HARP: A Hierarchical Asynchronous Replication Protocol for Massively Replicated Systems

Noha Adly

Computer Laboratory
Cambridge University
Cambridge, CB2 3QG, UK
E-mail: naa13@cl.cam.ac.uk

Abstract

This paper presents a new asynchronous replication protocol that is especially suitable for wide area and mobile systems, and allows reads and writes to occur at any replica. Updates reach other replicas using a propagation scheme based on nodes organized into a logical hierarchy. The hierarchical structure enables the scheme to scale well for thousands of replicas, while ensuring reliable delivery. A new service interface is proposed that provides different levels of asynchrony, allowing strong consistency and weak consistency to be integrated into the same framework. Further, due to the hierarchical pattern of propagation, the scheme provides the ability to locate replicas that are more up-to-date than others, depending upon the needs of various applications. Also, it allows a selection from a number of reconciliation techniques based on delivery order mechanisms. Restructuring operations are provided to build and reconfigure the hierarchy dynamically without disturbing normal operations. The scheme tolerates transmission failures, node failures and network partitions.

1 Introduction

Data is replicated in distributed systems to improve system availability and performance. In recent years, the growth of internetworks and distributed applications has increased the need for large scale replicated systems. However, existing replication protocols do not address the scale and autonomy issues adequately.

Traditional approaches for managing replicated data such as the primary copy technique [Sto79] or methods based on quorum consensus [Gif79] require that read and write operations be synchronized in order to ensure that replicas are mutually consistent. Strong consistency is inadequate for large scale systems, especially wide area sytems with mobile devices. Protocols requiring synchronization among a large number of replicas are difficult to implement across internetworks. They suffer from high latency and low throughput since links tend to be slow and unreliable and a large number of replicas generate considerable traffic over the network. Further, they reduce the system availability when one or more nodes fail, or when the network is partitioned.

In contrast, weak consistency protocols provide higher availability and better response time by allowing updates and queries to occur at any replica. When link or node failures result in network partitions, replicas are allowed to diverge and continue providing service. When the partition heals, replicas merge their state and converge to a consistent state. A weak consistency approach should provide a *propagation mechanism* which ensures that updates are efficiently and reliably propagated to all replicas even if the communication network does not provide such guarantee. However, this approach is based on the assumption that the applications can tolerate some inconsistency and *reconciliation methods* should be available to resolve conflicts. Typical applications that have used weak consistency are internet news, air traffic control, air line reservation and stock exchanges.

Grapevine [SBN84] and the Global Name Service [Lam86] were among the first systems to use weak consistency. Other weak consistency protocols were presented in [DGH$^+$87, DGP90b, Gol92, LLS90, QP93, WB84]. However, these protocols assumed that any node could communicate with any other node. This assumption, though appropriate for small networks with a few replicas, is unrealistic for wide area networks like the Internet.

This paper presents a Hierarchical Asynchronous Replication Protocol (HARP) that scales well for thousands of replicas while ensuring reliable delivery. It provides an efficient propagation scheme that is based on replicas organized into a logical hierarchy where each node has to communicate with a few nodes only. A new service interface is proposed that provides asynchronous operations that commit at any replica then propagate to other replicas and synchronous operations that commit after a quorum is assembled from nodes of the top level of the hierarchy then propagate downwards. This interface gives the application the ability to select strong or weak consistency semantics. Further, due to the hierarchical pattern of propagation, the scheme provides the ability to locate replicas that are more up-to-date than others, depending upon the needs of various applications. Also, reconciliation methods based on delivery order mechanisms are provided to resolve temporary inconsistencies and an application may choose from them. The protocol provides restructuring operations that allow the hierarchy to be built and reconfigured dynamically without disturbing normal operations. It also tolerates problems arising in internetworks such as transmission failures, node failures and network partitions.

The paper is organized as follows: the next section describes the system model and assumptions made. Section 3 presents the propagation algorithm and the data structures used. In section 4, the new service interface and the reconciliation methods are described. Section 5 presents the operations to restructure the tree and discusses methods for handling failures and partitions. The following section discusses some possible optimizations. In section 7 some related work is discussed and finally conclusions and future work are presented in section 8.

2 System Model and Assumptions

The system consists of n nodes connected by an internetwork. Processors may fail then restart, however, fail-stop processors [SS83] only are assumed and byzantine failures [LSP82] do not occur. The communication network is unreliable, it may lose or duplicate messages and does not guarantee any order of delivery. The network can partition, then merge again. A node can eventually send a message to any other node by retransmitting the message if it does not receive an acknowledgment after a certain timeout period. Each node has a local clock, but clocks are not necessarily synchronized.

3 Propagation scheme

The propagation scheme is responsible for spreading updates to every node in the network while ensuring reliable delivery. The scheme should tolerate transmission and site failures. Further, it is required that link failures and partitions do not prevent a partial propagation of the updates. So, strict methods like reliable diffusion, where global error handling is done based on knowledge of the global state are unsuitable.

Classic propagation schemes fall into two categories: *flooding* techniques, where the original site sends the data to its direct neighbors, which then send the data to their direct neighbors and so on. Internet news [KL86] uses this technique to distribute updates among its thousands of replicas. The flooding approach suffers from incurring a large number of messages and lots of redundancy which causes waste in network bandwidth. The second approach is based on *probabilistic* techniques and has been used by most of the previous weak-consistency protocols [DGH$^+$87, DGP90b, Gol92, LLS90, SBN84, QP93]. These schemes rely on *delayed propagation*, where nodes exchange status and new messages periodically in sessions. Partners in these sessions are selected to include either all nodes or a subset of nodes drawn from a probabilistic function (e.g. randomly or such that communication distance is minimized). However, it has been observed that performance is mediocre due to the periodic updates. Therefore, in order to achieve an acceptable speed of propagation and to reduce the vulnerability of losing updates they had to add an unreliable broadcast on top of the periodic updates which introduced a fair amount of redundancy, causing an increase in the traffic on the network as well as communication and processing overhead. Further, both approaches assume that any node can communicate with any other node, hence, they cannot scale up smoothly for a large number of replicas.

In HARP, the propagation scheme is based on organizing the replicas into a logical, multilevel hierarchy. In this hierarchy, nodes are grouped into clusters, and clusters are organised into a tree, such that each cluster is assigned a father node in its parent cluster. Nodes in the same cluster should have efficient communication between them as well as with their father. A node i, originating a message, sends it to its neighbors, father and sons, and then waits for their acknowledgments. Each receiving node j sends an acknowledgment to the sender, then it passes the message to the next level as follows: if the message is coming from a neighbor or from the father, then j sends the message to its sons; else, if it is coming from a son then j sends it to its neighbors, its father and to its sons of clusters other than the one the message is coming from. This works recursively and a message originating at any site will eventually propagate everywhere. Figure 1 shows 40 nodes organized into a hierarchy of three levels with the path that a message would follow while propagating if it originated at node d.

The scheme scales well for a large number of replicas and is well suited to wide area networks. This is due to the fact that each node has to communicate with only a few nodes (its neighbors, father and sons). Therefore, this scheme reduces message traffic, produces low communication overhead, limits propagation delay and balances the load among the nodes. Further, the scheme ensures that each node receives a mes-

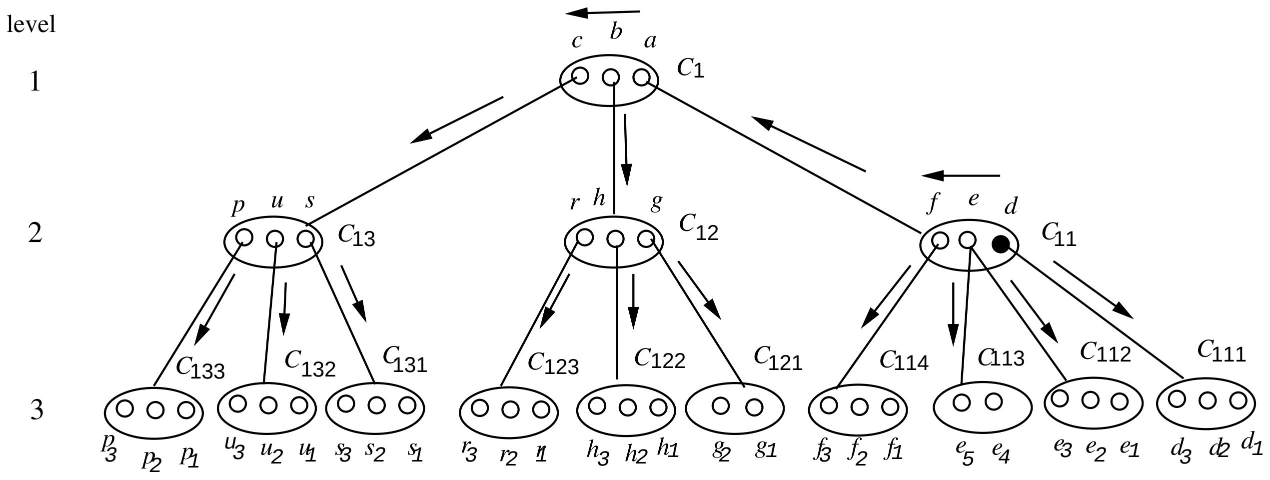

Figure 1: 40 nodes organized in a hierarchy of three levels

sage only once; hence, there is no redundancy and the network bandwidth is used efficiently. Also, the mechanism relies on nodes sending messages immediately (i.e. *immediate propagation* rather than *delayed propagation*) which speeds up the diffusion of updates.

3.1 Data structures

A node i, originating a message, assigns it a unique identifier uid=$\langle i, sequence\rangle$, where $sequence$ is an integer which is incremented by 1 to generate a new uid. Let N be the set of nodes having a replica and n the total number of nodes. Each node $i \in N$ should maintain the following:

- A Log keeping the set of messages received. When a message is received, it is added to the Log, then it is removed only if it has been reliably delivered everywhere and if it has been applied to the local database.
  Log_i = list of$(uid, message, purge_1, purge_2)$
  where $purge_1$ is a flag set to 1 if the message can be purged from the Log w.r.t. the reconciliation method (i.e. it has been applied to the local database) and $purge_2$ is a flag set to 1 if the message can be purged from the Log w.r.t. delivery (i.e. it has been already delivered to all nodes). A message can be purged from the log only when both flags are set to 1. In section 6, optimizations will be presented to show how to flush the log earlier and hence reduce its size.

- The node status which records what updates i has received so far from other nodes in the network. It consists of:

  - a Version Vector of dimension n denoted by $VV_i[node\_id]$ = $sequence$. If $VV_i[j]$ = k this means that node i has received all messages which originated at j up to message number k. Since messages are not necessarily received in FIFO order, node i should maintain:

– a Vector of Linked Lists $LL_i[k]$, for $k = 1$ to n where $LL_i[k]$ is a linked list keeping message uids received from node k out of sequence. Note that LL_i will be empty if messages are received in FIFO order.

VV and LL represent the history of what messages the replica has received. They will be used by the reconciliation methods, to determine which messages are missing from a node while restructuring, and to check for duplicate messages.

- An Acknowledgment Matrix $AckM$: used to tell what messages other nodes have received so far. It is of dimension n-1 $\times$ n and will be used to purge the log. If $AckM_i[j,u] = k$ this means that node i knows that node j has learned of all the messages that have originated at node u up to message number k. Basically, row j in $AckM_i$ is a copy of VV_j as seen by i. Again, optimizations to reduce the size of $AckM$ will be discussed in section 6.

- $View_i$ describes the view of the hierarchy as seen by i. It is a table of n rows kept in each node. Each row j includes: Cid, the cluster id j belongs to, F, the node id of j's father, Son_Cids, the set of cluster ids that are sons of j, seq, the uid of j's message that updated the row and up_flag, a flag set to 1 if the node j is up, 0 otherwise.

  $View$ = list of (row)
  row = list of($Cid, F, Son\_Cids, seq, up\_flag$)
  Son_Cids = list of(cluster ids)

Table 1 shows part of $View$ for Figure 1 as it should appear at each node. In the rest of the paper, we will use $View_i.a$ to denote the row of node a in the $View$ of node i and fields of $View$ will be referred to as follows: $View_i.a.Cid = C_1$, $View_i.e.Son\_Cids = \{C_{112}, C_{113}\}$, and so on.

Node	Cid	F	Son_Cids
a	C_1	-	C_{11}
b	C_1	-	C_{12}
c	C_1	-	C_{13}
d	C_{11}	a	C_{111}
e	C_{11}	a	C_{112}, C_{113}
f	C_{11}	a	C_{114}
d_1	C_{111}	d	-
d_2	C_{111}	d	-

Table 1 A part of the *View* data structure

These data structures are assumed to be kept on stable storage to survive failures. The data structures are summarized in Table 2.

Log_i	log of all messages received
VV_i	version vector
LL_i	vector of linked lists
$AckM_i$	acknowledgment matrix
$View_i$	structure describing the hierarchy

Table 2 Data structure kept at node i

3.2 Propagation algorithm

The propagation protocol relies on the hierarchical structure for sending messages to other nodes while ensuring eventual reliable delivery. Basically it works as follows: a node originating a message assigns it the next uid, increments its entry in VV and stores the message in the Log. Next, it sends the message to its neighbors, father and sons then waits for acknowledgments (timeouts and retransmissions are used to overcome lost messages). Each node j receiving a message checks if it is duplicate, updates its VV_j and LL_j from the uid of the message, sends an acknowledgment to the sender and inserts the message into Log_j. Next, if the message is coming from a neighbor or the father, then j sends the message to its sons; else, if it is coming from a son then j sends the message to its neighbors, its father and to its sons of clusters other than the one the message is coming from. This works recursively and a message originating at any site will eventually propagate everywhere. Further, each node after inserting a message in the Log delivers the update to the database according to the reconciliation method associated with the data item.

Steps taken by the originating node and receiving nodes are as follows:

when a node i initiates an update
. assign the update the next uid
. increment $VV_i[i]$
. insert the message into Log_i (with $purge_1 = purge_2$ = 0)
. send message to x $\forall x \in N$ s.t. $View_i.x.up\_flag = 1$ and
$View_i.x.Cid = View_i.i.Cid \neq$ null /* neighbors*/
$x = View_i.i.F$ /* father */
$View_i.x.Cid \in View_i.i.Son\_Cids$ /* Sons */
. Set timeouts to retransmit the message if acks are not received.
. perform the operation on the database if the reconciliation method allows it
and set $purge_1 = 1$

When a node j receives a propagated message m with uid =¡z,y¿ from node i:
. check if m is a duplicate: if $VV_j[z] \geq y$ or $y \in LL_j[z]$ then discard m and stop
. update $VV_j[z]$ as follows:
if $VV_j[z] + 1 = y$ then /* it is in order */
. $VV_j[z] = VV_j[z] + 1$
. /* check $LL_j[z]$ for messages that are now in the right order to increment $VV_j[z]$ */
Do while uid is in the head of $LL_j[z] = VV_j[z] + 1$
increment $VV_j[z]$
delete y from $LL_j[z]$
Endo
else /* m is out of order */
insert uid of m into $LL_j[z]$ in ascending order
endif
. send ack to i
. insert m into the Log_j (with $purge_1 = purge_2 = 0$)
. If $i = View_j.j.F$ or $View_j.i.Cid = View_j.j.Cid$
then /* i is a father or a neighbor, send m to all sons */

send m to x, $\forall x$ s.t. $View_j.x.Cid \in View_j.j.Son\_Cids \wedge View_j.x.up\_flag = 1$
. If $View_j.i.Cid \in View_j.j.Son\_Cids$
then /* i is a son, send m to neighbors, father and sons of other clusters */
send m to $x \quad \forall x \in N$ s.t. $View_j.x.up\_flag = 1$ and
$View_j.x.Cid = View_j.j.Cid \neq$ null /* neighbors*/
$x = View_j.j.F$ /* father */
$View_j.x.Cid \in View_j.j.Son\_Cids \wedge View_j.x.Cid \neq View_j.i.Cid$ /* Sons not in i's cluster */
. Set timeouts to retransmit the message if ack are not received.
. perform the operation on the database if the reconciliation method allows it
and set $purge_1 = 1$

Periodically, each node j propagates its VV (if it has changed since the last period) using the above propagation scheme. When a node i receives such a message from node j, it updates row j of its $AckM$ such that $AckM_i[j,k]$ = max $\{AckM_i[j,k], VV_j[k]\}$ for $k = 1$ to n.

To purge unneeded messages from the log, each node i summarizes what other nodes know by forming a vector min_ack such that $min\_ack[k] = min_j\{AckM_i[j,k]\}$ for $k = 1$ to n. So, if $min\_ack[k] = x$ this means that all nodes have acknowledged all messages initiated from k up to message number x. Therefore, node i can mark all messages in Log_i with $uid \leq min\_ack[k]$ for $k = 1$ to n as purgeable with respect to delivery (set $purge_2$ to 1). A message is purged from the Log when both $purge_1$ and $purge_2$ are set to 1. This ensures that messages are not deleted unless they are no longer needed by any site (including failed or isolated nodes) and they have been applied to the local database.

Theorem 1 *If any node $i \in N$ originates a message m and if no failures occur then, following the above protocol, m will be reliably delivered to every node $j \in N$ and every node will receive m only once.*

Proof: it follows directly from the pattern of the propagation described in the algorithm. □

4 Data Consistency

Different applications require different degrees of consistency. HARP supports a set of operations that provide different levels of asynchrony allowing strong and weak consistency to be integrated into the same framework. Hence, it gives the application the flexibility to tailor the service to achieve the required degree of consistency by choosing the appropriate operations to manage its data. Further, due to the hierarchical pattern of propagation, the scheme provides the ability to locate replicas that are more up-to-date than others, depending upon the needs of various applications. Also, it allows a selection from a number of reconciliation techniques based on delivery order mechanisms to resolve conflicts. In this section, these issues will be discussed.

4.1 Levels of asynchrony

A new service interface that provides different levels of asynchrony and gives the application the ability to select strong or weak consistency semantics is proposed. More specifically, the protocol supports the following operations:

- **fast_update**: A *fast_update* can be initiated and committed at any replica (the site of origin), and then propagated to other replicas using the propagation scheme.

- **fast_read**: returns the value read from any replica. It is associated with a parameter x denoting the node to read from. The default of x is the local node. If x is not the local node, then the local node should send the read request to x which should perform the read and return the read value.

- **slow_update**: a *slow_update* can be initiated by any replica; but does not commit until a quorum has been assembled from the $root$ nodes at the top of the hierarchy. For node x to perform a *slow_update*, it sends the message to any node y in the top cluster. Node y assembles a quorum from its neighbors (using majority quorum consensus or any other standard method). Once the nodes of the top cluster commit the update, they propagate it down the tree and the initiating node commits the update on receipt of the propagation message.

- **slow_read**: returns the value read from a quorum of nodes of the top cluster. A node x, wishing to perform a *slow_read*, sends the request to any node y in the top level. Node y assembles a quorum from its neighbors and returns the read value to x.

- **optimistic_update**: is similar to *slow_update*, but it is applied to the database of the site of origin as well as to some other selected replicas before committing. However, this update is subject to being $undone$ if while assembling the quorum at the top level, the decision was to abort the update. Then, every replica that has $done$ the update must $undo$ it.

Slow_update should be used by operations that are likely to result in conflict and where no conflict reconciliations are acceptable. *Slow_read*, if used in conjunction with *slow_update*, will return the most up-to-date value. *Optimistic_update* can be used by operations that conflict with very low probability; however, if conflicts occur their semantics can accept an abort. Also, it can be used instead of *slow_update* if a partition occurs, allowing operations to proceed; however, again, they should be ready to abort when the partition heals in case conflicts occur.

If an application restricts all its operations to *slow_update* and *slow_read*, then *strong consistency* (strict serializability) is achieved. It should be noticed that, by setting the number of nodes in the top level to one, this will be equivalent to the primary copy method and by setting it to n this will be equivalent to the conventional majority quorum consensus method (or whatever standard technique that will be used to reach consensus). However, by assigning the top level an adequate number of nodes, this scheme will result in better availability than the primary copy method and better latency and communication overhead than the usual quorum consensus methods since the number of nodes to synchronize with is small. Applications that require a lower degree of consistency i.e. *weak consistency* can use *fast_update*, *fast_read* and *optimistic_update*.

The advantage of having several types of read and write operations is that the more expensive ones can be reserved for special situations, while the cheaper ones may be used where slightly out of data is tolerated. The underlying assumption is that the special reads and writes would be required only rarely. Moreover, with the efficient propagation scheme used, the length of out-of-dateness is expected to be small.

4.2 Levels of Staleness

Due to the hierarchical pattern of propagation that the updates follow, the scheme provides the ability to locate replicas that are more up-to-date than others, depending upon the needs of various applications. Therefore, *fast_read* can be used to provide different levels of staleness by reading from replicas at different positions in the hierarchy. More specifically, assume h is the number of levels in the hierarchy and D is the estimated unit of time required to send a message from a node to its neighbors, father or sons. If a *fast_read* from a node at level l is used in conjunction with *slow_update*, then the returned value will be at most $D(l-1)$ time units in divergence (stale) from the most up-to-date value and will be D time units stale from nodes at level $(l-1)$. This feature gives the application the ability to provide different levels of services: querying from higher levels of the hierarchy is likely to give better service (more recent information) than querying lower levels, which may have slightly older data. If a *fast_read* is used in conjunction with a *fast_update* or an *optimistic_update*, then the answer will be at most $(h+l-1)D$ time units in divergence from any other node. This allows a node at a low level to have a better bound on the staleness of an item by querying nodes in higher levels and to get information that is at most hD stale by querying nodes at the top level. Further, each node will be at most D time units in divergence from its neighbors, father and sons. This statement can be used recursively by a node to compare itself with other nodes in its physical proximity. The usefulness of these features is application dependent. Applications in which updates of interest are local can see recent updates by reading from the local node or nodes in the local subtree, and those which need a global view should query higher levels.

For example, in [QP93] Queinnec and Padiou presented a flight plan service in a distributed air traffic control system. The size of the system, the geographical distribution of information and the high requirements of availability call for replication. The flight plan subsystem handles the flight plan data which describes a flight (model, speed characteristics, position and current speed) and its planned evolution. Its job is to plan arrivals, schedule gates and so on while guaranteeing aircraft safety, and ensuring fluidity of traffic. Performance fluctuation is crucial to this application. However, up-to-date global information is not necessary. The accurate current position of a plane that will arrive much later at some destination is not of interest to that destination. Further, the ability to update flight data is limited to a single site (or a few sites) at a time, hence, write conflict is low. Therefore, weak consistency was adopted. Controllers would like to know information about the current progress of flights which are still many hours away so that they can do their jobs. However, the further (in time) a controller is from the current position of a plane, the less precision he needs. Therefore, in order to carry out short-term planning, the needed flight plan data are those of the planes in their sectors and they may also need to consult information about the neighboring sectors. To perform long-term planning, more distant sectors need to be consulted in order to get a more global view. Such an application may benefit from HARP by reading from different levels of the hierarchy depending upon the kind of information needed. The same applies for other applications such as monitoring the status of the network, weather forecast systems and so on.

4.3 Reconciliation Methods

Fast_updates can create temporary inconsistencies because multiple users can update different replicas simultaneously. Techniques for resolving conflicts depend on the semantics and the requirements of the applications. In HARP, several reconciliation methods are provided and an application may choose from them. As in other weak consistency protocols [DGH$^+$87, DGP90b, Gol92, LLS90], the reconciliation methods are based on delivery order mechanisms. Each data item (or parts of the data base) should be associated with one of these methods according to the update semantics of the data. HARP provides reconciliation methods that support the following orders:

- **Unordered:** for some applications, all updates are commutative and associative. For example, certain data items can only be incremented or decremented. In this case, a node can deliver the update to the database as soon as it is received as the order is irrelevant. In [KS88], a good theoretical framework for commutativity in a replicated data environment has been presented that enables operations to be classified as commutative/non-commutative and it has been shown how a given a set of operations can be put into these two categories.

- **Latest-wins order:** in this method, a new update overwrites an old value and an update trying to overwrite a new version is ignored. The $View$ is one example of replicated data that uses this method: an $update\_view$ message updating row j is delivered only if it is a new update. Since by the nature of the algorithm node j only can propagate its own row then we can rely on the uid of the message to decide whether the incoming message is new or old. This method is also appropriate for other examples such as updating a mail-box of a user, or updating the state of a monitored machine and so on. In this case, if any node can update the data item then we should rely on global timestamps to decide whether the update is new or old. Grapevine [SBN84], the Global Name Service [Lam86], the Tattler system [Lon92], a distributed monitor for the internet and the CENA air traffic control system [QP93] used such a method to resolve inconsistencies.

- **FiFo order:** in this method, messages originating from the same node are delivered in order but messages from different nodes may be delivered in any order. So, a node i should deliver a message originating from node j only if all previous messages originating from node j have been delivered. This method is useful for applications where a replica has autonomous control on its data items, that is, it is the only one updating this particular data item, however, these items should be read by other replicas. With HARP, this delivery order is supported as follows: a node j receiving a message with $uid = \langle i, k\rangle$, delivers the message to the database only if $VV_j[i] + 1 = k$ otherwise inserts it into Log_j for later processing.

- **Causal order:** if two messages are causally related (one causes the other) [Lam78] then they are delivered in the same order at every node. But if not, they may be delivered in different orders. Mailing and news systems are among applications that can use this ordering as it will ensure that no user will see a reply for a message before seeing the message itself. In [L+91, LLS90], some applications were suggested that can use causal order such as garbage collection in a distributed heap, locating movable objects in distributed systems and so on. Note that causal order implies FiFo order. Causal delivery can be supported in HARP as follows:

. each node i originating a message m appends VV_i to the message.

. When a node j receives a message m for causal delivery, it compares VV_j with the VV_i included in the message and delays delivery of m until

(1) $VV_j[i] + 1 = VV_i[i]$, and

(2) $VV_j[k] \geq VV_i[k] \qquad \forall k \neq i$

The first condition ensures that j has received all messages that i has originated before m and the second condition ensures that j has received all messages that i has already received from other nodes before originating m.

- **Total order** messages are delivered in the same order at every node and causal relations between messages are preserved. This is the strongest order and it includes FiFo and causal order. Several weak consistency applications relied on total order to resolve conflicts [Gol92, LLS90]. Total delivery can be supported in HARP by treating the message as a *slow_update*: a node requiring to perform a total order update, sends the update to one of the nodes in the top level, nodes of the top level vote to agree on the sequence number (order) to give to the message then the message is propagated down the tree using the usual propagation scheme and every node has to deliver it to the database according to the sequence number assigned by the top level.

5 Restructuring the hierarchy

Restructuring operations provide mechanisms for a node to join or leave a cluster, to move from one cluster to another, to create or destroy clusters, to merge or split clusters etc. These operations are necessary to build the tree, expand it, reorganise it to maintain a high level of performance and cope with various problems such as node crashes, link congestion, partitions, and so on. HARP provides four basic operations to reorganize the structure while ensuring that no messages are lost and that the data structures are updated correctly to reflect the change. The operations are completely distributed. Further, they do not rely on synchronization with all nodes, nor on two- or three-phase commit protocols to perform the change; instead, they rely on the weak consistency approach, while ensuring no loss of information. The basic principle is that a change in one cluster does not affect the operation of other clusters and the $View$ can be considered as a replica that does not need to be seen consistently always by all nodes. For instance, if node j needs to make a change (leave/join a cluster), it does not have to wait for replies from all nodes in order to commit the change. Instead, it negotiates with the affected nodes only (nodes that j is leaving/joining). Each node x involved in the change should alter row j in its $View$ to discard/include j and send an ack to j. When these nodes acknowledge the change, then the operation is completed. Nodes in other clusters need not know about the change immediately since they are not communicating directly with j. Therefore, j can propagate its updated row, using the usual propagation scheme, and eventually everybody will be informed of the change. This feature enhances autonomy, reduces network traffic and enables dynamic reconfiguration without disturbing normal operations which make the scheme suitable for wide area and mobile systems.

In general, when a node j changes its row in $View_j$ (because it has performed one of the restructuring operations), it will propagate an $update\_view$ message containing its updated row $View_j.j$. Since messages can arrive from j in different orders, $View_x.j.seq$ will enable the latest update from j to be kept. That is, when node x receives $update\_view$ containing $View_j.j$, then x should check if $View_x.j.seq > View_j.j.seq$ then discard the message as it is an old update. Otherwise, replace $View_x.j$ by the new row as received from the message.

In this section, the four basic operations will be described along with their proofs of correctness. Next, it will be shown how these operations can be used to reconfigure the hierarchy. Finally, methods for handling failures and partitions will be discussed.

5.1 Basic Operations

HARP provides the following basic operations:

- leave: a node leaves its present cluster with its sons,
- join: a node joins a cluster with its sons,
- initiate: create a new empty cluster,
- destroy : destroy an existing cluster.

Figure 2 shows the hierarchy if node d leaves C_{11} then joins. It should be noted that the cluster names used in the figure are for clarity of exposition only and do not imply a hierarchical naming scheme.

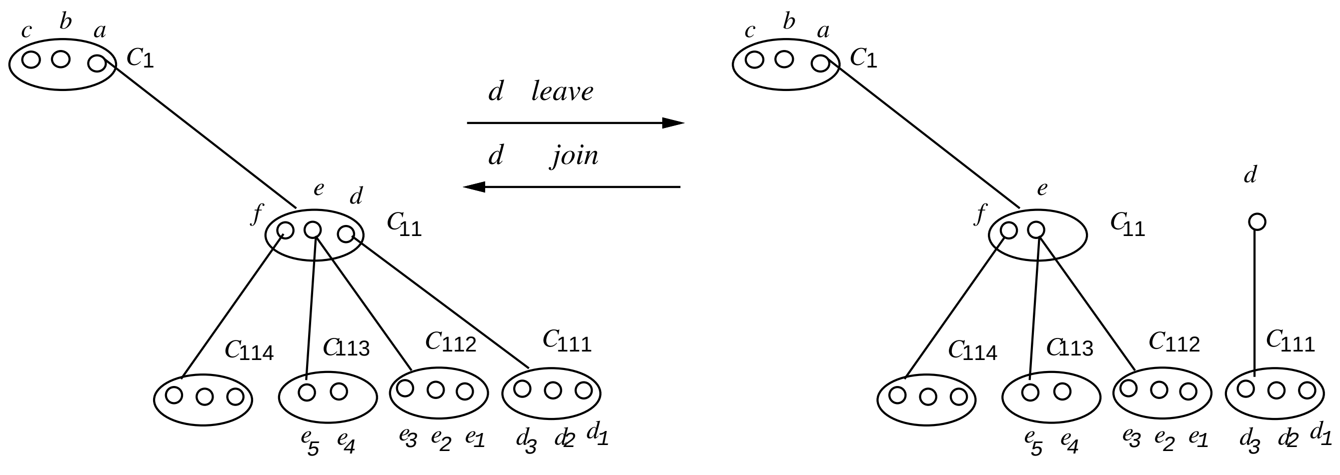

Figure 2: The hierarchical structure when node d leaves then joins

In general, node i will need to keep some of the following (temporary) lists while it is performing the change:

1- $join\_list_i$ contains set of nodes i has to join with. It is a list of (j,$flag_1$, $flag_2$) where j is the node to join with, $flag_1$ = 1 if j acknowledged the change, 0 otherwise and $flag_2$ = 1 if i has received from j the messages i asked for, 0 otherwise.

2- old_list_i contains set of nodes i has to keep in its view until they acknowledge that i is no longer in their view. It is a list of ($node, flag$) where $flag = 1$ if the node acknowledged the change.

3- $polling\_list$ is a list of messages $uids$ i has pulled from other nodes.

5.1.1 LEAVE operation

A node x leaves its cluster by executing the command leave. The leaving node should change its status to leaving, stop sending messages to its neighbors and father, wait until they have received and acknowledged every message sent by x previously then propagate an $update\_view$ message containing its updated row. Every neighbor and the father y receiving $update\_view$ from x declaring that it is leaving, should send x an ack_leave and update $View_y.x$ as received from the message. If y is leaving as well then it discards x from old_list_y. When x gets an ack_leave from the father and all neighbors then the operation is considered complete. If meanwhile x receives any messages for propagation from y such that $y \in old\_list_x$, x may either process the message and sends it to sons or just reject it. This situation can occur if y sends a message to x before receiving the $update\_view$.
The LEAVE protocol is described as follows:

Function Get_View(x, C_y**)**
/* returns list of members of cluster C_y and its father as seen by node x */
. Cl_view $= \phi$
. $\forall z \in View_x$
    if $(View_x.z.Cid = C_y \vee C_y \in View_x.z.Son\_Cids) \wedge View_x.z.up\_flag = 1$
        then Cl_view $=$ Cl_view $\cup$ z
. If $View_x.x.Cid = C_y \wedge$ status $=$ joining
    then Cl_view $=$ Cl_view $\cup$ z $\forall z \in join\_list_x$
. return(Cl_view)

Steps taken by leaving node x
. change status to leaving
. old_cluster $= View_x.x.Cid$
. /* insert neighbors and father in old_list_x */
$old\_list_x =$ Get_View($x, View_x.x.Cid$) - x
. /* stop sending messages to neighbors and father */
$View_x.x.Cid =$ null, $View_x.x.F =$ null
. $\forall y \in old\_list_x$
    wait until all messages sent to y have been acked (no timeouts are set)
. /* propagate an $update\_view$ message including x's row */
send to all y s.t. $y \in old\_list_x \vee View_x.y.Cid \in View_x.x.Son\_Cids$
    $update\_view$(leaving, old_cluster, $View_x.x$)

. when ack_leave is received from y
    if $y \in old\_list_x$ then set $flag$ in old_list_x to 1
. when $flag = 1 \quad \forall y \in old\_list_x$
    discard old_list_x
    set old_cluster = null
    change status to normal

A node y receiving $update\_view(stat, C_i, row_x)$ **from** x
. If $View_y.x.seq > row_x.seq$ then /* old update */
        discard the message and stop
. If $stat =$ leaving
    then /* x is leaving C_i */
        Case: ($View_y.y.Cid = C_i \wedge$ status = normal) $\vee C_i \in View_y.y.Son\_Cids$
            /* y is a *current* member or the *father* of C_i */
            send ack_leave to x
        Case: old_cluster $= C_i \wedge$ status = leaving
            /* y is leaving C_i as well */
            If $x \in old\_list_y$ then remove x from old_list_y
            send ack_leave to x
        Otherwise: skip
    endif
. /* update x's row in $View_y$ */
replace $View_y.x$ by row_x

Nodes leaving the top cluster decrease the quorum size of *slow_update, slow_read* and *optimistic_update* dynamically. When a node in the top cluster receives an $update\_view$ from a neighbor declaring it is leaving, it should decrease number of nodes of the cluster by one and compute the new quorum size. If a node while leaving receives a vote request, it should not send its vote in order to ensure quorum intersection.

5.1.2 JOIN operation

A node y joins a cluster C_i through the command join(C_i, z) where z is the father of C_i or any member of C_i. The joining node changes its status to joining, inserts z in $join\_list_y$ and sends z a $join\_req$. Node z replies by sending y his view of C_i (members and father) and y updates $join\_list_y$ to include additional members and sends them $join\_req$. The nodes reply by sending y their view of C_i, y merges their views with $join\_list_y$ and sends additional members (if any) a $join\_req$. This process occurs recursively until y sends $join\_req$ to all members of C_i. A member/father x, receiving $join\_req$ accepts y's membership by including y in $View_x$ as a neighbor/son and sends y ack_join including its state (VV_x and LL_x). When y receives ack_join from a member/father x, it exchanges missing messages with x by sending to x what y has and x has not and by asking x for messages that y is missing (missing messages are determined by comparing VV_x and LL_x with VV_y and LL_y). If a node x receives $join\_req$ when it has already left C_i then it sends y a nak. When y receives ack_join from the father and every member of C_i, then it propagates an $update\_view$ message including its new row $View_y.y$ informing every node that it became a member of C_i. By exchanging missing messages with every member x, it is ensured that y has received all messages other members have and that every member has all messages that y has. When y has done so with every node in $join\_list_y$ then the operation is over. If, while joining, y receives messages from x such that $x \in join\_list_y$ but y does not see x yet as a neighbor or a father in $View_y$, then y should keep these messages in a list to be processed when x replies with ack_join. This can happen if x receives $join\_req$ from y, replies with ack_join and starts propagating messages to y, however, y receives these messages before receiving the ack_join.

Definitions:
1. A node w is the *father* of C_i if $C_i \in View_w.w.Son\_Cids$
2. A node x is a *current* member of C_i if $View_x.x.Cid = C_i$ and status = normal
3. A node x is a *leaving* member of C_i if old_cluster = C_i and status = leaving
4. A node x is a *joining* member of C_i if $View_x.x.Cid = C_i$ and status = joining.

For a node y to join C_i through z i.e. it executes join(C_i, z), apart from the first node to join the hierarchy where $z =$ null, the following conditions should hold:

(C1) C_i should have a *father* or at least one *current* member while y is joining

(C2) z should be either the *father* of C_i or a *current/leaving/joining* member of C_i. Further, if z is a *joining* member and is joining through w, then w should be either the *father* of C_i or a *current/leaving* member of C_i. This condition works recursively if w is a *joining* member.

The JOIN protocol is described as follows:

Node y executing join(C_i, z)
. Change status to joining
. If $View_y.y$ already exists
    then /* keep sons only in row y */
        $View_y.y.Cid = C_i, View_y.y.F =$ null, $View_y.y.up\_flag = 1$
    else /* y is a new node */
        create a new row in $View_y$ s.t.
            $View_y.y.Cid = C_i, View_y.y.F =$ null
            $View_y.y.Son\_Cids = \phi, View_y.y.seq = 0, View_y.y.up\_flag = 1$
    endif
. /* don't send messages to any previous neighbors as this is an old view */
$\forall x \in View_y$ s.t. $View_y.x.Cid = C_i$ set $View_y.x.Cid =$ null
. if $z =$ null   /* y is the only member */
    then change status to normal and stop
    else
        $polling\_list = \phi$
        insert z in $join\_list_y$
        send z $join\_req(C_i, View_y.y)$
    endif

Node x receiving a $join\_req(C_i, row_y)$ from y
.If $View_x.x.Cid = C_i \vee$ old_cluster $= C_i \vee C_i \in View_x.x.Son\_Cids$

then /* x is a *current/joining/leaving* member of C_i or is the *father* of C_i

send ack_join including x's state, x's row and list of members

and father of C_i as seen by x */

. send y $ack\_join(VV_x, LL_x, View_x.x,$ Get_View$(x, C_i))$

. /* update row y */

If $View_x.y.seq \leq row_y.seq$ then replace $View_x.y$ by row_y

. /* if x is leaving then wait until y pulls missing messages if any */

If $View_x.x.Cid =$ null $\wedge$ old_cluster $= C_i$

then insert y in old_list_x

else /* x has already left C_i */

send y a nak

endif

Node y receiving reply from node x

Case 1: $ack\_join(VV_x, LL_x, row_x, View\_cl)$ is received from node x

. /* update $join\_list_y$ from view of x as received from the ack to include additional members if any */

$\forall z \in View\_cl$

if $z \notin join\_list_y \wedge View_y.z.Cid \neq C_i \wedge View_y.y.F \neq z$

then add z to $join\_list_y$ and send z a $join\_req(C_i, View_y.y)$

endif

. $pull = 0$

. /* update row x */

If $row_x.seq \geq View_y.x.seq$ then replace $View_y.x$ by row_x

. If $C_i \in View_y.x.Son\_Cids$ then $View_y.y.F = x$

. If $View_y.x.Cid = C_i \vee C_i \in View_y.x.Son\_Cids$

then /* x is a *current/joining* member of C_i or is the *father* of C_i */

/* exchange missing messages with x by comparing VV and LL */

. $\forall$ m_uid s.t. $m\_uid \in VV_y \wedge m\_uid \notin VV_x \vee m\_uid \in LL_y \wedge m\_uid \notin LL_x$

/* y sends x what x is missing */

retreive the message with m_uid from Log_y and send it to x

. $\forall$ m_uid s.t. $m\_uid \in VV_x \wedge m\_uid \notin VV_y \vee m\_uid \in LL_x \wedge m\_uid \notin LL_y$

/* pull from x what y is missing */

If $m\_uid \notin polling\_list$ then

pull the message with m_uid from x

add m_uid to $polling\_list$

pull $= 1$

endif

else /* x is a *leaving* member of C_i */

$\forall$ m_uid s.t. $m\_uid \in VV_x \wedge m\_uid \notin VV_y \vee m\_uid \in LL_x \wedge m\_uid \notin LL_y$

/* pull from x what y is missing */

If $m\_uid \notin polling\_list$ then

pull the message with m_uid from x

add m_uid to $polling\_list$

pull $= 1$

endif

endif

. Set $flag_1$ for node x in $join\_list_y$ to 1

. If pull = 0 then /* y hasn't pulled any messages from x */

. set $flag_2$ for node x in $join\_list_y$ to 1

. If $View_y.x.Cid$ = null then /* x is a *leaving* member */

send x an ack_leave

endif

Case 2: nak is received from node x /* x has already left C_i */

If $join\_list_y = \{x\}$

then /* y is joining through x */

retry join(C_i, x) several times, if nak is always received then return *error*

/* it was a wrong decision to join C_i through x */

else /* y should ensure to receive messages that x had before leaving */

choose any other node $z \in join\_list_y$ with $flag_1$ =1

reset $flag_1$ and $flag_2$ for z in $join\_list_y$ to 0

send $join\_req(C_i, View_y.y)$ to z

remove x from $join\_list_y$

endif

. when $flag_1$ =1 $\forall x \in join\_list_y$

/* all members have acked and have included y as a member */

then propagate $update\_view$(joining, $C_i, View_y.y$)

. when y receives pulled messages from x then

. set $flag_2$ for x in $join\_list_y$

. If $View_y.x.Cid$ = null then /* x is a *leaving* member */

send x an ack_leave

. when $flag_2$ is set $\forall x \in join\_list_y$

discard $join\_list_y$

$polling\_list = \phi$

change status to normal

It should be noted that y does not have to wait for ack_join from all members to start sending messages but if any node sends ack_join, then y can start sending it messages immediately. However, if y is in the top cluster, it should not start assembling quorums for *slow_update, slow_read* and *optimistic_update* until all members have sent ack_join so that it can compute the quorum size correctly. Further, when a node in the top cluster receives a $join\_req$ from a node joining the cluster, it should increase number of nodes of the cluster by one and compute the new quorum size. If a node while joining receives a vote request, it should send its vote only if all members have sent ack_join ($flag_1$ is set $\forall x \in join\_list_y$) in order to ensure quorum intersection.

5.1.3 INITIATE operation

The initiate operation is performed by a father to create an empty cluster as a son. Assume node x executes the command initiate(C_i), then x adds the cluster C_i to its sets of sons. That is, it sets $View_x.x.Son\_Cids = View_x.x.Son\_Cid \cup \{C_i\}$. Then, it propagates an $update\_view$ message containing its new row $View_x.x$.

5.1.4 DESTROY operation

The destroy operation is performed by a father to remove a cluster. The cluster should be empty for a deletion to be permissible. Assume node x executes the command destroy(C_i), then node x checks:

If $\exists y$ s.t.$View_x.y.Cid = C_i \wedge View_x.y.up\_flag = 1$
  then return *error*   /* cluster cannot be removed, it is not empty */
  else $View_x.x.Son\_Cids = View_x.x.Son\_Cids - \{C_i\}$
  propagate and $update\_view$ message containing its new row

Example: A possible sequence of commands that different nodes should execute to build part of the tree shown in Figure 1 is as follows:

1) i: join(C_1,-)	2) i: initiate(C_{11})	3) j: join(C_{11}, i)
4) l: join(C_{11}, i)	5) j: initiate(C_{111})	6) m: join(C_{111}, j)
7) e: join(C_{111}, m)	8) k: join(C_{111}, m)	etc ......

5.2 Correctness

In this section, the correctness of the basic operations will be established. More specifically, it will be proved that as a result of executing any of the basic operations it is guaranteed that messages are eventually reliably delivered everywhere and that the $View$ is kept consistent.

Lemma 1 *While node x is joining a cluster C_i, $join\_list_x$ is updated to include all current members of C_i and the father of C_i.*

Proof: The proof is based on the way the cluster is constructed. Assume $join\_list_x = \{z\}$ (x is executing join(C_i, z)) and w is the *father* of C_i ($C_i \in View_w.w.Son\_Cids$).

Case 1 : C_i is empty. Two cases can occur:

Case 1.1 $z =$ null then $join\_list_x = \phi$ as x is the only member in the hierarchy.

Case 1.2 $z = w$
w receiving $join\_req$ sets $View_w.x.Cid = C_i$ and replies with his view of C_i $Cl\_view_w = Get\_View(w, C_i) = \{w\}$ as there are no *current* members and the lemma holds.
Further, x receiving ack_join updates $View_x$ s.t. $C_i \in View_x.w.Son\_Cids$ then x becomes *current*. This implies that:
When a cluster has one *current* member x, then $View_w.x.Cid = C_i$ and $C_i \in View_x.w.Son\_Cids$. (1)

Case 2 : C_i contains one member y

Case 2.1 $z = w$
if y is *current* then from (1) $y \in Cl\_view_w$. Since x updates $join\_list_x$ from Cl_view_w to include additional members then $join\_list_x = \{w, y\}$ and the lemma holds.

Case 2.2 $z = y$

If y is *current* or *leaving*, from (1) $w \in Cl\_view_y$ and $join\_list_x = \{w, y\}$. If y is *joining*, from C2 y should be joining through w i.e. $w \in Cl\_view_y$ and the lemma holds.

Further, if y is *current* ($join\_list_x = \{w, y\}$) then w and y receiving a $join\_req$ from x will set $View_w.x.Cid$ and $View_y.x.Cid$ to C_i. Node x receiving ack_join from them includes them as father and neighbor then becomes *current*. This implies that:

When C_i has two *current* members x, y and a *father* w then $View_w.x.Cid = View_w.y.Cid = View_y.x.Cid = View_x.y.Cid = C_i$ and $C_i \in View_y.w.Son\_Cids$ and $C_i \in View_x.w.Son\_Cids$. (2)

Case 3 : C_i contains two members r and y

Case 3.1 $z = w$

If y (or r) are *current* members then from (1) y (or r) $\in Cl\_view_w$. If y and r are *current* then from (2) y and $r \in Cl\_view_w$ and the lemma holds.

Case 3.2 $z = r$

Case 3.2.1 r is *current* or *leaving*

If y is not *current*, then from (1) $w \in Cl\_view_r$, if y is *current* then from (2) w and $y \in Cl\_view_r$

Case 3.2.2 r is *joining*

From C2, either $w \in Cl\_view_r$ or $y \in Cl\_view_r$ given that y is *current* or *leaving*. Hence, x will send w (or y) a $join\_req$ and from (1) $y \in Cl\_view_w$ if y is *current* (or $w \in Cl\_view_y$) and the lemma holds.

Case 3.3 $z = y$

same proof as Case 3.2 replacing every r by y and vice versa.

Further, if r and y are *current* since $join\_list_x = \{w, r, y\}$ then w, r and y will set $View_w.x.Cid, View_r.x.Cid$ and $View_y.x.Cid$ respectively to C_i after receiving $join\_req$ from x. Since x becomes *current* only when it receives ack_join from w, r and y then by then it will have $View_x$ updated s.t. $View_x.r.Cid = View_x.y.Cid = C_i$ and $C_i \in View_x.w.Son\_Cids$. (3)

From (1), (2) and (3) we can generalize that from the way C_i is constructed, if C_i contains *n current* members and a *father* w then
For each *current* member y, $View_y.r.Cid = C_i\ \forall r$ *current* members and $C_i \in View_y.w.Son\_Cids$.
For the *father* w, $View_w.r.Cid = C_i\ \forall\ r$ *current* members of C_i. (4)
Therefore, while x is joining C_i through z

If z is either *current* or *leaving* or is the *father* of C_i then from (4) $y \in Cl\_view_z$ $\forall y$ *current* members and *father* of C_i.

If z is *joining*, from C2, x updates $join\_list_x$ s.t. it will contain at least one node r where r is a *leaving* or *current* member of C_i or the *father* of C_i. Since x sends a $join\_req$ to r and updates $join\_list_x$ from Cl_view_r then from (4) the lemma holds. □.

Corollary 1 *If x is a current or leaving member of C_i or is the father of C_i then the function Get_view(x, C_i) returns all current members and the father of C_i.*

Proof: Since x becomes *current* only when it receives ack_join from $y, \forall\, y \in join\_list_x$. By lemma 1, when x is *current* $View_x.y.Cid = C_i$ for every *current* member y and if the *father* is w then $C_i \in View_x.w.Son\_Cids$ then the corollary holds if x is *current*. Since x while *leaving*, changes only $View_x.x$ then its view of C_i is the same as when it was *current* only excluding x and the corollary holds. □

Lemma 2 *If two nodes x,y are leaving cluster C_i simultaneously then old_list_x and old_list_y are formed or updated such that*
2.1 $x \notin old\_list_y$
2.2 $y \notin old\_list_x$

Proof: Assume node $x(y)$
propagates an $update\_view$ declaring it is leaving at time t= t_x (t_y)
receives $update\_view$ from y (x) at t = t_x^1 (t_y^1)
receives ack_leave from y (x) at t = t_x^2 (t_y^2)
The leave can occur in two ways:

Case 1 : in serial $t_y^1 < t_y$ or $t_x^1 < t_x$

Case 1.1 $t_y^1 < t_y$ (y receives x's $update\_view$ before it sends its own) Figure 3a.
At t = t_x, from Corollary 1, $y \in old\_list_x$ and x sends y an $update\_view$. At t = t_y^1, y sets $View_y.x.Cid$ = null and sends x ack_leave. Then at t = t_y, old_list_y is formed s.t. $x \notin old\_list_y$ and 2.2 holds. At t = t_x^2, x removes y from old_list_x and 2.1 holds.

Case 1.2 $t_x^1 < t_x$ same proof as case 1.1 replacing every x by y and vice versa.

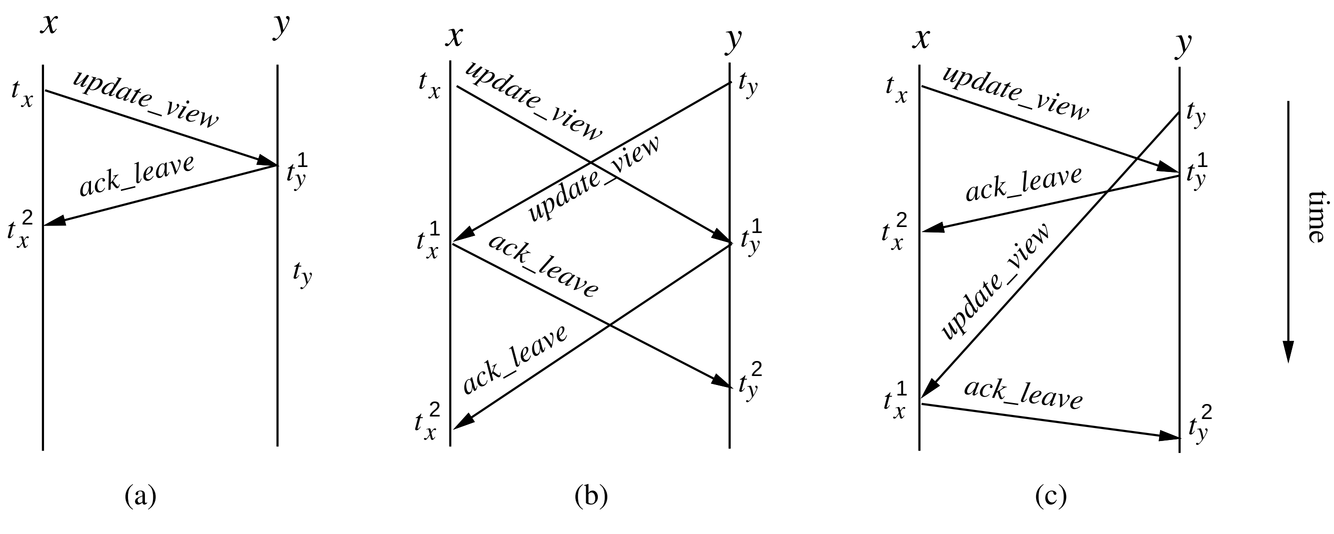

Figure 3: Two nodes leaving the same cluster simultaneously

Case 2: in parallel $t_x < t_x^1$ and $t_y < t_y^1$
Then $y \in old\_list_x$ and $x \in old\_list_y$. For node x two cases could occur:

Case 2.1 $t_x^1 < t_x^2$ (Figure 3b)

At t $= t_x^1$, x removes y from old_list_x and sends y ack_leave.

Case 2.2 $t_x^2 < t_x^1$ (Figure 3c)

At t= t_x^2, x sets $flag$ of y in old_list_x. At t= t_x^1, x discards y from old_list_x and sends y ack_leave.

Therefore, in either cases x removes y from old_list_x and 2.2 holds. For node y, the same proof applies for both cases ($t_y^1 < t_y^2$ or $t_y^2 < t_y^1$) by replacing every x by y and vice versa and in either case y will remove x from old_list_y and 2.1 is true.

Therefore, the lemma holds. □

Lemma 3 *If x is joining C_i and y is leaving C_i simultaneously then $join\_list_x$ and old_list_y are updated such that x will receive any message that y had before leaving C_i and x does not have.*

Proof: Two cases can occur:

Case 1: while x is joining, $y \in join\_list_x$. Assume:

y propagates $update\_view$ at t= t_y

y receives $join\_req$ from x at t $= t_y^1$

y receives ack_leave from z $\forall z \in old\_list_y$ at t $= t_y^2$ ($t_y^2 > t_y$)

Case 1.1 $t_y^1 < t_y^2$ ($join\_req$ reaches y while it is *leaving*) Figure 4a.

At t= t_y^1, y adds x to old_list_y and sends x ack_join. As x will send ack_leave to y only when it has pulled missing messages from y and since y will discard old_list_y and leave only when it has received ack_leave from $z \forall z \in old\_list_y$ then the lemma holds.

Case 1.2 $t_y^1 > t_y^2$ (req_join reaches y after it discarded old_list_y) Figure 4b.

At t= t_y^1, y will send a nak to x, then x updates $join\_list_x$ by replacing y by any other member z of C_i and pulls missing messages from z. Since y stopped sending messages at t= t_y and left at t= t_y^2, then at t= t_y^2, z has received every message y had at t $< t_y$. Since x pulls messages from z at t$>$ t_y^2 then x receives every message y had at t $< t_y$ and the lemma holds.

Case 2: while x is joining, $y \notin join\_list_x$ (x is joining after y has already left).

From C1 there exists at least one *current* member of C_i. From lemma 1, $join\_list_x$ contains all *current* members and send them a $join\_req$. Assume any *current* member z sends ack_join to x at t= t_z, then x updates $join\_list_x$ from z's view and as $y \notin join\_list_x$ then $View_z.y.Cid \neq C_i$ at t= t_z. This implies that y left C_i at t $< t_z$ and at t $= t_z$, z has received every message y had at t $< t_y$. As x pulls missing messages from z at t $> t_z$ then will receive every message y had at t $< t_y$ and the lemma holds. □

Theorem 2 *If node j is leaving C_i at time t_j, then*

2.1 $\forall$ messages m that j has originated or received from its sons at t$<$ t_j the current/joining members of C_i and the father of C_i will receive m.

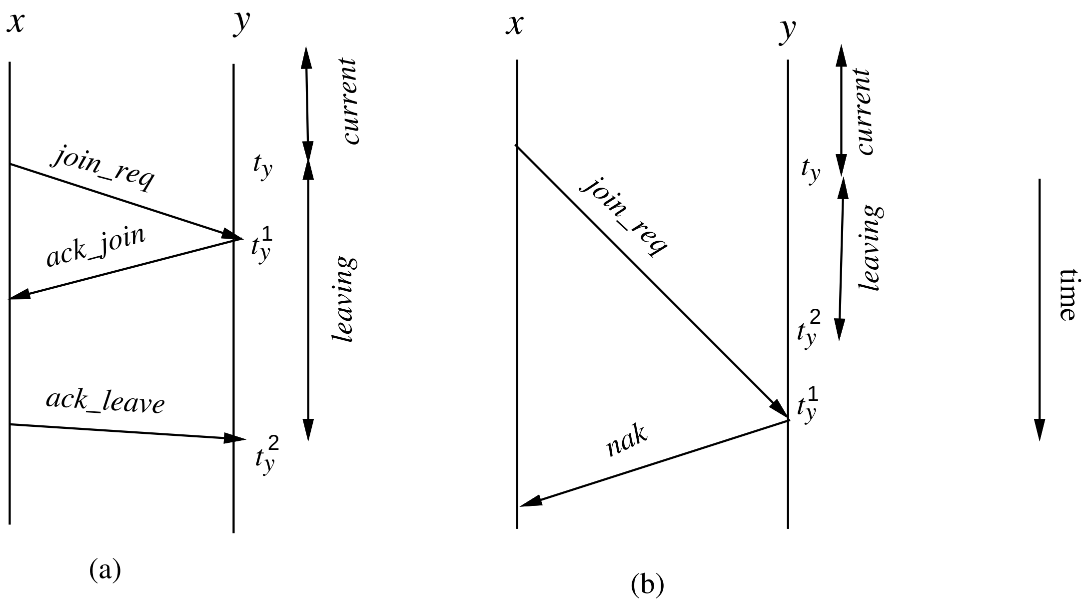

Figure 4: Node x joining a cluster and Node y leaving the cluster at the same time

2.2 Messages generated or received by j at t $\geq t_j$ *are not sent to members or father of* C_i.

2.3 After leaving, j will not receive any messages from members or father of C_i

2.4 $View_x, old\_list_x$ *and* $join\_list_x$, $\forall x \in N$ *are updated to reflect the change.*

Proof: From Corollary 1, $\forall y$ *current* members and *father* of C_i, $y \in old\_list_j$. Since before propagating $update\_view$, j waits until all messages sent to y at t $\leq t_j$ have been acknowledged $\forall y \in old\_list_j$ and since y sets $View_y.j.Cid$ to null only after receiving $update\_view$ from j then 2.1 holds for *current* members and *father* of C_i. From lemma 3 it holds for the *joining* members as well. Then 2.1 holds.

At t= t_j since j sets $View_j.j.Cid$ = null and $View_j.j.F$ = null then j will not send messages except to his sons, if any, and 2.2 holds.

$\forall y$ *current* members and *father* of C_i, y updates $View_y.j$ when it receives $update\_view$ from j such that $View_y.j.Cid$ = null then y will stop sending messages to j. $\forall y$ *joining* members of C_i, since y will receive either nak or ack_join including $View_j.j$ from j then in either case $View_j.y.Cid \neq C_i$ and y will not send any messages to j. $\forall y$ *leaving* members of C_i since they stopped sending messages to any member of C_i then 2.3 holds.

Since j's row is the only row affected in the $View$ and as $\forall y$ *current* members and *father* of C_i they will receive $update\_view$ from j and will proceed in propagating the message, then every node z in any other cluster will update $View_z.j$.

Since $\forall y \in N$, old_list_y is affected by j leaving C_i only if y is leaving C_i at the same time. From lemma 2, old_list_j and old_list_y are updated correctly.

Since $\forall x \in N$, $join\_list_x$ is affected by j leaving C_i only if x is joining C_i, lemma 3 showed how $join\_list_x$ is updated such that x will receive any messages j had before leaving and x does not have. Then 2.4 is true and the lemma holds. $\square$

Lemma 4 *If two nodes x,y are joining C_i simultaneously, then at least one of them will include the other in its join_list and will send it a req_join.*

Proof: Assume x issues the command join(C_i, z) and y issues join(C_i, w) i.e. x sends z a $join\_req$ and y sends w a $join\_req$ where z and w are *current/leaving/joining* members of C_i or *father* of C_i.

Case 1 $w = z$

Assume z receives $join\_req$ from x, y at t= t_z, t_z^1 respectively.

Case 1.1 $t_z < t_z^1$ (Figure 5)

At t= t_z, z includes x in its view setting $View_z.x.Cid = C_i$. So, at t = t_z^1, z sends y its view of C_i including x. Consequently, y adds x to $join\_list_y$ and sends x a $join\_req$ (if x has not already joined) then the lemma holds.

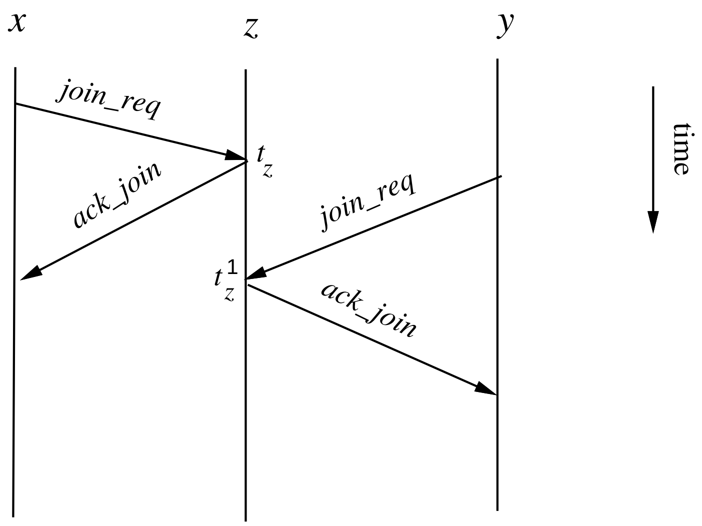

Figure 5: Nodes x and y joining the same cluster simultaneously

Case 1.2 $t_z > t_z^1$

Similarly, it can be shown that x adds y to $join\_list_x$ and sends y a req_join then the lemma holds.

Case 2 $w \neq z$

From C1, while x and y are joining there exist at least one node r s.t. r is the *father* of C_i or a *current* member of C_i. From lemma 1, $r \in join\_list_x$ and $r \in join\_list_y$. Then, x and y will send r a $join\_req$ and the same proof of case 1 applies by replacing z by r. Then the lemma holds. □

Theorem 3 *If a node x joins cluster C_i then*

3.1 x receives every message any member of C_i or the father of C_i had/will receive.

3.2 every current/joining member of C_i and the father of C_i receives every message x had/will receive.

3.3 $View_y, old\_list_y$ and $join\_list_y$ $\forall y \in N$ are updated to reflect the change.

Proof:

$\forall$ nodes y *current/joining* members or *father* of C_i receiving a req_join at t $= t_y$, y sends VV_y and LL_y and sets $View_y.x.Cid = Ci$. Therefore,

If y is a member of C_i, then $\forall$ messages that y generates or receive from his sons at t $\geq t_y$ will be sent to x (1)

If y is the *father*, then $\forall$ messages that y generates or receive from his neighbors, father or sons of other clusters than C_i at t$\geq t_y$ will be sent to x (2)

When x receives ack_join from y at t= t_y^1 ($t_y^1 > t_y$), x sets $View_x.y.Cid = C_i$ if y is a neighbor and $View_x.x.F = y$ if y is a father. Then,

$\forall$ messages that x generates or receives from its sons at t $\geq t'_y$ will be sent to y (3)
Further, x comparing VV_x and LL_x with VV_y and LL_y, x pulls from y every message y has and x has not. Since VV_y and LL_y reflect y's state at t= t_y, then

x will receive every message y had at t $\leq t_y$ (4)
And, x pushes to y every message x has and y has not, then

y will receive every message x had at t $\leq t'_y$. (5)

From (1) and (4) x receives every message a neighbor y has generated/will generate or has received/will receive from a son (6)

From (2) and (4) x receives every message the father y has generated/will generate or has received/will receive from a neighbor, father or a son of another cluster (7)

From (6) and (7), 3.1 holds $\forall$ y *current/joining* members and *father* of C_i provided y receives a req_join. From (3) and (5) $\forall y$ *current/joining* members of C_i or *father* of C_i, y will receive every message x already had and every message x generates or will receive from a son. Then 3.2 holds provided that y receives a req_join. From lemma 1 as all *current* members and *father* receive a req_join then 3.1 and 3.2 hold for *current* members and *father*. From lemma 4 since at least one of the joining nodes will include the other and sends a req_join then 3.1 and 3.2 hold for *joining* members. From lemma 3 x is guaranteed to receive any message that a *leaving* member had before leaving C_i and x did not have. Then 3.1 holds for *leaving* members.

After x receives ack_join from y, $\forall y \in join\_list_x$, x propagates an $update\_view$ message containing its new row $View_x.x$. From 3.2 every *current/joining* member and the *father* of C_i will receive the message and will proceed in propagating the message, then every node y in any other cluster will update $View_y.x$.

$\forall y \in N$, $join\_list_y$ should be updated to include x only if y is joining C_i. From lemma 4 $join_list$s are updated correctly.

$\forall y \in N$, old_list_y is affected by x joining C_i only if y is leaving C_i. Lemma 3 showed that old_list_y is updated such that x will receive every message y had before leaving. Then, 3.3 holds and this completes the proof. $\square$

5.3 Composite Operations

The four basic operations allow the hierarchy to be reconfigured and any change in the logical hierarchy can be done as a sequence of these operations. In the following, several examples of operations that allow the hierarchy to be reorganised will be described.

- **move** a node x may need to move from cluster C_i to another cluster C_j due to performance reasons e.g. if the links between x and nodes of C_i failed or became congested and communication between x and nodes of C_j are better or if a mobile computer needs to disconnect then reconnect joining another cluster that will be in its new physical proximity. The move is a sequence of leave and join operations. For example, for node f to move from C_{11} to C_{12} as shown in Figure 6 it should perform the following operations:

- leave
- join(C_{12}, g) (or join(C_{12}, h), join(C_{12}, r), join(C_{12}, b))

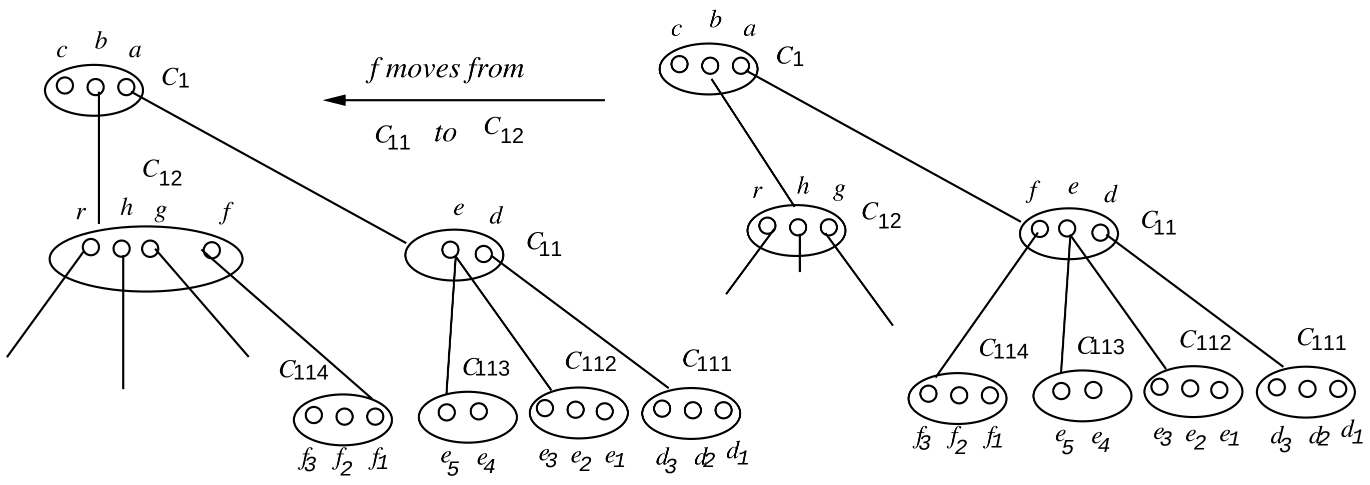

Figure 6: A node f moves from C_{11} to C_{12}

- **join-new** a new node joins the set of replicas by a simple join operation. If it is joining through node z, then z should send the new node a copy of the database, the message log Log_z, the view of the hierarchy $View_z$, as well as VV_z and LL_z.
- **merge** it merges two (or more) clusters into one e.g. to merge C_{113} into C_{114} as shown in Figure 7, then the following sequence should be executed:
  - e_4, e_5: leave
  - e_4, e_5: join(C_{114}, f) (or join(C_{114}, f_1), join(C_{114}, f_2), join(C_{114}, f_3))
  - e: destroy(C_{113})

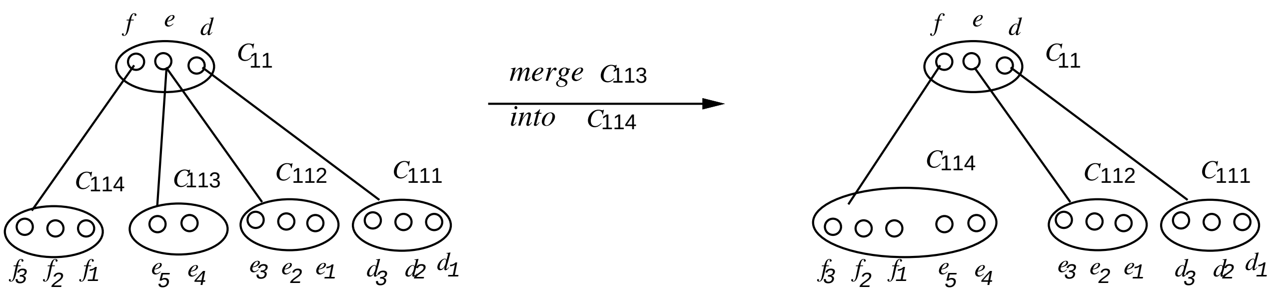

Figure 7: C_{113} merged into C_{114}

- **split** splits one cluster into two e.g. assume it is required to split C_{112} such that e_1, e_2 form another cluster C^*_{112} with d as a father as shown in Figure 8, then this is achieved by the following sequence:
  - e_1, e_2: leave
  - d : initiate(C^*_{112})
  - e_1, e_2: join(C^*_{112}, d)

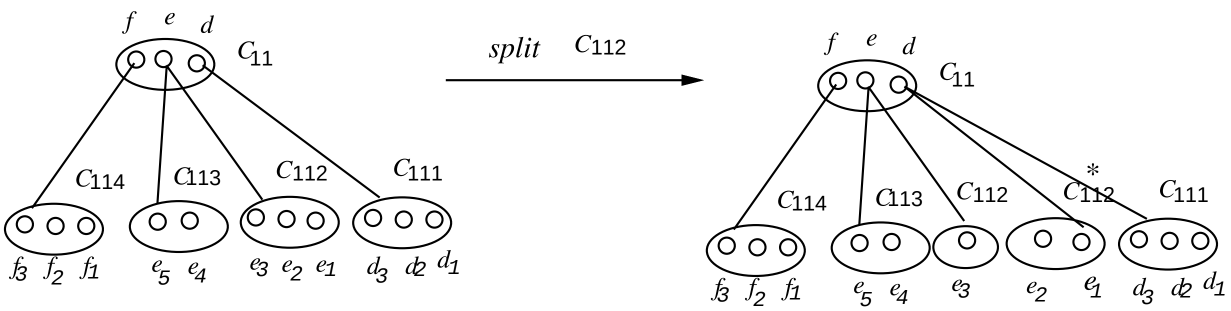

Figure 8: Split C_{112} to C_{112} and C^*_{112}

- **change-father**: assume C_{112} needs to change its father from e to d as shown in Figure 9, then the following sequence should be executed:
  - e_1, e_2, e_3: leave
  - e: destroy(C_{112})
  - d : initiate(C_{112})
  - e_1, e_2, e_3: join(C_{112}, d)

  This sequence of operations ensures correctness (no messages are lost due to the change and the data structures are updated correctly to reflect the change). However, it involves excessive overhead which could be optimized as will be discussed in section 6.

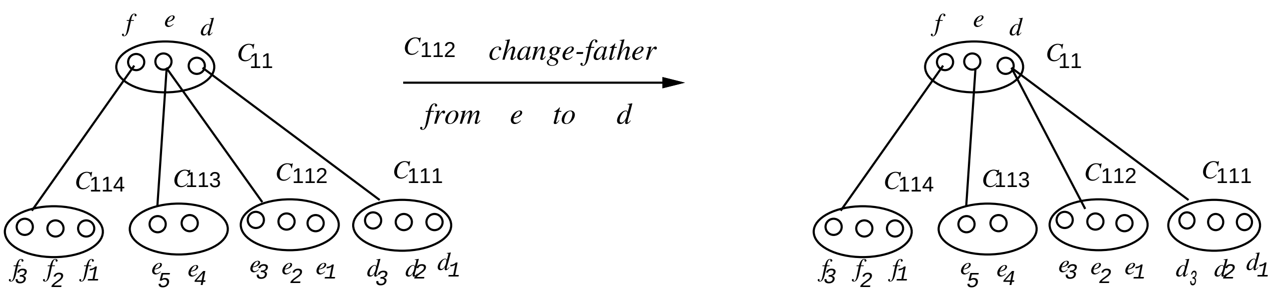

Figure 9: C_{112} changes father from e to d

- **shutdown** assume node f needs to shutdown say for repair then f's sons should change their father from f to another node say e as shown above then f should perform a simple leave.

Therefore, we can see that any operation needed to restructure the tree can be composed as a sequence of the basic operations.

5.4 Failures and partitions

When site or link failures occur, the propagation scheme continues to propagate information between connected operation sites. In this section, methods for reconfiguring the hierarchy when failures or partitions are detected and after repair will be discussed such that eventual reliable delivery is achieved.

5.4.1 Failures

It is assumed that a standard failure detection mechanism such as timeouts is available. Assume node f fails. When a node x is informed that f has failed, x should set $View_x.f.up\_flag = 0$ in order to mark that f is down and to stop sending messages to it. If $f \in join\_list_x$ or old_list_x then x should delete f from the list. Further, if f was in the top cluster, then nodes in the top cluster should decrease the number of nodes in the cluster by one and compute the new quorum size accordingly.

In order to take over f's responsibilities, two functions should be accomplished:

F1: f's sons should link with a new father, and

F2: f's neighbors, father and sons should ensure that if any of them has received a message m from f, then the others have or will receive m. They should exchange missing messages between them to avoid situations where f sent a message to a subset of them then failed. This will ensure that if f had a message m and sent it to at least one node before it fails, then m will be propagated everywhere.

These two functions can be achieved through the basic operations as follows: f's sons should perform change-father as explained above (without giving a destroy instruction to the old father). This covers F1 and the sons part of F2. Further, f's neighbors should rejoin f's cluster which is necessary to complete F2. For example, assume e will be the new father, the following sequence should be executed:

e : initiate(C_{114})

f_1, f_2, f_3: join(C_{114}, e)

e, d: join(C_{11}, a)

When a node recovers after failure, it can be integrated to the network by performing a simple join operation, joining any cluster in the hierarchy. Hence, it will receive any messages it has missed during the failure, will propagate any message it had and did not send before it fails, and everybody will be informed of its recovery as it will propagate an $update\_view$ message.

5.4.2 Partitions

In large scale systems, it is essential to consider link failures. A sequence of link failures could lead to partitions. It is assumed that an external process is responsible for detecting partitions as well as their healing.

When partitions are detected, nodes in each partition should remove the isolated nodes from their $View$ in order not to attempt sending them messages. When the partition heals, the structure can be integrated easily by having nodes in the top level of one of the partitions perform a simple join operation. For example, assume a partition occurs, isolating nodes e, d and their descendants from the rest of the hierarchy as shown in Figure 10 . When partitions are detected, nodes in P1 (a, f) will remove e and d from their $View$ by resetting their up_flag to 0, hence stopping sending them messages, and delete them from $join\_list$ and old_list if they exist in them. Similarly, nodes in P2 (e, d) will remove a and d from their $View$. After healing, nodes e and d should perform join(C_{11}, f) or join(C_{11}, a). This will ensure that any messages that were missed during the partition will be exchanged and the $View$ will be updated correctly.

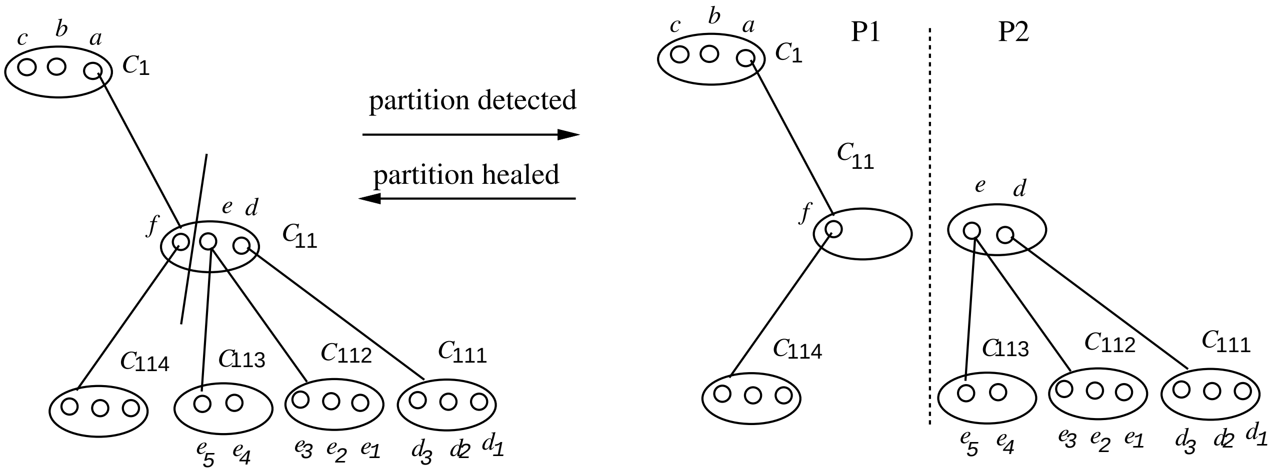

Figure 10: Partition isolating e, d and their descendants from the rest of the hierarchy

During partitioning, *fast_update* can commit and propagate within one partition. Hence, replicas are allowed to diverge while continuing to provide service. After the partition heals, missing messages will be exchanged and replicas will converge to a consistent state. Similarly, *fast_read* can proceed normally during partitions. *Slow_update* and *slow_read* can commit only if the node initiating the operation is in the *majority partition* (the partition that contains more than half of the nodes in the top cluster). *Optimistic_update* can be applied to nodes in any partition but the decision to commit or abort (and subsequently $undo$) can be known only by nodes in the *majority partition*. Nodes in other partitions will know the decision when the partition heals.

If the partition occurs in the top cluster, then nodes in the *majority partition* should update the number of replicas in the top cluster to include only the nodes they can communicate with and compute the new quorum size. Any dynamic voting algorithm such as [Dav89, JM90] can be applied to the nodes of the top cluster.

6 Optimizations

6.1 Space

- ***AckM* and *Log***

  The fact that $AckM$ has size *O(*n^2*)* may cause excessive storage. Here, two separate optimizations can be made:

  1. Since a node x sends messages to nodes in its neighborhood only (sons, neighbors and father) then instead of discarding a message from the Log only when the message has been received by *all* nodes in the network, it can discard the message when nodes in its neighborhood (and perhaps also their neighborhood) have acknowledged the message. This optimization will enable x to flush Log_x more quickly, hence reducing its size. Further, the number of rows in $AckM_x$ will be reduced to include only nodes around x (e.g. nodes in x's cluster, in x's father's cluster and nodes in the clusters of sons of x's neighbors) instead of $n-1$ rows. However, this optimization will impose a restriction on reconfiguring the hierarchy. Basically, a node x cannot move to other clusters freely as nodes of the new cluster might not have in their Log messages that x is missing. Similarly, a node after recovering, has to join the same cluster it was affiliated with before it failed as its nodes are the only ones that are keeping the messages the node has missed when it was down. For instance, node e_1 in Figure 1 cannot move and be a son of p, or if e fails and recovers it cannot join C_{12} instead of C_{11}. These kinds of movement are possible if we consider mobile computers or if the connection between e and p becomes better than that between e and d (e.g. due to link failure, or network congestion). Without this optimization it is feasible to support such reconfiguration as p will have in its Log_p messages that e has not received yet and will pass them to e while joining. However, they cannot be supported if the optimization is adopted. Hence, there is a tradeoff between space and flexibility.
  2. If clocks are synchronized (i.e. global timestamps can be used) then the matrix can be optimized to a vector $AckV$ of dimension 1 x n. When node i initiates a message, it assigns it a $timestamp$ from the local clock. $AckV_i[j] = ts$ means that node i knows that j has received every message from any sender with $timestamp \leq ts$. In this case, periodically, each node k instead of propagating VV_k, should propagate the time t such that it has received every message from any sender up to time t. When i receives such message it updates its entry k of its vector $AckV_i$. Then, node i can purge a message from the log if its $timestamp \leq$ minimum entry in $AckV[k]$ $\forall k$. Again optimization 1 can apply here and the number of columns in $AckV$ can be reduced.

- **The *View***

  Although each node is keeping the view of the whole tree, this is not necessary for the correctness of the operations. A node can keep rows of itself, its neighbors, its sons only and forget about the other nodes. This will reduce the size of *View* while all operations are still correct, however, each node will not have any idea about the structure of the rest of the network.

6.2 Overhead

The non-basic restructuring operations are correct since they only consist of a sequence of the four basic operations. However, some optimizations can be achieved by writing each composite operation as an independent operation with the expense of adding more code. For instance, change-father could be optimized by having the new sons negotiating with the old and new father only to change the view and exchanging missing messages rather than leaving then joining the same cluster. Similarly, assume C_i needs to merge into C_j. The nodes of the merging cluster C_i could skip joining and exchanging missing messages with each other as they were already in the same cluster and they can form their $join\_list$ to include members of C_j only.

Also, the method for handling failures is correct as it includes only basic operations but it could be optimized by writing a separate operation for take-over by having a single node (say the new father) performing F2 by exchanging missing messages with j's neighbors, father and sons merged with change-father operation. Although, this is more complex but it can save the neighbors from rejoining the cluster and exchanging messages with each other.

7 Related work

Due to space limitiations, only a few representative examples of the many papers related to this topic will be listed.

Grapevine [SBN84] and Clearinghouse [OD83] are early examples of replicated directory systems that achieve eventual consistency. Copies of the databases are exchanged and merged periodically. The *latest-wins* order was used to resolve conflicting updates. To speed up propagation, unreliable *direct mail* was used to send an update, upon initiation, to all other nodes. The periodic updates imposed a large load on the network since each comparison involved sending a complete copy of the database. Another example is the Global Name Service [Lam86], where updates are done at one copy and propagation is done through a *sweep* operation moving deterministically around servers which are organised in a logical ring. Periodically, the sweep operation visits every replica, collects a complete set of updates then writes this set back to every replica. A problem with this scheme is that dynamic configuration of the ring was not supported. Further, the sweep operation was expensive.

In [DGH$^+$87], several randomized algorithms based on epidemiology theory are described for distributing updates asynchronously. They are intended to maintain a widely replicated directory or name-lookup database. Three epidemic communication methods are specified: *direct mail, rumor mongering*, and *anti-entropy*. Direct mail propagates an update to other replicas using a single unreliable multicast datagram. A replica can use rumor mongering to send recent updates to other replicas selected randomly, again using unreliable datagrams. Periodically, pairs of replicas perform a reliable exchange of database contents in an anti-entropy session which ensures that updates will eventually propagate to all sites. The techniques can be combined, using direct mail or rumor mongering for fast, unreliable propagation, while anti-entropy provides a reliable backup if the other methods fail. However, anti-entropy is an expensive procedure since it involves comparing the contents of two copies of the database, one of them sent over the network.

Wuu and Bernstein in [WB84], present a propagation scheme for maintaining a replicated dictionary using logs. They use a two dimensional table (2DDT) of size On^2 similar to the union of $AckM$ and VV which allows only missing updates to be exchanged rather than the whole copy of the database. Periodically, each node x sends to every node y a message containing its 2DDT and all messages in its log that x believes y does not have. Node y, on receiving this message, extracts the new messages and updates its 2DDT by merging it with the incoming one, taking the component-wise maximum of each. Again, only *latest-wins* order is supported. However, redundant messages are still sent as 2DDT is just an approximate view of what other nodes have. Also, the scheme does not scale very well and the speed of propagation is very sensitive to the activation period. In [HHW89], Heddaya, Hsu and Weihl present a very similar approach, the *two phase gossip* protocol. Messages are propagated in the same way but the status stored and sent on every message is reduced by a factor of n. More precisely, instead of storing n^2 timestamps, each replica has to store two vectors of size n. However, this optimization is only feasible provided that clocks are synchronized and global timestamps can be used. Also, in comparison with [WB84], this scheme incurs a larger number of duplicate messages and causes the log to be purged less quickly, hence, it is larger in size.

In [Gol92], Golding proposes a weak consistency group mechanism called *timestamped anti-entropy* (TSAE) which is based on the epidemic anti-entropy protocol [DGH$^+$87] and uses a set of data structures much like those in [WB84] and [HHW89] to exchange updates efficiently. Each replica keeps a *summary vector* containing the latest message timestamps it has received from other replicas. Periodically, a replica selects a partner at random and starts an anti-entropy session. First, they exchange their summary vectors, then the missing messages only are exchanged. A best-effort multicast is combined with the anti-entropy scheme to speed propagation. FIFO and total (but non-causal) delivery orderings are provided to resolve conflicts. Group membership protocols are described to allow a node to join and leave the group of replicas dynamically. The *Refdms* [Wil91], a replicated bibliographic database, and the Tattler system [Lon92], a distributed monitor for the internet, use the TSAE protocol. The scheme cannot scale to a large number of replicas since it is assumed that any node can communicate with any other node. Further, it involves a fair amount of redundancy especially with the combination of the best-effort multicast.

Liskov and Ladin in [LL86a] describe the *lazy replication* method, where updates take place at one replica, then replicas *lazily* exchange new information through gossip messages. They rely on *multi-part timestamps* similar to the 2DDT used in [WB84]. Periodically, each replica sends a gossip message containing its timestamp and its state (message log) to *all* other replicas and replicas are made consistent by merging the states and timestamps. Total and causal orders are suported by relying on a central node generating the sequence order. In [L$^+$91, LLS90, LLSG92], an extension is made that allows clients to define a partial order indicating the causal order of operation calls, so weaker orderings can be specified and the service is responsible for scheduling operation execution to respect that order. This method is suitable for applications such as garbage collection in a distributed heap [LL86b, Lad89], and locating movable objects in distributed systems [Hwa88]. However, *lazy replication* does not take full advantage of the information available in its timestamps and sends a complete copy of the message log in a gossip message which increases the communication and processing overhead. Also, it does not scale very well as every node has to communicate with *all* nodes. Further, it needs an extra log, where messages are kept temporarily to prevent executing duplicate updates, using excessive storage. Finally, the scheme suffers from a slow speed of propagation due to the periodic updates.

In [TD92], Tait and Duchamp introduce a method for managing a replicated file system with two goals: minimizing synchronous operations and letting clients determine the level of consistency desired. Their model is that of a primary-secondary server organization. To perform an update, a client just writes the update in its cache and continues execution. The primary performs periodic *pickups* from the client's cache and propagates the updates to the secondaries. To read a file, the interface provides a *loose-read* (equivalent to *fast_read*) and *strict-read* which returns "the most recent value". However, the strict-read requires synchronization with at least the majority of the secondaries as well as with all caches, hence, it results in mediocre performance. Further, to ensure its correctness *every* write should be preceded by a strict-read to update a fresh copy of the file which incurs a large overhead and the benefits of the asynchrony are lost. An optimization is offered by giving *currency tokens* to strict readers to reduce the cost of strict-read, but this optimization is beneficial only if writes are rare.

OSCAR [DGP90a, DGP90b] implements weak consistency replication using a mixture of distributed and centralized elements. The architecture is based on two cooperating agents, called replicators and mediators. When an update is initiated, the local replicator sends the message to all other nodes using an unreliable multicast. Periodically, a master mediator polls every replicator, obtaining a version vector for every database item. The version vectors from different replicas are combined, and the result is multicast to every replicator. Replicators use this vector to detect messages they have missed and ask for them. The protocol supports FIFO order and *latest-wins* order to resolve conflicts. A negative side of the scheme is that the mediator may become a bottleneck, cause huge network traffic and prohibit scalability.

In [ABGM88, ABGM90], the notion of *quasi-copies* is introduced. Quasi-copies are out-of-date replicas that are allowed to diverge in a controlled fashion by taking advantage of the application semantics. Users define how a quasi-copy is managed by specifying a predicate based on a time, version or arithmetic condition. A central location exists, where *all* updates are processed. The central node constantly watches for updates and is responsible for propagating them when the predicate is about to be violated. In [BGM90], Barbara and Garcia-Molina extend the model to involve multiple central sites where the database is divided into segments, each one controlled by a single node. However, the central components present a handicap for scalability. Also, the bound of consistency given is only approximate since the possible occurrence of updates during the transmission delay from the central node to the quasi-copies was not taken into account.

Pu and Leff in [PL91b], present an asynchronous replication approach that applies an extension of serializability called *epsilon-serializability* (ESR) [PL91a]. ESR is a correctness criterion which allows temporary and bounded inconsistency in replicas to be seen by queries. Four divergence control methods were presented that use operation semantics to increase concurrency. The *ordered updates* method executes update transactions in total order, however, a reliable and ordered message delivery was not provided and was assumed to be available. The *commutative* method is limited to commutative updates, and the *read-independent timestamp updates* method supports operations that either produce immutable versions or overwrite older versions. These methods are supported by HARP's ordering methods: total, unordered and latest-wins order respectively. The *backward* method is based on compensation, where operations are optimistically allowed to execute in any order, then uses rollbacks to undo the effects of transactions if inconsistencies are detected later. This method is similar to the *optimistic-update* in HARP. Queries may interleave any other transactions, hence they may see intermediate results of update transactions. Methods for bounding the amount of inconsistency seen by queries were presented. The divergence was measured by the number of concurrent update transactions with which the queries interleave. Unlike the mechanisms described previously, including HARP, this model is concerned with transactions where several operations must be executed as a group. Although updates occur at one replica and propagate asynchronously to other replicas, it was not specified how propagation occurs nor how eventual reliability is reached. Therefore, the *espsilon transactions* described could be considered as a high-level interface to methods such as HARP, OSCAR and so on.

The Coda distributed file system [SKK$^+$90, KS92] uses an optimistic replication protocol to manage distributed file services and support disconnected operations. It uses a synchronous *read-one-write-all* strategy during normal operation and allows operations to occur asynchronously only if partitions occur. Synchronization among all sites is necessary to detect conflicts once the partition heals. It provides semantic-dependent rules for automatic recovery of specific applications such as directories [KS91]. The scheme aims at improving availability of the system in the face of partition rather than reducing access time and it does not address the issue of scalability.

Isis [BJ87, BSS91] is a distributed programming toolkit that provides atomic, interactive delivery with total or causal message ordering. It is based on virtual synchronous process groups and has been used to develop a variety of applications including replicated file systems. However, Isis is aimed for small systems and ensures strong consistency of group views at the expense of latency and communication overhead.

8 Conclusions and future work

In this paper, a hierarchical asynchronous protocol for managing massively replicated data was presented. It provides an efficient propagation scheme that is based on organizing nodes into a logical hierarchy. The scheme has the following features:

- Each node has to communicate with only few nodes, typically ones in its physical proximity. This localized communication produces low message traffic, improves latency and balances the load among the nodes which makes the scheme scalable and autonomous.

- The propagation scheme ensures reliable delivery with no redundancy which makes efficient use of the network bandwidth. Further, the scheme relies on immediate propagation, rather than delayed propagation, which speeds up the diffusion of updates.

- The hierarchical structure provides the ability to provide strong consistency by assembling quorums from nodes of the top cluster of the hierarchy. These operations are not expensive since they need to synchronize with a small number of nodes.
- A new service interface is proposed that provides different levels of asynchrony which gives the application the ability to select strong or weak consistency semantics.
- Due to the hierarchical pattern of propagation, the scheme provides the ability to locate replicas that are more up-to-date than others, hence, it provides different levels of staleness. This feature gives the application the ability to provide different levels of service.
- It allows the application to select from a number of reconciliation methods based on delivery order mechanisms to resolve conflicts.
- Dynamic restructuring operations were presented which allow the hierarchy to be built and reconfigured, including the restarting of failed nodes and remerging partitioned networks, without disturbing normal operations. The scheme is therefore suitable for wide area and mobile systems.

In future work, it is intended to:

- optimize some of the relevant non-basic operations such as change-father and take-over.
- model and evaluate the performance of HARP and compare it with other synchronous and asynchronous approaches.

Acknowledgments

I wish to thank Dr. Jean Bacon, my supervisor, for her support and suggestions. I am grateful to Prof. Magdy Nagi of Alexandria University, who assisted with the initial development of the protocol and gave many invaluable suggestions during the course of this work. Also, I would like to thank Prof. Akhil Kumar of Cornell University for his constructive comments and useful discussions of the issues presented in this paper. I have been supported by an FCO scholarship from the British Council.

References

[ABGM88] R. Alonso, D. Barbara, and H. Garcia-Molina. Quasi-copies: efficient data sharing for information retrieval systems. In *Proceedings of the 2nd International Conference on Extending Data Base Technology*, Venice, Italy, March 1988.

[ABGM90] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM TODS*, 15(4), September 1990.

[BGM90] D. Barbara and H. Garcia-Molina. The case for controlled inconsistency in replicated data. In *Proceedings of the Workshop on Management of Replicated Data (Houston, Texas)*, pages 35–38, November 1990.

[BJ87] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):272–314, February 1987.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):47–76, August 1991.

[Dav89] D. Davcev. A dynamic voting scheme in distributed systems. *IEEE transactions on Software Engineering*, 15(1):93–97, January 1989.

[DGH+87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Symposium on Principles of Distributed Computing, ACM SIGACT-SIGOPS*, pages 1–12, 1987.

[DGP90a] A. Downing, I. Greenberg, and J. Peha. OSCAR: a System for Weak Consistency Replication. In *Proceedings of the Workshop on Management of Replicated Data (Houston, Texas)*, pages 26–30, November 1990.

[DGP90b] A. Downing, I. Greenberg, and J. Peha. OSCAR: an Architecture for Weak Consistency Replication. In *Proceedings of IEEE PARABASE-90*, pages 350–358, March 1990.

[Gif79] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th Symposium on Operating System Principles*, December 1979.

[Gol92] R. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California, Santa Cruz, 1992.

[HHW89] A. Heddaya, M. Hsu, and W. Weihl. Two Phase Gossip: Managing Distributed Event Histories. *Information Sciences*, 49(1):35–57, 1989.

[Hwa88] D. Hwang. Constructing a highly-available location service for a distributed environment. Technical Report MIT/LCS/TR-410, M.I.T. Laboratory for Computer Science, Cambridge, MA, January 1988.

[JM90] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining consistency of a database. *ACM TODS*, 15(2), June 1990.

[KL86] B. Kantor and P. Lapsley. Network news transfer protocol - a proposed standard for the stream-based transmission of news. Internet Request for Comments, RFC 977, February 1986.

[KS88] A. Kumar and M. Stonebraker. Semantic Based Transaction Management Techniques for Replicated Data. In *Proceedings ACM SIGMOD Conference, Chicago*, June 1988.

[KS91] P. Kumar and M. Satayanarayanan. Log-based directory resolution in the Coda file system. Technical Report CMU-CS-91-164, School of Computer Science, Carnegie Mellon University, 1991.

[KS92] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the CODA File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[L^{+}91] Barbara Liskov et al. Lazy Replication: Exploiting the Semantics of Distributed Systems. *ACM Operating Systems Review*, 25(1), January 1991.

[Lad89] R. Ladin. *A method for constructing highly available services and an algorithm for distributed garbage collection*. PhD thesis, M.I.T. Laboratory for Computer Science, Cambridge, MA, 1989.

[Lam78] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[Lam86] B. Lampson. Designing a global name service. In *Proceedings of the 5th Symposium on Principles of Distributed Computing, ACM SIGACT-SIGOPS*, pages 1–10, 1986.

[LL86a] B. Liskov and R. Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th Symposium on Principles of Distributed Computing, ACM SIGACT-SIGOPS*, pages 29–39, Vancouver, CA, August 1986.

[LL86b] B. Liskov and R. Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, Calgary, Alberta, Canada, August 1986.

[LLS90] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: exploiting the semantics of distributed services. In *Proceedings of the 9th ACM symposium on Principles of Distributed Computing*, pages 43–57, Quebec City, CA, August 1990.

[LLSG92] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[Lon92] D. Long. A replicated monitoring tool. In *Proceedings of the 2nd Workshop on the Management of Replicated Data*, Montery, California, November 1992.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3), July 1982.

[OD83] D. Oppen and Y. Dalal. The Clearinghouse: a decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems*, 1(3), July 1983.

[PL91a] C. Pu and A. Leff. Epsilon-Serializability. Technical Report CUCS-054-90, Columbia University, New York, NY 10027, January 1991.

[PL91b] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. Technical Report CUCS-053-90, Columbia University, New York, NY 10027, January 1991.

[QP93] P. Queinnec and G. Padiou. Flight plan management in a distributed air traffic control system. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, Kawasaki, Japan, March 1993.

[SBN84] M. Schroeder, A. Birrell, and R. Needham. Experience with Grapevine. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[SKK+90] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), April 1990.

[SS83] R. Schlichting and R. Schneider. Fail-Stop Processors: an approach to designing fault-tolerant computing systems. *IEEE Transactions on Computer Systems*, 1(3), April 1983.

[Sto79] M. Stonebraker. Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES. *IEEE Transactions on Software Engineering*, 5(3):188–194, March 1979.

[TD92] C. Tait and D. Duchamp. An efficient variable-consistency replicated file service. In *Usenix File System Workshop Proceedings*, Michigan, May 1992.

[WB84] G. Wuu and A. Bernstein. Efficient Solutions to the Replicated Log and Dictionary Problems. In *Proceedings of the third ACM Symposium on Principles of Distributed Computing*, pages 233–242, August 1984.

[Wil91] J. Wilkes. The Refdbms bibliographic database user guide and reference manual. Technical Report HPL-CSP-91-11, Hewlett-Packard Laboratories, May 1991.