 be a time after which no group suffers a disaster failure, and no database crashes or is suspected to have crashed; let $\tau = \max(\tau_1, \tau_2)$. It follows that at some time $\tau' > \tau$, c connects to some database d_i^x that does not crash nor is suspected to have crashed in a group G_x that does not suffer a disaster. From the contradiction hypothesis, c never delivers j, and so, either (a) c always receives an abort message, or (b) blocks forever while waiting for the result of some operation. From case (a) and Algorithm 1, database d_i^x always decides to abort the transactions created by the execution of j, contradicting the assumption that $\tau' > \tau_1$. In case (b), d_i^x either blocks while executing some operation, contradicting the liveness properties of databases, or commits a transaction t associated with job j, but j has level 2 and before d_i^x delivers message $(1\text{-SAFE}, -, -, -, t_a, c, -, -, -)$, it delivers message $(2\text{-SAFE}, -, -, -, t_a, c, -, -, -)$, contradicting property HB-4 of HABcast. □

Proposition 3 (CP-3) *If a database in an operational group commits a transaction t, then all correct databases in all operational groups commit t.*

PROOF (SKETCH): Consider that some database d_i commits transaction t_a. From Algorithm 1, d_i executed Deliver$(1\text{-SAFE}, -, -, c, e_j, t_a, -, -, -)$ such that $e_j \geq e_i$. From property HB-1 of HABcast, every correct database d_k in each operational group eventually executes Deliver$(1\text{-SAFE}, -, -, c, e_j, t_a, -, -, -)$. From the HB-3 property of HABcast, it follows that after executing Deliver$(1\text{-SAFE}, -, -, c, e_j, t_a, -, -, -)$, $e_j \geq e_k$, and so, d_k commits t_a. □

Proposition 4 (CP-4) *If a client delivers a job j then all correct databases in all operational groups commit the transactional updates performed by j.*

PROOF (SKETCH): Since c delivers j, c received a message of the type (t_a, COMMIT), and there is some database d_i that executed send(t_a, COMMIT). Thus, from Algorithm 1, d_i executed Deliver$(2\text{-SAFE}, -, -, c, -, t_a, -, -, -)$ and Deliver$(1\text{-SAFE}, -, -, c, -, t_a, -, -, -)$. From line 50 of Algorithm 1, d_i Broadcast such a message, and by property HB-2 of HABcast, every correct database d_j in each operational group eventually executes Deliver$(2\text{-SAFE}, -, -, c, -, t_a, -, -, -)$. By property HB-4 of HABcast, d_j executes Deliver$(1\text{-SAFE}, -, -, c, -, t_a, -, -, -)$ before, and it follows from Algorithm 1 that d_j commits t_a. □

Proposition 5 (CP-5) *If two databases commit conflicting transactions, they do so in the same order.*

PROOF (SKETCH): Immediate from the fact that transactions are committed in Algorithm 1 according to the total order property HB-3 of HABcast. □