Continental Pronto

Svend Frølund Fernando Pedone
Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA
{frolund, pedone}@hpl.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.

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. For efficiency reasons, we do not want both d and d' to propagate the same updates to x to the backup data centers. 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, and does so for general system configurations with an arbitrary number of data centers that each contain an arbitrary number of databases. Although Continental Pronto can be run in arbitrary system configurations, its performance is similar to classical point solutions when run in system configurations for which those point solutions are defined. For example, we can run Continental Pronto in a single data center only. In this configuration, its performance is sim-

ilar to protocols that provide local-area replication only [13]. We can also run Continental Pronto in a configuration with 2 data centers (or more), each with a single database. Continental Pronto then 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 it uses transaction shipping for both local-area and wide-area replication. For example, database replication within a data center is used to ensure that the disaster-recovery protocol itself can tolerate local failures. Residing above the database 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.

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.

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 subdivide 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 all processes—in practice, several logical links can be multiplexed over the same physical link, and 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.

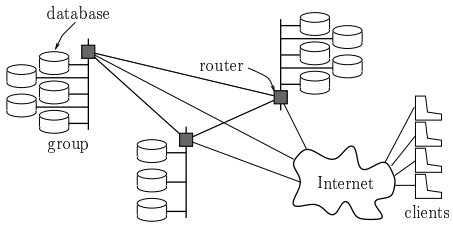

Figure 1: System model

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 [5]. 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 *non-operational*. In this paper, we assume that at least one group is operational.

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

[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, $k_x = \lceil (n_x + 1)/2 \rceil$.

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 [3]).

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 non-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 [3]. 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 [3].

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 non-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.

The HABcast abstraction 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 (in stating the 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,

[2]Notice that HB-2 is not the uniform counterpart of HB-1, in the sense of [8]—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.

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.

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:

- 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 $\mathcal{E}$ [2]. 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 $\mathcal{E}$ [2].

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

3 Problem Specification

We outline next the properties that characterize what it means for a data replication protocol such as Continental Pronto to be correct. The first property requires the replication algorithm to provide the illusion that there is only a single copy of each data item:

- CP-1: Every execution consisting of all committed transactions in a group is 1-copy serializable [2].

The second property forces the replication algorithm to make progress if it is initiated by a client. To state this 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.

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

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).

Property CP-2 ensures that the effects of a job are durable in a single database. However, for a replicated database system, we need a global notion of durability. 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 [7], 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.

Property CP-3 in conjunction with CP-2 ensures that if the client delivers a job, and no disasters happen, the job's transactional updates are propagated to all databases in the system. If a disaster happens, 1-safe durability may give rise to "lost transactions:" the client may deliver a job and then subsequently fail over to a database whose state does not reflect the job's updates. In contrast, 2-safe durability prevents lost transactions:

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

Continental Pronto ensures either CP-3 or CP-4. Furthermore, the choice between these durability levels is configurable.

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

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.

Algorithm 1 Database d_i^x in group G_x

```
1:  Initialization:
2:    e_i ← 1
3:    prmy_grp_i ← 1
4:    prmy_db_i ← 1
5:  To execute a transaction:
6:  when receive (t_a, request) from c do
7:    case request = begin(job_id, durability, t_a):
8:      if d_i^x ≠ prmy_db_i or G_x ≠ prmy_grp_i then
9:        send (t_a, "I'M NOT PRIMARY") to client(t_a)
10:     else
11:       client(t_a) ← c
12:       job(t_a) ← job_id
13:       level(t_a) ← durability
14:       state(t_a) ← EXECUTING
15:       begin(t_a)
16:       wait for response(t_a, result)
17:       send (t_a, result) to client(t_a)
18:   case (request = exec(t_a, sql-req) or
                      request = abort(t_a)) and
                       state(t_a) = EXECUTING:
19:     exec task
20:       exec(t_a, sql-req)
21:       wait for response(t_a, result)
22:       if result = ABORTED then
23:         state(t_a) ← ABORTED
24:       send (t_a, result) to client(t_a)
25:   case request = commit(t_a) and
                       state(t_a) = EXECUTING:
26:     state(t_a) ← COMMITTING
27:     sql-seq ← all exec(t_a, sql-req) in order
28:     Broadcast(d_i^x, G_x, e_i, t_a, client(t_a), job(t_a),
                                level(t_a), sql-seq)
```

In the following we present an overview of Continental Pronto and its main algorithm. The complete algorithm and its detailed explanation and proof of correctness can be found in [5].

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. 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. Continental Pronto implements primary-backup on top of 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

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

29: **To commit a transaction:**

30: **when** Deliver(1-SAFE, $d_j^y, G_y, e_j, t_a, client(t_a), job(t_a), level(t_a), \texttt{sql-seq}$) **do**
31: **if** $\exists\, t_b \neq t_a$,s.t. $state(t_b) =$ COMMITTED **and** $job(t_b) = job(t_a)$ **then**
32: **if** $level(t_a) =$ 1-SAFE **then**
33: send $(-,$ COMMITTED$)$ to $client(t_a)$
34: **else**
35: **if** $e_j < e_i$ **then**
36: execute $\texttt{abort}(t_a)$
37: **wait for** $\texttt{response}(t_a, result)$
38: $state(t_a) \leftarrow$ ABORTED
39: **else**
40: **if** $d_i^x \neq prmy\_db_i$ **or** $G_x \neq prmy\_grp_i$ **then**
41: **for each** $(t_a, \texttt{sql-req})$ in $\texttt{sql-seq}$ **do**
42: execute $\texttt{sql-req}$
43: **wait for** $\texttt{response}(t_a, result)$
44: execute $\texttt{commit}(t_a)$
45: **wait for** $\texttt{response}(t_a, result)$
46: $state(t_a) \leftarrow$ COMMITTED
47: **if**($state(t_a) =$ ABORTED **or** $level(t_a) =$ 1-SAFE) **and** $d_i^x = d_j^y$ **and** $G_x = G_y$ **then**
48: send $(t_a, state(t_a))$ to $client(t_a)$
49: **when** Deliver(2-SAFE, $d_j^y, G_y, e_j, t_a, client(t_a), job(t_a), level(t_a), \texttt{sql-seq}$) **do**
50: **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**
51: send $(t_a,$ COMMITTED$)$ to $client(t_a)$

52: **To request primary server/group change:**

53: **when** $prmy\_db_i \in \mathcal{D}_i$ **and** $G_x = prmy\_grp_i$ **do**
54: Broadcast(e_i, "CHANGE SERVER")
55: **when** $prmy\_grp_i \in \mathcal{DD}$ **do**
56: Broadcast($prmy\_grp_i$, "CHANGE GROUP")

57: **To change primary server/group:**

58: **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**
59: **if** $prmy\_db_i = d_i^x$ **and** $G_x = prmy\_grp_i$ **then**
60: **for each** t_a s.t. $state(t_a) =$ EXECUTING **do**
61: execute $\texttt{abort}(t_a)$
62: **wait for** $\texttt{response}(t_a, result)$
63: $state(t_a) \leftarrow$ ABORTED
64: send $(t_a,$ ABORTED$)$ to $client(t_a)$
65: $e_i \leftarrow e_i + 1$
66: **if** Delivered (1-SAFE, "CHANGE GROUP") **then**
67: $prmy\_grp_i \leftarrow prmy\_grp_i + 1$
68: $prmy\_db_i \leftarrow e_i$ **mod sizeof**(group $prmy\_grp_i$)

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 [13]. 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 preassigned 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 job using the current primary.

5 Performance Assessment

5.1 Implementing HABcast

The performance of the broadcast abstraction has a major impact on the performance of Continental Pronto, and so, we discuss here how to implement it. HABcast [6] 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. The basic communication pattern in HABcast is the following. To broadcast a message m, a process in the primary group first executes a local atomic broadcast within its group. The primary group has a coordinator process, and when this coordinator delivers the local broadcast message, it executes (1-SAFE, m) and sends m to a single process in each backup group. When a process p in a backup group receives m, it atomically broadcasts m within that group. When p delivers m as part of the local atomic broadcast mechanism, it sends an acknowledgement to the coordinator in the primary group. Upon receiving this acknowledegement, the coordinator reliably broadcasts the acknowledgement to the rest of the primary group and then executes (2-SAFE, m). A detailed discussion about how HABcast handles failures, suspicions, and disasters is out of the scope of this paper, and can be found in [6].

5.2 Analytical Evaluation

We compare Continental Pronto to two algorithms that deal with data center disasters: 1-safe and 2-safe [7]. Although [7] considers a single backup only, 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 the number of messages and the latency as metrics for our comparison. For the latency analysis, we distinguish between δ_l, the transmission delay along local-area network 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 the 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 the communication as 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 [3] with some optimizations [16], 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, $-$). This latency is composed of a local Atomic Broadcast in the Primary group and 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**	**WAN**	**LAN**
1-safe	0	$(n-1)$	0
CP 1-safe	$2\,\delta_l$	$2(n-1)$	$(k-1)(3n+1)$
2-safe	$2\,\delta_w$	$2(n-1)$	0
CP 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-SAFE, $-$) for the previous message.

Figures 2 and 3 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 2 compares the times to deliver messages of the type (1-SAFE, −) and (2-SAFE, −) in a system with 3 groups. Not surprisingly, most of the overhead to deliver a message of type (2-SAFE, −) is related to wide-area messages.

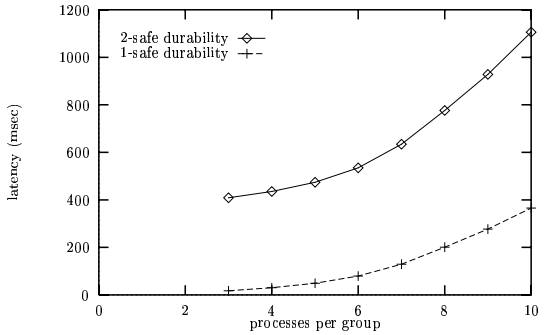

Figure 2: 1-safe and 2-safe durability

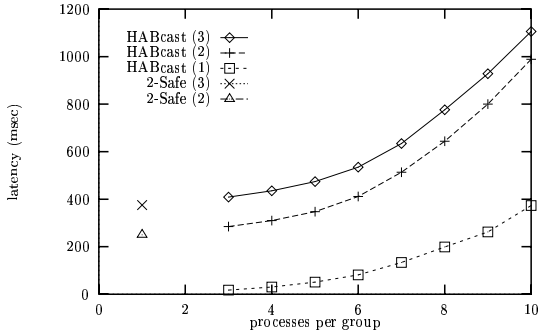

Figure 3: Comparing HABcast to 2-safe

Figure 3 compares the time to deliver messages of the type (2-SAFE, −) with the time of the 2-safe algorithm, and 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

Due to space constraints, we only provide a brief summary of related work. In [5], we give a more detailed description of these related approaches. Existing approaches address either disaster recovery or high availability, but typically not both. We start out by comparing Continental Pronto to existing work on disaster recovery. Then we compare Continental Pronto to replication algorithms that provide high availability.

The algorithms in [11] seek to reduce the resource consumption of 1-safe algorithms by parallelizing the processing of log entries at backup sites. The algorithms in [14] provide 1-safe semantics for a system where both the primary and backup sites contain multiple database instances, each with their own partition of the database. As an extension, [10] allows the same site to contain both primary and backup partitions. The algorithms in [9] do not consider such mixed sites. Instead, the algorithms provide 2-safe semantics with early release of locks at the primary (using the lazy commit optimization in [7]). The notion of 0-safe is introduced in [4] to allow for multiple primaries. The basic assumption to avoid inconsistency is that all transactions commute.

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.

In terms of replication for high availability, one approach is to use a parallel database system, such as OPS [12] or XPS [17]. These par-

allel database systems typically require special hardware for disk sharing between the various instances of the database. A number of systems provide local-area replication without special hardware (e.g., [1, 15, 13]).

These protocols provide high availability only: if we ran these protocols in a multi-data-center environment, the cost would be prohibitive in terms of wide-area messages. For example, if we ran the Pronto protocol [13] 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

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.

References

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

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

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

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

[5] S. Frølund and F. Pedone. Continental pronto. Technical Report HPL-TR 166 (R.1), Hewlett-Packard Laboratories, 2000.

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

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

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

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

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

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

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