Dealing Efficiently with Data-Center Disasters

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

Abstract

High-end, mission-critical computer systems commonly guard against disasters. Such systems are composed of data centers (i.e., local-area networks of failure-independent computers) in distributed geographical locations, connected through wide-area network links. Wide-area network links are a major source of overhead, and to build efficient disaster-resilient protocols, their use should be reduced without compromising the overall reliability of the system.

This paper claims that efficient disaster-resilient protocols can be devised by adequately modeling wide-area distributed systems. To support our claim, we define a model for wide-area distributed systems that distinguishes between data-center disaster failures and computer failures, and develop a hierarchical Atomic Broadcast protocol for this model. The main idea behind a hierarchical protocol is to run a local sub-protocol within each local-area network, and then use a global protocol to orchestrate the communication between the local protocols across wide-area links. The hierarchical nature of the protocol, and the accuracy of disaster detection, allows us to achieve disaster resilience with few messages across wide-area links.

1 Introduction

High-end mission-critical computer systems commonly guard against disasters. Disasters can be caused by the environment (e.g., floods, earthquakes, fires), and also a coincidence of events within the computer system itself (e.g., operator errors, simultaneous crash of critical components due to software faults). To deal with environment disasters, computer systems typically run in multiple geographically dispersed data centers. Data centers are connected through possibly redundant wide-area network links, and typically operate in a primary-backup manner: If the primary data center suffers a disaster, the backup data center takes over. To enable the take over, the data in the backup data center is continuously kept up-to-date.

A computing infrastructure with multiple data centers has some interesting characteristics. First, since data centers are connected via wide-area networks, the difference in communication cost within a data center (i.e., a local-area network) and across data centers (i.e., a wide-area network) may be significant. Therefore, it makes sense to reduce the number of messages between data centers at the expense of increasing the number of messages within a data center. Second, disasters are rare events with very serious consequences, and so, in practice, disaster detection involves human intervention: disaster detection may involve phone calls and emergency radio systems in addition to timeout violations from within the computer system itself. Such out-of-band confirmation of disasters has an impact on the system, since it is reasonable to assume that no disaster is suspected unless it has actually happened. This aspect of disaster detection may be counter intuitive at first. From the computer system's perspective, disasters are detected via wide-area links, which are known to be un-predictable. Thus, if one considers disaster detection from within the computer system only, one would expect the detection of disasters to be less reliable than the detection of failures within a given data center. However, this view ignores the out-of-band confirmation of disasters by human operators.

Previous work on wide-area distributed systems use a homogeneous model in which all processes communicate via wide-area network links [ADS00, KSMD00, CHD98, RFV96, MS95]. The main goal of these previous approaches is to reduce inter-process communication in general and at the same time handle highly unreliable failure detection. In this paper, we take a different look at disaster-resilient wide-area distributed systems. In particular, we consider that (a) a data center can be composed of several processes, which may fail independently of one another, (b) communication between processes in different data centers is slow when compared to communication between processes in the same data center, and (c) disaster detectors do not make mistakes, that is, a data center that does not suffer a disaster is never suspected by processes in other data centers, and a data center that suffers a disaster is eventually suspected by processes in other data centers.

We introduce an explicit notion of disaster as an event that makes a data center become inoperational. A disaster is defined as the aggregate failure of a number of processes within the same data center. We also introduce *disaster detectors*, which, like failure detectors [CT96], is a distributed oracle. Where failure detectors give hints about process crashes, disaster detectors give hints about data centers disasters. Therefore, processes can rely on failure detectors to monitor the failures of other processes in their data center, and disaster detectors to monitor the disasters of data centers. Disaster detectors is a novel concept to model data center disasters. Alternatively, one could use, for example, failure detectors to detect disasters. By having disaster detectors as a primitive concept, however, we can represent the idea that a process p in a data center d can suspect that another data center d' has suffered a disaster, even if p has no knowledge about individual processes in d'. Having a failure detector against each individual process in d' gives (strictly) more information than having a disaster detector against d' (for detectors with similar types of accuracy and completeness).

Disaster-resilient systems often rely on some replication mechanism. For example, to implement disaster recovery for a database system, a typical configuration is to have an active database in a primary data center and a stand-by database in a backup data center. Atomic Broadcast can be used as a basic building block to implement replication in this scenario. As shown in [PF00], Atomic Broadcast is the only inter-site mechanism necessary to synchronize replicas, and so, one can move from a system composed of a single data center to a system composed of several data centers—as the one described here—to cope with disaster failures by using an Atomic Broadcast appropriate to the multi-data-center case. In this paper, we develop an Atomic Broadcast protocol for such cases. Our protocol exchanges only $2(n-1)$ messages between data centers to deliver a message, where n is the number of data centers. The protocol is hierarchical in the sense that each data center has a local Atomic Broadcast protocol, and an inter-data-center protocol orchestrates the execution of the local protocols to solve Atomic Broadcast globally. The protocol illuminates some of the difficulties in building such hierarchical protocols for wide-area networks.

Our contributions are the following:

- We present a distributed system model in which to represent disasters and the detection of disasters. Disaster detection allows us to model a system in which a process can reliably detect the in-operability of a part of the system, without giving each process reliable knowledge about the crash of individual processes.

- We show how to use a notion of *hierarchical* algorithm to deal efficiently with underlying heterogeneity in the network (a network that has both wide-area and local-area links). We use a Hierarchical Atomic Broadcast protocol to illustrate the concept of hierarchical algorithm. The main idea is to run a local Atomic Broadcast algorithm within each data center, and then use a global wide-area and disaster-aware algorithm to orchestrate the local algorithms. With this hierarchical organization, we can coalesce the many-to-many communication between processes in different data centers, that would arise if we were to use a conventional Atomic Broadcast protocol globally, into a one-to-one communication pattern between "coordinators" in each data center.

The paper is organized as follows. Section 2 defines the system model and introduces the notions of disaster and disaster detectors. Section 3 presents in detail a hierarchical Atomic Broadcast protocol for wide-area networks. Section 4 briefly discusses the cost of such protocol, and Section 5 concludes the paper.

2 System Model

We consider a system Π with a finite set of processes. Processes can fail by crashing, that is, when a process fails, it permanently stops executing its algorithm—we do not consider Byzantine faults nor do we rely on the ability for processes to recover. A *correct* process is one that does not fail. To model the notion of data center, we subdivide the set Π of processes into a number of disjoint subsets, $g_1, \ldots g_n$, where each subset g_x represents a group (or data center). We use G to denote the set of groups, that is, for any group of processes $g_x = \{p_1^x, \ldots, p_{k_x}^x\}$, $g_x \in G$.

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 processes—a group g_x suffers a disaster if a majority of processes in g_x have failed.[1] A group that does not suffer a disaster is *operational*; a group that suffers a disaster is *in-operational*.

Processes communicate by message passing. We assume that the system is asynchronous: message-delivery times are un-bounded, as is the time it takes for a process to execute individual steps of its local algorithm. Each process in the system has access to a timer. The presence of timers does not introduce any notion of (real) time or synchrony into the system. A timer has two primitives: set and expire, and guarantees that if process p_i sets a timer, and does not crash, then p_i's timer will eventually expire.

2.1 Failure Detectors

We equip the system with failure detectors [CT96]. Each process p_i in a group g_x has access to a failure detector that gives hints about the crash of processes in g_x. A failure detector does not detect failures across groups. Although the classical notion of failure detection in [CT96] is global (i.e., it has no notion of groups), we can reuse the basic definitions directly by considering each group a separate system of processes with respect to the classical definitions.

A failure detector returns the set of processes that it believes to have crashed. If p_i's failure detector returns a set that includes a process p_j, we say that p_i *suspects* p_j. The failure detector D_x available to processes in a group g_x is in the class of eventually strong failure detectors (i.e., $\diamond\mathcal{S}$). That is, D_x satisfies the following properties:

- *Strong Completeness*: Eventually, every process in g_x that crashes is permanently suspected by every correct process in g_x, and
- *Eventual Weak Accuracy*: If the group g_x contains a correct process, then there is a time after which some correct process in g_x is never suspected by any correct process in g_x.

[1] We could give a more general definition of disasters where the number of process failures is a parameter. We use this specific definition to simplify the presentation. We define disasters relative to the failure of a *majority* of processes because the intended function of a group is to solve Atomic Broadcast with an unreliable failure detector (see Section 2.3).

2.2 Disaster Detectors

A disaster detector gives hints about which groups are in-operational. Each process in the system has access to a disaster detector, DD, that returns the set of processes that DD believes to be in-operational. If DD returns a set including a group g_x to a process p_i, we say that p_i suspects g_x (to be in-operational). We define accuracy and completeness requirements for disaster detectors as follows:

- *Strong Completeness*: Eventually, every group that contains fewer than a majority of correct processes is permanently suspected by every correct process in the system, and
- *Strong Accuracy*: No group is suspected by any process in the system before it contains fewer than a majority of correct processes.

We define disasters in terms of process crashes, and we can formalize the notion of disaster detection using similar machinery as [CT96].

2.3 Communication

Messages are structured values. Besides the actual data being transmitted, a message has fields that contain meta data. A message has a field called sender, which identifies the process that sent the message. A message also has a field called id, which uniquely identifies the message. Given a message m, we refer to the meta data using "." notation, for example m.sender.

We capture the notion of message passing through the primitives **send** and **receive**. These primitives provide reliable channels:[2]

- *Completeness*: If a process p_i sends a message to a process p_j, then if neither process fails, the message will eventually be received by p_j,
- *No duplication*: A message sent is received at-most-once, and
- *No creation*: A message is only received if it was sent.

Processes in an operational group can communicate using Reliable Broadcast and Atomic Broadcast. Reliable Broadcast is defined by the primitives r-broadcast and r-deliver. Atomic Broadcast is defined by the primitives a-broadcast and a-deliver. Given an operational group g_x, Reliable Broadcast guarantees the following properties:

- *Validity*: If a correct process in g_x r-broadcasts a message m, then all correct processes in g_x eventually r-deliver m,

[2] These properties do not exclude link failures, they simply assume that failed links are eventually repaired.

- *Agreement*: If a correct process in g_x r-delivers a message m, then all correct processes in g_x eventually r-deliver m, and

- *Integrity*: For any message m, every correct process r-delivers m at most once, and only if m was previously r-broadcast by $m.\mathsf{sender}$.

Atomic Broadcast has the same guarantees as Reliable Broadcast plus the following one:

- *Total Order*: If two correct processes p_i and p_j both a-deliver messages m and m', then p_i a-delivers m before m' if and only if p_j a-delivers m before m'.

Solving Atomic Broadcast in a group g_x of processes whose failure detectors are of class $\diamond\mathcal{S}$ requires that a majority of processes in g_x be correct [CT96]. This is why we require an operational group to contain a majority of correct processes.

2.4 Hierarchical Atomic Broadcast

In the following, we define Hierarchical Atomic Broadcast, a broadcast communication primitive appropriate for multi-data-center systems. Hierarchical Atomic Broadcast is defined by the primitives HA-Broadcast and HA-Deliver, which take the notion of groups into account. Hierarchical Atomic Broadcast guarantees the following properties:

- *Validity*: If a correct process p_i in an operational group g_x HA-Broadcasts a message m, then p_i eventually HA-Delivers m,

- *Agreement*: If a correct process p_i in an operational group g_x HA-Delivers a message m, then every correct process p_j in each operational group g_y eventually HA-Delivers message m,

- *Integrity*: For any message m, each process HA-Delivers m at most once, and only if m was previously HA-Broadcast by $m.\mathsf{sender}$, and

- *Total Order*: If a correct process p_i in an operational group g_x and a correct process p_j in an operational group g_y both HA-Deliver messages m and m', then p_i HA-Delivers m before m' if and only if p_j HA-Delivers m before m'.

Atomic Broadcast is a special case of Hierarchical Atomic Broadcast, that is, when G contains only one group, Hierarchical Atomic Broadcast becomes Atomic Broadcast.

3 Solving Hierarchical Atomic Broadcast

In this section, we present a protocol that solves Hierarchical Atomic Broadcast. We first describe an overview of the protocol, and then discuss it in detail.

3.1 Protocol Overview

The Hierarchical Atomic Broadcast protocol distinguishes between a primary group and backup groups. The primary group determines the order in which messages are delivered. Each group has a coordinator process, responsible for the interaction between groups. During periods when no process in a group is suspected, the group has only one coordinator, but, due to false failure suspicions, more than one coordinator may exist in a group. As we show below, the protocol can cope with more than one coordinator in the same group at the same time. To simplify the presentation, in the following, we focus first on executions without failures and failure suspicions.

Executions without Failures and Suspicions. To HA-Broadcast a message m, a process p_i in the Primary Group executes an a-broadcast(m). If p_i is in a Backup Group, p_i sends m to some process p_j in the Primary Group, which will a-broadcast m. The initial local Atomic Broadcast in the Primary Group determines m's order. If p_i is the coordinator of the Primary Group, after a-delivering m, p_i sends m to some process p_j in each operational Backup Group g_x. Process p_i knows which Backup Groups are operational since it has access to a perfect disaster detector. Upon receiving m, p_j executes a-broadcast(m). Every process in g_x HA-Delivers m after a-delivering it, but only the coordinator in g_x sends a reply to p_i. By doing so, the number of messages exchanged between the Primary Group and any Backup Group is limited to 2. When p_i gathers replies from processes in every operational Backup Group, p_i r-broadcasts a message to processes in the Primary Group, confirming that m can be HA-Delivered.

The confirmation message r-broadcast by p_i is not necessary to guarantee agreement of Hierarchical Atomic Broadcast, but it guarantees that any message m HA-Broadcast by a process in the Primary Group is only HA-Delivered by processes in the Primary Group if m is received by the Backup Groups. The latency of HA-Delivery in the Primary Group can be reduced by allowing processes in the Primary Group to HA-Deliver m right after they a-deliver it. However, we require processes to wait for m to be r-delivered because we need a stronger agreement property when we use Hierarchical Atomic Broadcast as a building block for primary-backup database replication [PF00].[3] Figure 1 depicts an execution of the protocol without failures and failure suspicions.

Executions with Failures and Suspicions. The Hierarchical Atomic Broadcast protocol deals with process failures and disaster failures. Moreover, because of the characteristics of the failure detector used to detect process failures, the algorithm also has to handle incorrect failure suspicions.

- **Dealing with process failures and failure suspicions.** Let p_i be a process in some group g_x that fails or is suspected to have failed. If p_i is not the current coordinator in

[3] In particular, it means that a primary database never confirms the commit of a transaction to a client before the backups have also committed the transaction, preventing transactions from being lost in case of failures.

g_x, p_i's failure, or suspicion of failure, has no impact on the Hierarchical Atomic Broadcast protocol (even though it may have an impact on the local Atomic Broadcast protocol). If p_i is the current group coordinator and fails, then the remaining processes will eventually suspect it, and another one will be elected coordinator. Other groups do not have to be informed about this change in role because messages can be received by any process in a group, be it a coordinator or not; the coordinator is only responsible for sending messages to other groups. Due to false failure suspicions, it can happen, for certain periods of time, that several coordinators co-exist in the same group. Although this has an impact on the performance of the protocol, it does not compromise its correctness: since processes execute in a deterministic way, all coordinators will end up sending the same messages to other groups, and it does not matter which message is received and processed first; furthermore, processing a message is an idempotent operation, and so, it can be done several times.

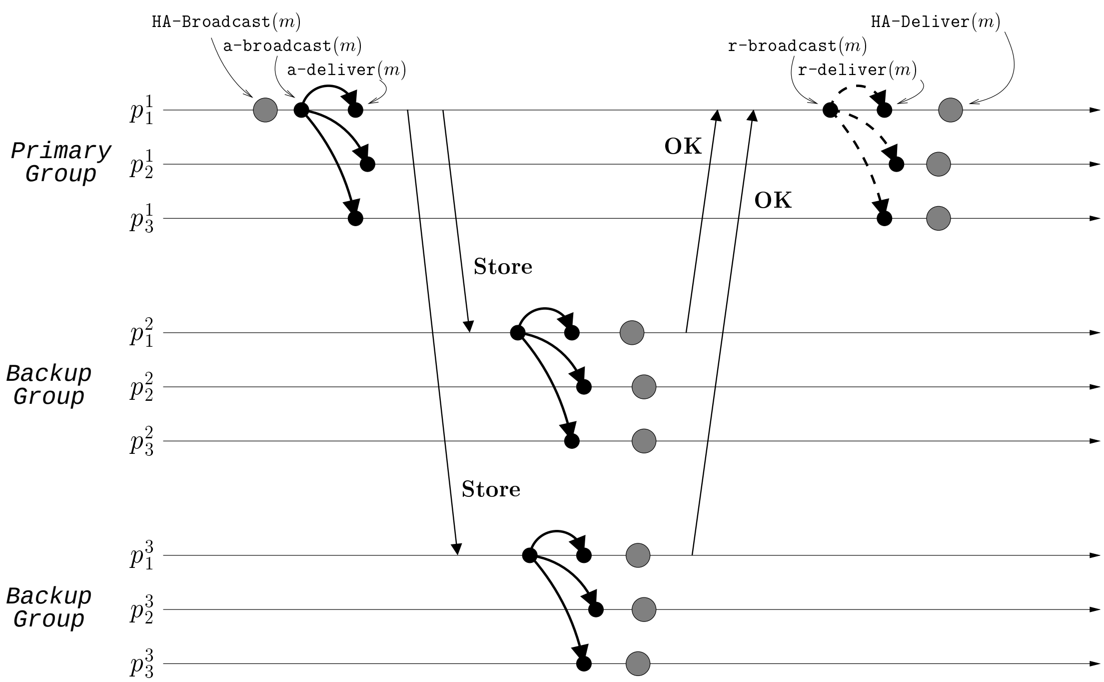

Figure 1: Execution without failures and failure suspicions

- **Dealing with disaster failures.** The action taken by a process p_i upon detection of a disaster failure depends on p_i's group. If p_i is in the primary group, and suspects some backup group g_x to have suffered a disaster, p_i simply starts to ignore g_x (i.e., by not sending messages to processes in g_x). The mechanism is more complex when processes in a backup group suspect the primary group. Groups are assigned an identification number a priori that allows them to know which group should become the next primary when the current one suffers a disaster failure. If the next pre-determined group also suffers a disaster failure, the one that succeeds it takes over, and so forth. Before a backup group takes the role of primary, it becomes a transition group. Processes in a transition group determine all messages that have been HA-Delivered by every operational backup group so far, and make sure that every operational group receives such messages before receiving any new HA-Broadcast message.

Figure 2 depicts an execution of the protocol where process p_1^1, the current coordinator of the primary group fails, process p_3^1 suspects p_1^1, and process p_2^1 takes over as next coordinator and re-contacts the backup groups.

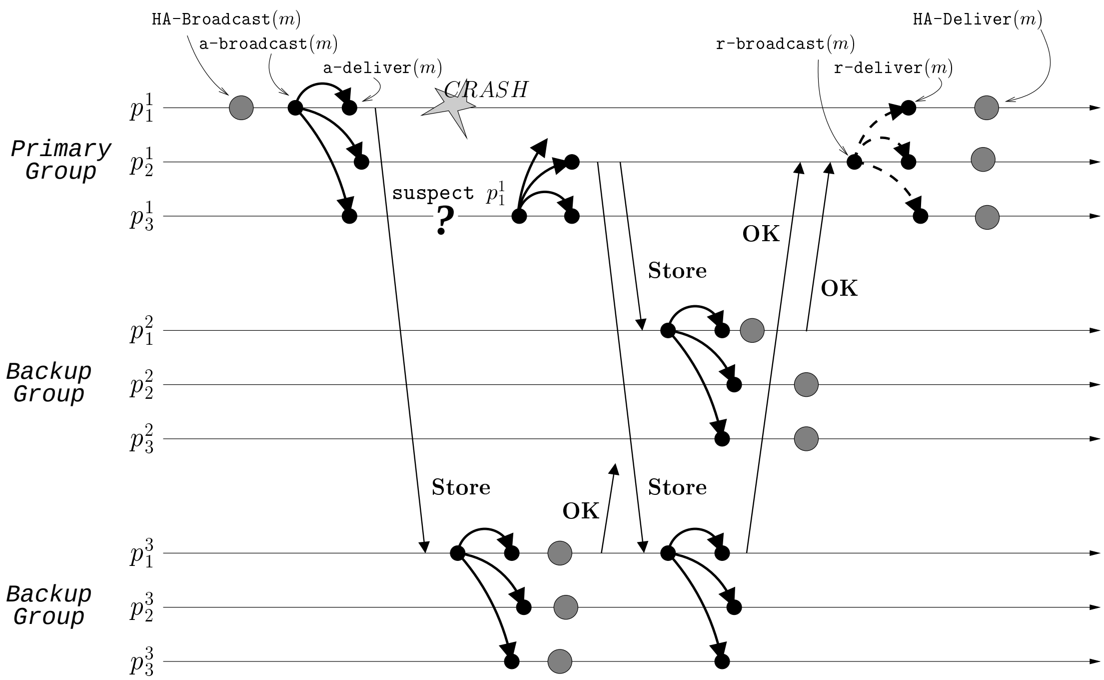

Figure 2: Execution with process failures

3.2 Protocol Description

We present our protocol in the form of object-oriented pseudo-code. Each process in the system instantiates an object from the class `habcast` (see Figure 3). These resulting set of objects collectively implement the Hierarchical Atomic Broadcast protocol. The pseudo-code in Figure 3 declares the state of each object as a number of variables. Furthermore, each object has a number of private methods that it may call during its execution. Finally, the class `habcast` declares three behaviors: `PrimaryGroup`, `BackupGroup`, and `TransitionGroup`. These behaviors capture the various roles that an object can play throughout its lifetime. The initial roles are assigned to objects in the `initially` clause in the `habcast` class. Subsequently, objects can then transition from one behavior to another by means of executing a `become` statement. The notion of behavior transition is inspired by the Actor model of computation [Hew77, Agh86]. We describe the pseudo code for the three possible object behaviors in Figure 4, 5, and 6. We give a more detailed description of the pseudo-code semantics in Appendix B.

The `current-epoch` variable denotes the number of times a group has changed coordinator. The coordinator in a group orchestrates the interaction with other groups. The `local` and `global` variables each refer to a set of messages. The `local` set contains the messages which have been a-broadcast within the primary group; the `global` set contains the messages which have been HA-Broadcast. For any process, the integer variable `del-index` is the largest sequence number of an HA-Delivered message. That is, all messages with sequence numbers smaller than `del-index` have all been HA-Delivered. The primary group assigns sequence numbers to messages. Each process has a counter, `msg-index`, which is the next sequence number to be assigned. A process in a backup group may also initiate the HA-Broadcast of a message. If it does so, it stores the message in a set called `broadcast`. Messages are stored in this set so that they can be re-submitted if the primary group fails. The variable `my-identity` is the identity of a given process; the variable `my-group` is this process' group. The current coordinator of group `x` is the value of `coord[x]`.

The method `send-to-groups` captures the behavior involved in sending a request to a number of groups. Basically, for each group, the method keeps sending the request to a process in the group until the request is acknowledged or the group is suspected to be in-operational. The return value of `send-to-groups` is a vector of replies, one from each group. The method `insert` adds a set of messages, `from`, to another set of messages, `into` (with duplicate elimination based on message identifiers). The method `deliver` checks to determine if the set `global` contains any messages that can be HA-delivered. If so, those messages are HA-delivered, and removed from the `broadcast` set.

Objects export a number of methods and event handlers (`when` clauses). An object in the primary group exports a method, called `HA-Broadcast`, which other objects (in the same process) can call to HA-Broadcast messages (see Figure 4). The `HA-Broadcast` method is the "entry point" into our protocol. To HA-Broadcast a message `m`, an object in the primary group first performs a local a-broadcast within the group, assigns a sequence number to the message, sends the message to all the backup groups (using `store`), and then delivers the message. We use the a-broadcast primitive to ensure that processes in the primary group agree on the set of a-delivered messages, the assignment of sequence numbers, and suspicion of coordinator crashes. The `local` set of messages contains the messages which have been a-broadcast within the primary group and assigned sequence numbers. Once these messages have been a-broadcast (or "stored") in the backup groups, they can be HA-Delivered in the primary group. The `store-messages` method is a private method that is called from within objects only; it "pushes" messages from the primary group to the backup groups. The `initially` clause is executed when an object first instantiates the `PrimaryGroup` behavior.

An object with the `BackupGroup` behavior also exports a `HA-Broadcast` method (see Figure 5). This method allows processes in a backup group to initiate the HA-Broadcast of messages. To HA-Broadcast a message, an object sends the message to some process in the primary group. The primary group then determines the order for the message through the sequence-number mechanism, and then pushes the message to all the backup groups, including the group that initiated the HA-Broadcast. To push messages to a backup group, the primary group sends a `store` message to an individual process in the backup group. This process then a-broadcasts the `store` message within the backup group. Only the coordinator in the backup group sends an acknowledgement to the primary group—this is to limit the inter-group communication to 2 messages. When an object in a backup group a-delivers a `store` message, it can HA-Deliver that message. The coordinator rotation mechanism in a backup group is similar to the one employed in a primary group: processes use local broadcast to agree on suspicions. In addition to process failures within the group, a backup group also detects the failure of the primary group through a disaster detector `DD`. The group uses local broadcast to order the detection of disasters with the receipt of `store` messages. If a backup group detects a disaster, and if it supposed to take over as primary group, all objects in the group take on a new behavior, called `TransitionGroup`. The event handlers for `update-state` and `send-state` are concerned with the state synchronization between backup groups when the primary group fails and one of the backup groups takes over as new primary. During the take-over (or transition phase), the groups need to agree on which messages have been HA-Delivered.

The behavior `TransitionGroup` (see Figure 6) contains the pseudo-code for the transition phase. This is the behavior executed by objects in a backup group that is about to take over as a new primary group. The behavior captures the state synchronization among the backup groups. The coordinator in the a transition group invokes the `obtain-state` method. For each backup group, this method obtains the set of messages that has been HA-Delivered in that group. First, the method sends a `send-state` message to all backup groups. In response to the this message, each group simply sends all messages which have been HA-Delivered in that group. Based on the replies, the coordinator in the transition group updates its own view of HA-Delivered messages, and sends this view to all backup groups. The coordinator then a-broadcasts an `update-state` message within the transition group to make all group members take on the behavior of a primary group.

4 Evaluation of the Protocol

We compare the Hierarchical Atomic Broadcast protocol with the Chandra and Toueg [CT96] Atomic Broadcast protocol (CT-broadcast) and the Optimistic Atomic Broadcast protocol in [Ped99] (OPT-broadcast). This comparison is done for reference purposes only as both Atomic Broadcast protocols assume a different system model than the Hierarchical Atomic Broadcast protocol:

```
class habcast {

  current-epoch := 1;
  local := ∅;                  // The set of all messages seen by a process
  global := ∅;                 // The set of messages HA-Delivered by a process
  del-index := 0;              // The sequence number of the last delivered message
  msg-index := 1;              // The current sequence number for this process
  broadcast := ∅;              // The set of broadcast, but not delivered messages
  my-identity := ...;
  my-group := ...;
  all-groups := ..;
  for y in all-groups do coord[y] := 1;
  for y in all-groups do size[y] := |y|;

  private method Result[] send-to-groups(Groups g,Request req)
    for all y in g do
      fork task:
        repeat
          send req to coord[y];
          set timeout;
          wait until (receive(res) from p in y) or (time expires) or (y in DD);
          if time expired then coord[y] := (coord[y] mod size(y)) + 1
        until (received(res) from p in y) or (y in DD);
        if received(res) from p in y then
          coord[y] := p;
          response(y,res);

    wait until (for all y in g: response(y,res) or y in DD);
    return vector of all response values;

  private method insert(Message-set from,Message-set into)
    for all m in from do
      if ∄ m' in into such that m.id == m'.id then
        into := into ∪ { m };

  private method deliver()
    while ∃ m in global such that m.index == del-index + 1 do
      HA-Deliver(m);
      if ∃ m' in broadcast such that m.id == m'.id then
        broadcast := broadcast \ { m' };
      del-index++;

  behavior PrimaryGroup { ... }
  behavior BackupGroup { ... }
  behavior TransitionGroup { ... }

  initially do
    if my-group == 1 then
      become PrimaryGroup()
    else
      become BackupGroup();
}
```

Figure 3: Hierarchical atomic broadcast protocol

```
 behavior PrimaryGroup {

   public method HA-Broadcast(Message m)
     a-broadcast([Send,m,nil]);

   private method store-messages()
     mset := { m in local | m.index > del-index };
     send-to-groups(G \ me,[Store,mset,me]);
     r-broadcast(mset);

   initially do
     for all m in broadcast do HA-Broadcast(m);
     local := global;

(a) when receive(req) from dest do
     a-broadcast(req);

(b) when a-deliver([Send,m,dest]) do
     if for all m' in msg-set: m'.id ≠ m.id then
       m.index := msg-index++;
       insert({ m },local);
     if coord[my-group] == my-identity then
       if dest != nil then
         send [OK,m] to dest;
       store-messages();

(c) when r-deliver(mset) do
     insert(mset,global);
     deliver();

(d) when coord in D do
     a-broadcast([Change-epoch,current-epoch]);

(e) when a-deliver([Change-epoch,e]) do
     if e == current-epoch then
       coord[my-group] := (current-epoch mod size(my-group)) + 1;
       current-epoch++;
       if my-identity == coord[my-group] then
         store-messages();
 }
```

Figure 4: Primary group behavior

```
 behavior BackupGroup {

   public method HA-Broadcast(Message m)
     broadcast := broadcast ∪ { m };
     send-to-groups(primary-group,[Send,m,me]);

(a) when receive(req) from dest do
     a-broadcast(req);

(b) when a-deliver([Store,mset,dest]) do
     if dest is in primary-group then
       insert(mset,global);
       deliver();
       if coord[my-group] == my-identity then
         send [OK,mset] to dest;

(c) when a-deliver([Send-state,g,dest]) do
     if g > primary-group then
       primary-group := g;
     if coord[my-group] == my-identity then
       send [OK,{ m in global | m.index ≤ del-index }] to dest;

(d) when a-deliver([Update-state,mset,dest]) do
     insert(mset,global);
     deliver();
     send [OK,mset] to dest;
     for all m in broadcast do
       send-to-groups(primary-group,[Send,m,me]);

(e) when primary-group in DD do
     a-broadcast([Change-group,primary-group]);

(f) when a-deliver([Change-group,y]) do
     if y == primary-group then
       primary-group++;
       if my-group == primary-group then
         become Transition();

(g) when coord[my-group] in D do
     a-broadcast([Change-epoch,current-epoch]);

(h) when a-deliver([Change-epoch,e]) do
     if e == current-epoch then
       coord[my-group] := (current-epoch mod size(my-group)) + 1;
       current-epoch++;
 }
```

Figure 5: Backup group behavior

```
 behavior TransitionGroup {

   private method obtain-state()
     replies := send-to-groups(G \ me,[Send-state,primary-group,me]);
     for all sets s in replies do insert(s,global);
     send-to-groups(G \ me,[Update-state,global,me]);
     a-broadcast([Update-state,global]);

   initially do
     if my-identity == coord[my-group] then
       obtain-state();

(a) when a-deliver([Update-state,mset]) do
     insert(mset,global);
     deliver();
     become PrimaryGroup();

(b) when coord[my-group] in D do
     a-broadcast([Change-epoch,current-epoch]);

(c) when a-deliver([Change-epoch,e]) do
     if e == current-epoch then
       coord[my-group] := current-epoch mod size(my-group)) + 1;
       current-epoch++;
       if coord[my-group] == my-identity then
         obtain-state();
 }
```

Figure 6: Transition group behavior

CT-broadcast and OPT-broadcast consider a single group of processes. Moreover, the optimistic assumptions made about OPT-broadcast to achieve high performance are not usually satisfied in wide-area networks. In our comparison, we consider a single group of n processes for the CT-broadcast and the OPT-broadcast protocols, and divide this group in subgroups of m processes each (i.e., we assume that n is a multiple of m) for the Hierarchical Atomic Broadcast protocol.

CT-broadcast and OPT-broadcast use an unreliable failure detector and can tolerate an infinite number of false failure suspicions. Briefly, with the CT-broadcast protocol, broadcast messages are first sent to all processes, and then the processes decide on a common delivery order for the messages. To reach a decision, processes use a Consensus algorithm based on rotating coordinator paradigm [CT96]. The OPT-broadcast makes some optimistic assumptions about the system (e.g., by taking into account the hardware characteristics of the network) to deliver messages fast. The key observation is that in some cases, there is a good probability that messages arrive at their destinations in a total order, and so, processes do not have to decide on a common delivery order. Processes have to check whether the order is the same, and if this is the case, OPT-broadcast is "cheaper" (i.e., requires fewer messages) than CT-broadcast. Otherwise, OPT-broadcast is "more expensive" than CT-broadcast.

Let g_x and g_y be two groups of processes. To compare the three protocols, we consider the number of messages exchanged between g_x and g_y to HA-Deliver some message m, and the number of communication steps between g_x and g_y to HA-Deliver m. Such a division of processes in groups has no effects on CT-broadcast and OPT-broadcast, but it allow us to highlight the strength of the Hierarchical Atomic Broadcast protocol. In all cases, we assume that message m is HA-Broadcast by some process in group g_x. For the CT-broadcast protocol, we assume that the coordinator is a process in g_x, and for the Hierarchical Atomic Broadcast protocol, we assume that g_x is the primary group. We consider executions where no process fails or is suspected to have failed.

Table 1 presents the number of messages and the number of communication steps for the broadcast protocols considered. From Table 1, it is clear that the Hierarchical Atomic Broadcast protocol performs remarkably better than CT-broadcast and OPT-broadcast in the way groups communicate. When interpreting such results, however, one should bear in mind that the Hierarchical Atomic Broadcast protocol was devised to minimize the communication between groups, and, more importantly, makes stronger assumptions about the way processes and groups fail than CT-broadcast and OPT-broadcast. Nevertheless, such assumptions are reasonable when considering how wide-area networks are used in practice, and, as presented in Table 1, lead to great performance improvements.

5 Conclusion

The key challenge in devising efficient hierarchical protocols is to be able to distinguish wide-area communication from local-area communication. The Hierarchical Atomic Broadcast proto-

Protocol	Number of messages between g_x and g_y	Number of steps between g_x and g_y
Hierarchical ABcast	2	2
CT-broadcast	$3m$	3
OPT-broadcast	$2m^2 + m$	2

Table 1: Broadcast implementations

col uses per-group coordinators to reduce the inter-group communication from many-to-many to coordinator-to-coordinator and meet such a challenge. Although this is a simple idea, its implementation is complicated because both (global) disasters and (local) coordinator failures have to be handled. Finding other ways to modularize hierarchical protocols is an interesting topic for future work.

In practice, failures and disasters are often detected in different ways. Failure detection usually relies on some timeout mechanism, whereas disaster detection usually involves human intervention. To capture and exploit this distinction, we have integrated into our model the notions of disasters and disaster detectors. The resulting model is a more faithful representation of real wide-area distributed systems. It also provides a foundation for specifying the properties of disaster-resilient protocols, such as Hierarchical Atomic Broadcast. We plan to use Hierarchical Atomic Broadcast as a modular building block for disaster recovery protocols applied to replicated databases.

References

[ADS00] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of IEEE Conference on Distributed Systems and Networks (DSN)*, June 2000.

[Agh86] G. A. Agha. *Actors, a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[CHD98] G. V. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *PODC: 17th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1998.

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

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[KSMD00] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A client-server oriented algorithm for virtually synchronous group membership in WANs. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, April 2000.

[MS95] C. Malloth and A. Schiper. View synchronous communication in large scale networks. Technical Report TR95-92, ESPRIT Basic Research Project BROADCAST, June 1995.

[Ped99] F. Pedone. *The Database State Machine and Group Communication Issues.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, December 1999. Number 2090.

[PF00] F. Pedone and Svend 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.

[RFV96] L. E. T. Rodrigues, H. Fonseca, and P. Veríssimo. Totally ordered multicast in large-scale systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 503–510, May 1996.

A Algorithm Correctness

We prove that the algorithm in Figure 3, 4, 5, and 6 is a correct implementation of Hierarchical Atomic Broadcast. We use the expression $\mathsf{set}_*(m)$ to refer to any set that contains message m.

Proposition 1 (Agreement). *If a correct process p_i in an operational group g_x HA-Delivers message m, then every correct process p_j in each operational group g_y eventually HA-Delivers message m.*

Proof (sketch): The proof builds on a simple induction on the index associated with messages. From the deliver($-$) method, no process HA-Delivers a message m before HA-Delivering every message m', such that $m'.index < m.index$. In the following we only present the inductive step of the proof, as the base step follows similarly. Assume process p_j has HA-Delivered every message m', $m'.index < m.index$. We prove that p_j HA-Delivers m. There are four cases to consider:

Case (a). Process p_i and p_j are in the Primary Group, and, therefore, do not change their behavior. To HA-Deliver m, p_i first r-delivers m, and by agreement of Reliable Broadcast, p_j eventually r-delivers m. Since p_j has HA-Delivered every message m' such that $m'.index < m.index$, p_j eventually HA-Delivers m.

Case (b). If p_i HA-Delivers m while in the Primary Group, and p_j is not in the Primary Group, p_j HA-Delivers m in a Backup Group. Notice that since p_i's group is operational, it is never suspected and, thus, no group takes the role of Transition Group. Before HA-Delivering m, p_i r-delivers m, and by uniform integrity of Reliable Broadcast, there is a process p_r that r-broadcasts m. Thus, p_r executed send-to-groups($-$, [Store, $\mathsf{set}_*(m), -$]), and from the send-to-groups method, p_r received a reply from every Backup Group that was not suspected by p_r. Since g_y is operational, p_r never suspects g_y, and receives a response from some process p_s in g_y. Process p_s only replies to message [Store, $\mathsf{set}_*(m), -$] after p_s a-delivers [Store, $\mathsf{set}_*(m), me$]. Therefore, from agreement of Atomic Broadcast, p_j also a-delivers [Store, $\mathsf{set}_*(m), -$], and it follows that p_j HA-Delivers m.

Case (c). If p_i HA-Delivers m while in a Backup Group, (c.1) p_j HA-Delivers m in a Backup Group, or (c.2) p_j HA-Delivers m in a Transition Group, or (c.3) p_j HA-Delivers m in the Primary Group. The proof is by contradiction:

(c.1) Consider initially that $g_x = g_y$. From the algorithm, p_i a-delivered [Store, $\mathsf{set}_*(m), -$], and by agreement of Atomic Broadcast, p_j also a-delivers [Store, $\mathsf{set}_*(m), -$] and eventually HA-Delivers m, a contradiction. Therefore, it must be that $g_x \neq g_y$. Process p_j is correct, and from the contradiction hypothesis, p_j does not HA-Deliver m, thus, by the *when* statements at lines (b) and (d), p_j does not a-deliver [Store, $\mathsf{set}_*(m), -$] nor does it a-deliver [Update-state, $\mathsf{set}_*(m)$]. But since g_y is operational, this can only happen if no process p_r in g_y

executes an a-broadcast with those messages. From the send-to-groups method, either there is no process in the Primary Group that executes send-to-groups($-$, [Store, $\mathsf{set}_*(m), -$]), or the Primary Group suffers a disaster failure before any process can complete the execution of the send-to-groups() method. Process p_i HA-Delivered m, and it can be proved that there exists a process in the Primary Group that executes send-to-groups($-$, [Store, $\mathsf{set}_*(m), -$]). We conclude that the Primary Group suffers a disaster failure. So, eventually all correct processes in operational groups suspect the Primary Group, and some group becomes Transition Group. Assume first that this group is not g_y. From the send-to-groups() method, either there is no process in the Transition Group that executes send-to-groups($-$, [Update-state, $\mathsf{set}_*(m)$]), or the Transition Group suffers a disaster failure before any process can complete the execution of the send-to-groups method. If the Transition Group suffers a disaster failure, it is eventually suspected and another is chosen. Since p_i's group is operational, it eventually becomes Transition Group, and so, we conclude that no process in the Transition Group executes send-to-groups($-$, [Update-state, $\mathsf{set}_*(m)$]). From the Transition Group behavior and the fact that the group is operational, if the coordinator process crashes before executing the obtain-state() method, another process becomes coordinator and executes the obtain-state() method until the end. Therefore, assume that g_y is the Transition Group (case (c.2)).

(c.2) Since p_j does not HA-Deliver m, p_j does not execute a-deliver([Update-state, $\mathsf{set}_*(m)$]). But g_y is operational, and so, there must be a process p_r in g_y that does not crash and executes a-broadcast([Update-state, $\mathsf{set}_*(m)$]). By the properties of Atomic Broadcast, p_j eventually a-delivers message ([Update-state, $\mathsf{set}_*(m)$]), a contradiction. Therefore, it has to be that p_j is in the Primary Group (case (c.3)).

(c.3) By the contradiction assumption, p_j does not HA-Deliver m, and so p_j does not execute r-deliver($\mathsf{set}_*(m)$). But since p_j's group is operational, and executes send-to-groups($-$, [Store, $\mathsf{set}_*(m), -$]), it follows that there is a correct process that executes r-broadcast($\mathsf{set}_*(m)$), which by the agreement of Reliable Broadcast contradicts the fact that p_j does not execute r-deliver($\mathsf{set}_*(m)$).

Case (d). If p_i HA-Delivers m while in a Transition Group, p_j HA-Delivers m in a Transition Group, or in a Backup Group. Assume initially that p_j is in a Transition Group. To HA-Deliver m, p_i first a-delivers [Update-state, $\mathsf{set}_*(m)$], and from the properties of Atomic Broadcast and the fact that g_y is operational, p_j eventually a-delivers [Update-state, $\mathsf{set}_*(m)$] and HA-Delivers m. Thus, assume that p_j is in a Backup Group. All messages HA-Delivered by p_i are in *global*, and before HA-Delivering m, p_i executes send-to-groups($-$, [Update-state,global, $-$]). From the send-to-groups() method, p_i received a [OK, $\mathsf{set}_*(m)$] message from some process p_r in group g_y. Before sending [OK, $\mathsf{set}_*(m)$], p_r a-delivered the message [Update-state, $\mathsf{set}_*(m), -$] and HA-Delivered m.

By the agreement of Atomic Broadcast, and the algorithm, p_j also a-delivers [Update-state, $\mathsf{set}_*(m)$] and HA-Delivers m. □

Proposition 2 (Total Order). *If a correct process p_i in an operational group g_x and a correct process p_j in an operational group g_y both HA-Deliver messages m and m', then p_i HA-Delivers m before m' if and only if p_j HA-Delivers m before m'.*

Proof (sketch): Assume, for a contradiction, that there are two processes p_i and p_j that HA-Deliver messages m and m' such that p_i HA-Delivers m before m', and p_j HA-Delivers m' before m. All messages are HA-Delivered in the deliver() method, and such that message m is only HA-Delivered if all messages that precede m, considering m's index, have been HA-Delivered. Thus, it has to be that $m.index_i < m'.index_i$ at process p_i and $m'.index_j < m.index_j$ at process p_j, which can only happen if messages m and m' are assigned indeces twice. Processes in a Primary Group assign indeces to messages in the same way, and only once, thus, we conclude that some Primary Group assigns indeces to m and m', suffers a disaster failure, and another Primary Group assigns different indeces to m and m'. Without loss of generality, we consider that m is assigned an index twice. Before becoming Primary Group, g_z was a Transition Group, and some process p_k in g_z executed the obtain-state() method. It follows that since p_i is a correct process in an operational group that has HA-Delivered m, p_k will a-deliver [Update-state, $\mathsf{set}_*(m)$], and will thus insert m into its *global* set. When becoming the Primary Group, g_z will assign the set *global* to the set *local*. But to assign an index to m, when processes in g_z execute the (b) when clause, there must be no message m' in the set *local* such that $m'.id = m.id$, a contradiction. □

Proposition 3 (Validity). *If a correct process p_i in an operational group g_x HA-Broadcasts a message m, then p_i eventually HA-Delivers m.*

Proof (sketch): There are two cases to consider:

Case (a). Assume that g_x is the Primary Group. Since g_x is operational, it remains the Primary Group. HA-broadcast(m) leads to a-broadcast([Send, m, *nil*]), and by the validity of Atomic Broadcast, p_i eventually executes a-deliver([Send, m, −]). It can be proved that there exists a process that eventually executes store-messages() after a-delivering message ([Send, m, −]), and executes r-broadcast($\mathsf{set}_*(m)$). Upon r-delivering a set that contains m, p_i HA-Delivers m.

Case (b). Assume that g_x is a Backup Group. Since p_i executes HA-Broadcast(m), p_i includes m in its *broadcast* set and executes send-to-groups(g_y, [Send, m, −]). We consider first that (b.1) g_y is operational, and then that (b.2) g_y is in-operational:

(b.1) From the send-to-groups() method, there is some process p_j in g_y that executes send([OK, $\mathsf{set}_*(m)$]) to p_i. Before doing that, p_j a-delivers message (−, [Send, m, −]), and it follows that some process p_r in g_y eventually executes send-to-groups([Store, $\mathsf{set}_*(m), -$]). Since g_x is operational, from the send-to-groups() method, some process p_s in the g_x executes send([OK, $\mathsf{set}_*(m)$]) to p_r. But before doing that, p_s a-delivered message ([Store, $\mathsf{set}_*(m), -$]) and HA-Delivered m. By the agreement property of Atomic Broadcast, p_i also a-delivers ([Store, $\mathsf{set}_*(m), -$]) and HA-Delivers m.

(b.2) For a contradiction, assume that p_i never HA-Delivers m. Consider first that some operational group g_z becomes Primary Group. Hence, some process p_j in g_z executes send-to-groups($-$, [Update-state, $global_j, -$]), and either m is in $global_j$, or m is not in $global_j$. In the former case, since p_i's group is operational, p_i eventually HA-Delivers m, a contradiction. So, consider the case where m is not in $global_j$. After receiving the message ($-$, [Update-state, $global_j, -$]), some process in g_x a-broadcasts it, and eventually p_i a-delivers ($-$, [Update-state, $global_j, -$]). Since m is not in global, when p_i executes the *for* statement in the (d) when clause, $m \in broadcast_i$, and p_i executes send-to-groups(g_z, [Send, $m, -$]), but from the fact that g_z is operational, p_i eventually HA-Delivers m. Thus, no operational group ever becomes the Primary Group, which leads to a contradiction since g_x is operational, and if all the other groups suffer disaster failures, g_x eventually becomes Primary Group. □

Proposition 4 (UNIFORM INTEGRITY). *For any message m, each process HA-Delivers m at most once, and only if m was previously HA-Broadcast by sender(m).*

PROOF (SKETCH): We first prove that messages are HA-Delivered at most once. Assume, for a contradiction, that m is HA-Delivered twice. Message m can only be HA-Delivered in the deliver() method, and only if *m.index* =*del-index*+1. After HA-Delivering m, *del-index* is incremented. Therefore, it has to be that m appears more than once in *global* with different indeces. However, from the algorithm, no message is included more than once in *global*, regardless of its index, a contradiction.

We now show that m is only HA-Delivered if it was previously HA-Broadcast by *sender*(m). Only messages in *global* are HA-Delivered, and a message m is included in *global* only by executing the insert() method. Assume first that the Primary Group never suffers a disaster failure. Thus, there are two cases to consider:

(a) If p_i is in the Primary Group when it executes insert($\mathsf{set}_*(m), global$), then p_i also a-delivers a message of the form [Send, $m, -$]. By uniform integrity of Atomic Broadcast, there is a process p_j in the Primary Group that executes a-broadcast([Send, $m, -$]). From the algorithm, there is a process p_j in the Primary Group that either executes method HA-Broadcast(m), or receives a message ([Send, $m, -$]) from some process p_k. From the algorithm, in the former case, p_j is *sender*(m), and in the latter, p_k is *sender*(m).

(b) If p_i is in a Backup Group, then p_i executes insert($\mathsf{set}_*(m), global$) after receiving message ([Store, $\mathsf{set}_*(m), -$]) from some process p_j in the Primary Group, and as shown in item (a), m has been HA-Broadcast by $sender(m)$.

Now consider that the Primary Group suffers a disaster failure, and let g_a be the first Transition Group where some process p_i in g_a executes send-to-groups($-$, [Update-state, $global_i, -$]) in the obtain-state() method. Thus, p_i executed insert(s) and received the replies from each Backup Group not suspected by p_i, with the messages HA-Delivered by each group. By the first part of the proof, all messages received by p_i have been HA-Broadcast by some process. If p_i executes insert($mset, global$) in the (a) when clause, then p_i first executes a-deliver([Update-state, $mset$]), and there is a process p_j in g_a that executed a-broadcast([Update-state, $global_j$]), such that for each message m in $global_j$ there exists a reply s received by p_j such that $m \in s$. Therefore, from the first part of the proof, m has been HA-Broadcast by some process.

Finally, if p_i is in a Backup Group and executes insert($mset, global_i$) in the (d) when clause, it has a-delivered message [Update-state, $mset, -$], and there is some process p_j in p_i's group that a-broadcast it. Process p_j received message [Update-state, $mset, -$]) from some process p_k in the Transition Group, and as shown in the previous paragraph, each message m in $mset$ has been HA-Broadcast by some process. □

Theorem 1 *Class habcast in Figure 3 solves the Hierarchical Atomic Broadcast problem.*

PROOF: Immediate from Propositions 1, 2, 3, and 4. □

B Pseudo Code

We first give an overview of the main abstraction mechanisms, and then give a more detailed description of events, values, types, statements, and expressions.

B.1 Overview

We describe our protocol as a class, and processes can then instantiate this class to generate objects that execute the behavior specified in the class. Such objects have a local state, which is described by the variables declared in a class. Objects also export a number of methods that other objects (in the same process) can call. Objects in different processes communicate through message-passing, not method invocation. Objects can also export a number of event handlers. With event handlers, we can describe object behavior as a reactive system—objects can generate events, which other objects can then react to. As for method invocations, events are only triggered within a process. Method invocation is synchronous (the caller waits) whereas event triggering is asynchronous.

Objects have access to a number of built-in methods: `send`, `a-broadcast`, and `r-broadcast`. These methods have the semantics that we outlined in Section 2. In addition, there are a number of pre-defined events that are triggered by the underlying runtime system: `receive`, `a-deliver`, `r-deliver`. These events model upcalls from the underlying communication substrate to hand-off messages to the algorithm. With these built-in facilities, we describe message-passing through combinations of calling the `send` method and reacting to `receive` events.

In our algorithm, processes may play different roles over time. For example, a process may initially be in a backup group, and then during execution become part of the primary group. For presentation simplicity, we introduce an explicit notion of role (or behavior) in our pseudo code. A behavior defines a number of methods and event handlers. These methods and event handlers capture a particular role that an object may play during its lifetime. Objects can change their role by installing a new behavior. Inspired by the Actor model of computation [Hew77, Agh86], we use a `become` primitive to describe the (atomic) change of behavior.

An object may handle a particular type of event in one behavior, but not in another. We assume that events are queued until they can be handled (if ever). In other words, we assume that events are not dropped if they cannot be delivered to an object. Calling a method that is not defined in the current behavior yields an exception.

B.2 Specifics

Events are typed values. The event value "`receive`(44)" is an event of type "`receive`" with parameter 44. We use patterns over events to match on particular event types and to bind the parameter values in an event to a local variable. For example, the pattern "`receive`(req)" matches events of type `receive` and when a match occurs, the parameter of the event is bound to the variable `req`. There are a number of built-in event types that capture reception of messages—we outlined those types above in the previous section. Besides the built-in events mentioned above, we also use events that denote expiration of timers and suspicion of groups and processes. Finally, it is possible to declare user-defined event types. For example, the declaration "`event` response(Result)" to declares an event type with name "response" that is parameterized by values of type "Result."

We describe event handlers in terms of *when* clauses. A when clause has an event pattern and a sequence of actions. For example, the clause "`when receive(req): send(req) to p`" has the event pattern "`receive(req)`" and action "`send(req) to p`." The semantics of a when clause is to execute the action sequence when reacting to an event that matches the pattern. The binding established in the pattern is visible in executing the associated statements. In the example, the name `req` would be bound to the parameter of an event while executing the `send` statement. An object reacts to events one at a time. A behavior may have a special when clause, called `initially`, which is executed when the behavior is instantiated.

To test if a `receive` event has happened, we invoke a predicate, called `received`, that takes the same parameters as an event type. For example, to test if an event of type "`receive(res)`" has happened we would invoke "`received(res)`."

We use the traditional control structures for sequential computation, such as `if`, `for`, `while`, and `repeat`, with their conventional semantics. To describe event synchronization as part of a computation, we use a `wait until` construct. This construct takes a list of event patterns, and blocks the computation until an event happens that matches one of the patterns. To explicitly create concurrency, we introduce the ability to fork new tasks (or threads). The construct "`fork task: send(req) to p`" starts a new thread to execute the `send` statement.

For statements, we use "`:=`" for assignment; for expressions we use "`==`" for equality, "`!=`" for inequality, and "`++`" to increment integer variables. In terms of data types, we use integers, sets, and arrays. We also assume data types to describe groups, process identities, requests, and results; for a given group `g`, the expression "`|g|`" denotes the cardinality of `g`. We use the normal set operations, such as set construction, set difference, union, and intersection. In addition to these operations, we employ existential quantification over sets. Messages are typed; the construct "`[Send,m,nil]i`" is a message of type "Send" with parameters "m" and "nil." The type `Message` contains all messages.