Technical Report

UCAM-CL-TR-331
ISSN 1476-2986

Number 331

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

HPP: a hierarchical propagation protocol for large scale replication in wide area networks

Noha Adly, Akhil Kumar

March 1994

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

HPP: A Hierarchical Propagation Protocol for Large Scale Replication in Wide Area Networks

Noha Adly
Computer Laboratory
Cambridge University
Cambridge CB2 3QG, UK.

Akhil Kumar
Graduate School of Management
Cornell University
Ithaca, NY 14853, USA.

Abstract

This paper describes a fast, reliable, scalable and efficient propagation protocol for weak-consistency replica management. This protocol can be used to implement a bulletin board service such as the Usenet news on the Internet. It is based on organizing the nodes in a network into a logical hierarchy, and maintaining a limited amount of state information at each node. It ensures that messages are not lost due to failures or partitions once they are repaired and minimizes redundancy. Further, the protocol allows messages to be diffused while nodes are down provided the parent and child nodes of a failed node are alive. Moreover, the protocol also allows nodes to be moved in the logical hierarchy, and the network to be restructured dynamically in order to improve performance while still ensuring that no messages are lost while the switch takes place and without disturbing normal operation.

1 Introduction

Data is replicated in distributed systems to improve system availability and performance. There are two approaches for managing replicated data: *synchronous* protocols that enforce strict serializability by means of quorums [Gif79, CAA90, Kum91], and *asynchronous* protocols where updates and queries can occur at any replica. Synchronous protocols are impractical for large networks as 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. Asynchronous protocols would provide higher availability and give better response time. Such an approach should provide a *propagation scheme* which ensures that updates are efficiently and reliably propagated to all replicas even if the communication network does not provide such a guarantee. However, this approach is based on the assumption that the applications can tolerate some inconsistency and reconciliation methods are available to resolve conflicts.

The semantics of some applications are such that they do not require strict serializability, and weaker forms of consistency are adequate and acceptable. For instance, in a bulletin board application, the main consistency requirements are that messages generated by a node must be seen by all other nodes in the same order they were generated, and if a single node receives a message and posts a response or a follow-up message, it should be seen by all other nodes after the original message to which the follow-up relates. Hence, what is required is a method that can asynchronously propagate messages generated at any node to all other nodes in a network while respecting the above ordering constraints. In a large network such as the Internet with nearly 2 million host computers, the challenge lies in ensuring that such asynchronous propagation is fast, reliable and scalable. Other applications that have used weak consistency are air traffic control [QP93], resource discovery systems e.g. archie [ED92], stock exchanges and so on.

In this paper we describe a propagation protocol for managing replicated data asynchronously. This method is especially suited for the bulletin board application mentioned above. 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, DGP90, Gol92, LLS90, QP93, WB84]. These protocols are useful and interesting; however, they assume that while propagating messages any pair of nodes can communicate with one another as easily as any other pair of nodes in the network. This assumption, though appropriate for small networks with a few replicas, is unrealistic for wide area networks like the Internet. Moreover, most of these proposals involve redundancy during normal operation in varying degrees, and, thus increase communications overhead and waste network bandwidth. Our scheme is based on a logically hierarchical arrangement of the nodes in a network, such that pairs of nodes with faster communication between them are nearer to one another in the hierarchy. It does not involve redundant messages to be sent during normal conditions; and, by maintaining minimal state information, it minimizes redundancy in a novel manner in case failures or restructuring occur.

Hierarchies are a natural and logical way of organizing a group of nodes for message propagation. The root node of the hierarchy could send a message to each of its child nodes; and, these nodes in turn would propagate the message to their child nodes. In this way, every node in the network would receive the message. A message generated at any other node (which is not the root node) of the hierarchy could also be propagated similarly. In this case, the originating node would send the message to its child nodes and the parent node, and each receiving node would further send the message to its parent and child nodes, except the node from where it came. The nice property of such a propagation scheme is that it is scalable, because each node sends a message to only a few other nodes, and the burden of propagation is distributed quite uniformly throughout the network. Another factor which makes this scheme scalable is that each node needs to keep information only about its own parent and child nodes, and need not know anything about the rest of the network. It is also efficient because the communications overhead is low and can be reduced even further by batching messages together. Moreover, it is fast because, if N nodes are arranged in

a hierarchy such that the root and each successive non-leaf node has n child nodes, then there would be $\log_n N$ levels in the hierarchy, and even in the worst case, propagation in this manner would require $2 \log_n N$ steps.

The disadvantage of a scheme based on hierarchies, however, is that it does not ensure immediate (or prompt) message delivery if failures occur. Since there is only one path by which the message can propagate from a source node to a destination node, any node or link failures along this path can cause the message propagation to stop until the failure is restored. This is by far the biggest drawback of a hierarchical propagation scheme. The message propagation on the Usenet takes place in a somewhat hierarchical manner using *flooding techniques*. Propagation is based on the notion of "upstream" and "downstream" sites. A site can attach itself (with permission, of course) to one or more sites and start receiving news feeds from them. The receiving site becomes a "downstream" site for the sending "upstream" sites. The sending site might itself be a "downstream" site for another "upstream" site which feeds it. This approach creates a hierarchy indirectly, and therefore, it suffers from the same problem just mentioned. For instance, if any one node in the path from an upstream site to a downstream site is down, the downstream site will not receive feeds. In the Usenet, this problem is overcome by having more than one "upstream" sites and receiving simultaneous feeds from them, and then eliminating duplicates. However, this causes a large amount of redundant traffic since a message is received from multiple upstream sites. Moreover, messages can still get lost because various sites have different message deletion policies. Finally, sometimes follow-up or reply messages can get posted even before the original message to which the follow-up relates. If the original message is lost, it might never be posted.

Our goals in designing a new hierarchical propagation scheme are as follows. First, it should be reliable, i.e. messages must not get lost. Second, it must minimize redundancy. Third, it must be scalable and efficient. This means that the workload should be uniformly distributed and no node should have to maintain a global view of the network. Fourth, it must allow messages to propagate in spite of failed nodes. Our failure model enables such propagation to occur by means of *diffusion* past any failed node provided its parent node and child nodes are up. Fifth, it must ensure that messages are seen in the same order that they are posted and replies or follow-ups are seen after the original message. Finally, it must also allow the network to be reorganized dynamically without any loss of messages. This means, for instance, that a node anywhere in the hierarchy should be able to move to a new position elsewhere in the hierarchy without losing any messages during the time it switched positions and without affecting the operation of other nodes in the network.

Another proposal based on hierarchies was given recently by one of the authors of this paper in [Adl93, ANB93]. However, that proposal requires global state information to be maintained, while the present scheme relies on local state information and does not involve exchange of global information. A more detailed comparison between these two proposals is given in Section 6.

This paper describes in detail our propagation protocol called HPP (Hierarchical Propagation Protocol) for managing replicated data. The organization of this paper is as follows. Section 2 briefly reviews the overall system model, and states our assumptions about the processors and the communications network. Then, Section 3 describes the operation of our propagation algorithm during normal conditions. Section 4 turns to explain how network reorganization and restructuring take place. Next, Section 5 describes operation in failure mode. Section 6 reviews the strengths and limitations of the protocol in detail, and Section 7 concludes the paper.

2 System model

The system consists of N nodes connected by an internetwork. Processors may fail, then restart; however, fail-stop processors only are assumed, and byzantine failures do not occur. The communication network is unreliable: it may lose or duplicate messages and does not guarantee any order of delivery. Link failures can cause the network to be partitioned. These partitions are eventually merged again. In the special case where a node loses its communication with all other nodes in the network, it is treated like a node failure. Messages are delayed due to transmission over the network, but a finite delay is assumed. Therefore, 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.

Nodes are organized in a logical hierarchy, where a node i at level l has a *parent* at level *l-1*, a *grandparent* at level *l-2*, n *children* at level *l+1*, $n \times n$ *grandchildren* at level *l+2* and so on. *Neighbors* are nodes with the same parent node. Each node communicates only with its parent and children, which are together referred to as its *correspondents*. We use the notation $P(i)$ to denote the parent node of node i, and $C_j(i)$ to denote the j^{th} child of node i (see Figure 1).

Messages on the network are classified into one of three categories: *normal, reply* or *control* messages. *Normal* messages are assumed to be messages that are unrelated to any previous messages. These are treated slightly differently from *reply* messages which relate to a previous message. This distinction is necessary to maintain the correct ordering between a normal message and a reply message that might relate to it. Finally, control messages are special messages which are propagated through the network like other normal and reply messages, but these messages only perform a control function, such as the join_request and drop messages described in Section 4.

3 Propagation during normal operation

The basic scheme for propagation is very simple: a node generating a message sends it to all its correspondents (parent and children). A node receiving a message from a correspondent,

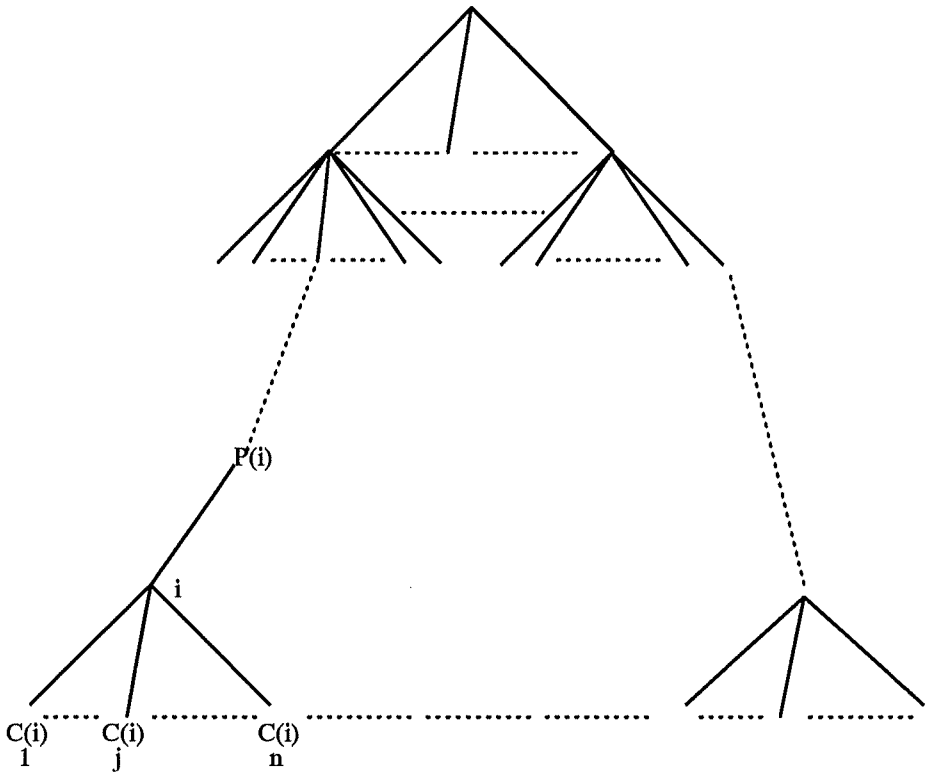

Figure 1: A multi-level hierarchy of nodes

sends the message to all correspondents *except* the one the message is coming from. This works recursively and a message originating at any site will eventually propagate everywhere. A node receives each message only once, so there is no redundancy during normal operation.

Every message m must have the identity of the originator *m_orig* (i.e. the node that creates the message) and a corresponding sequence number *m_seq*. The combination $\langle$*m_orig, m_seq* $\rangle$ creates a unique message identifier. Also, a node tags every message it sees with two values W and W', where W is a sequence number of messages sent downwards, and W' is a sequence number of messages sent upwards by the node. Messages are also tagged with an indication of the previous sender *m_prev* (*m_prev* $= -1$, if the message came from the parent; *m_prev* $= k$, if it came from the k^{th} child; and *m_prev* $= 0$, if the node itself originated the message). A bit map is kept with each message (number of bits $= n + 1$) to keep track of which correspondent has acknowledged the message.

Each node i keeps the following state vectors:

- V_i, a vector describing what messages node i has generated or received from its own correspondents.

  $V_i = (U, D_1, D_2, ..., D_n, L, W, W')$, where

  U = the number of the last message received by i from *up* i.e. from the parent $P(i)$
  D_j = the number of the last message received by i from *down* i.e. from child $C_j(i)$
  L = the number of the last *local* message generated by node i itself
  W = sequence number of the downward stream i.e. a sequence number assigned by i to messages sent to its child nodes.
  W' = sequence number of the upward stream i.e. a sequence number assigned by i to messages sent to its parent.

(In the rest of the paper, we use the notation $V_i.x$ to denote entry x in the vector V_i.)

- V_i^P, a vector describing node i's view of its parent's V vector: $V_i^P = (U, D_1, D_2, ..., D_n, L, W)$. The definition of each entry is the same as V_i but w.r.t the parent.

- V_i^C, a vector keeping track of received messages that were originated by child nodes of node i.

  $V_i^C = (L_1, L_2, L_3, ..., L_n)$ where

  $V_i^C.L_j$= the number of the last message generated locally by the child $C_j(i)$ and received by i.

- Each node maintains a separate *Message Vector MV* which keeps track of the last message number received from every other node in the network. This vector is local to each node. An entry $MV_i[j]$ in this vector means that node i has received all messages generated by node j up to message number $MV_i[j]$.

Therefore, state vector V_i contains information about messages received by node i itself, and V_i^P and V_i^C contain information that node i maintains about the states of its parent and child nodes respectively. By aggregating messages this way, in terms of *local* (L's), received from *down* (D's), and received from *up* (U's), these three vectors *encapsulate* a large part of the information that i needs to maintain. The MV vector is used to detect duplicate messages.

The state vectors V, V^P, V^C and MV are updated upon the generation or receipt of any message. The details of updating the state vectors are shown in Figure 2. The figure lists the steps that a node must perform depending upon whether it originates a message, receives a message from the parent node, or receives a message from a child node. In the figure, we assume that node i is the r^{th} child of its parent $P(i)$.

When node i sends messages to correspondent j, j receives these messages in the same order they were sent from i, i.e., in FIFO order. Messages received out of order are inserted

When node i generates a message m, it takes the following steps:

```
{
increment V_i.L, V_i.W, V_i.W' and MV_i[i];
set m_prev = 0 ;
send m to child nodes;
send m to parent and increment V_i^P.D_r;
On receiving acknowledgment from parent, increment V_i^P.W and V_i.U;
}
```

When a node i receives a message m from its parent, it performs the following steps:

```
{
If m_seq ≤ MV_i[m_orig] Then
   discard m (it is a duplicate) ;
   Else {
      increment V_i.U, V_i^P.W, V_i.W and MV_i[m_orig] ;
      If m_prev = −1 Then increment V_i^P.U ;
      If m_prev = k Then increment V_i^P.D_k ;
      If m_prev = 0 Then increment V_i^P.L ;
      set m_prev = −1;
      send m to child nodes ;
      }
}
```

When node i receives a message m from its j^{th} child $C_j(i)$

```
{
If m_seq ≤ MV_i[m_orig] Then
   discard m (it is a duplicate) ;
   Else {
      increment MV_i[m_orig], V_i.W, V_i.W' and V_i.D_j ;
      set m_prev = j ;
      If m_orig = C_j(i) Then increment V_i^C.L_j ;
      send m to its C_k(i), ∀k ≠ j ;
      send m to parent and increment V_i^P.D_r ;
      send acknowledgment to C_j(i) ;
      On receiving acknowledgment from parent, increment V_i^P.W and V_i.U;
      }
}
```

Figure 2: Algorithm for updating the state vectors upon originating or receiving a message

in a queue for later processing. We call these queues FIFO queues. In general, there are $n+1$ such FIFO queues kept by each node, one per correspondent, and this ensures that messages are received in the same order that they were sent between a pair of correspondent nodes.[1] Since all nodes observe FIFO order while propagating messages, and there is only one unique path for a message between any pair of nodes, causal ordering is maintained during normal operation, i.e., if a node i receives a message m_1 at time t_1 and generates a message m_2 at time t_2 s.t. $t_2 > t_1$, then every other node in the network receives the two messages in the order: m_1, m_2. Theorem 1 given below formally proves that causal order is achieved during normal operation.

Definition 1 *A* **path** *from node i to node j is the set of nodes that a message traverses while propagating from node i to node j by following the hierarchical propagation protocol.*

Corollary 1 *There is a unique path between any pair of nodes i and j.*

Proof: The proof follows directly from the propagation algorithm. Node i sends a message to its correspondents, which in turn send the message (recursively) to their correspondents until it reaches j. Since the nodes are organized in a tree hierarchy, messages from node i to node j must take the same, unique path. □

Theorem 1 *If every node observes FIFO order while propagating messages, then causal order is achieved provided the tree hierarchy is not reorganized and no node failures occur.*

Proof: Assume node o originates a message m_1; node i receives m_1 at $t = t_1$ and then posts m_2 after it sees m_1. We shall show that all nodes in the network will see m_2 *after* m_1. There are two cases:

Case 1: $i = o$

From Corollary 1, all other nodes (correspondents of i, their correspondents, etc.) will receive both m_1 and m_2 from node i by the same path. Since every node along a path processes messages in FIFO order, it follows that all nodes will see m_1 before m_2.

Case 2: $i \neq o$

Consider S, the set of nodes along the unique path (see Corollary 1) from node i to node o. For any node $x \in S$, at $t = t_1$ x has already received m_1, and therefore, it will see m_2 after m_1. Any node $x \notin S$ will receive both m_1 and m_2 from a node y, where $y \in S$. From Corollary 1, the path between x and y is unique. Since every node along a path processes messages in FIFO order, x will see m_2 after m_1.

[1] To support FIFO order , each node should maintain two vectors VS and VR where $VS[j]$ = last sequence number sent to correspondent j, and $VR[j]$ = last sequence number received from correspondent j. When node i sends a message to correspondent j, it increments $VS[j]$ by 1 and tags the message with this value. When node i receives a message from correspondent j it checks the order by comparing $VR[j]$ with the tag on the message.

Therefore, every node in the network sees m_2 after m_1. □

Each node must know the identities of its own correspondents (i.e., its parent node and all child nodes) and their correspondents. Therefore, each node maintains a $[n+2][n+1]$ matrix $View$, where, row 1 holds the id's of its own correspondents, row 2 gives the id's of its parent's correspondents, and row 3 thru n+2 gives the id's of the correspondents of its children (i.e., child nodes 1 thru n, respectively). In each row, the first entry denotes the id of the parent and entries 2 thru n+1 are the id's of the child nodes 1 thru n, respectively.

Messages are kept in a *log*. A message is inserted into the log when received (in FIFO order) and removed from the log when

1) all correspondents have acknowledged the receipt of the message, and

2) a timeout T has elapsed after the last acknowledgment was received

The duration of T has to be specific to each site at the discretion of the site manager. We anticipate that a value of anywhere between 1 and 7 days would be reasonable. A summary of the data structure maintained at each node i, their descriptions and their sizes is presented in Table 1.

Variable name	Description	Size
$View_i[n+2][n+1]$	Partial view of the tree	$(n+2)(n+1)$
V_i	Vector describing i's state	$n+3$
V_i^P	Vector describing i's record of its parent's state	$n+3$
V_i^C	i's record of messages originated by its child nodes	n
$MV_i[N]$	Message Vector	N
log_i	Log keeping received messages	variable
n+1 FIFO queues	Queues maintained for FIFO ordering messages	variable
$VS_i[n+1]$	Vector keeping sequences of sent messages	$n+1$
$VR_i[n+1]$	Vector keeping sequences of received messages	$n+1$

Table 1: Summary of various data structures, their descriptions and sizes

4 Reorganization

This section describes an algorithm that allows a node v to change its parent node without affecting normal operation. The moving node v must either be a leaf node or it should move all its descendants along with it to the new position. Reorganization of the hierarchy helps to maintain a higher level of performance and overcomes various problems such as link congestion.

The reorganization scheme is designed so as to guarantee that no messages will be lost while the switch is taking place. To achieve this, the moving node acts as if it is in two positions simultaneously; for a short duration, it receives messages from both its old and new parent nodes, and discards duplicates. Consider a node v with an *old parent* OP that wishes to join a *new parent* NP. v sends a **join_request** message to NP. NP, on receiving **join_request**, will send a **drop** message to OP so that OP can cease being the parent of v, add v to its $View$, inform all its correspondents about the new child, and send an acknowledgment **ack** to v. The **join_request** and **drop** messages are *special control messages* which will travel through the hierarchy exactly like a normal data message. When OP receives the **drop** message, it can stop sending further messages to v, discard v from its $View$ and inform its own correspondents of the change. Therefore, for a certain time period (between NP receiving the **join_request** message and OP receiving the **drop** message), both NP and OP will have v in their $View$ and consequently, v will receive messages from both of them. This will ensure that v will not miss any messages due to the move. The move is considered complete when node v receives both the **ack** and **drop** messages.

It should be noted that the scheme does not need synchronization between all nodes to commit a change and it does not disturb normal operation. This feature of the reorganization scheme enhances scalability. The details of the protocol are given in Figure 3. This description assumes that v is the r^{th} child of OP and will become the $n+1^{th}$ child of NP.

It can be shown that no messages will be lost as a result of the reorganization by arguing that OP and NP will receive all messages v receives from below, and that v will receive all messages that NP has received before the move or will receive after the move. Assume v initiates the change at time t_1. Then, NP will receive from v every message received by v from below at time $t \geq t_1$. All messages received by v at $t < t_1$ from below would be propagated by it to OP, and then OP will further propagate them to NP by the normal propagation mechanism. Therefore, NP receives *all* messages received by v from below. Similarly, OP receives from v all messages it has received from below at $t < t_1$. All messages received by v from below at $t \geq t_1$ will be sent to NP and NP will further propagate them to OP by the normal propagation mechanism. Therefore, OP receives *all* messages received by v from below.

Assume NP received v's join request at time t_2. Messages received by NP at t $\geq$ t_2 will be sent to v. Meanwhile, OP continues sending messages to v until it receives the **drop**. NP sends the **drop** at $t = t_2$ (NP sends the **drop** immediately after receiving **join_request**; this happens atomically). Since the **drop** propagates through the network like other normal messages and since the FIFO order is preserved in transmitting messages between correspondents, any message that is received by NP at $t < t_2$ will be received by OP before receiving the **drop**. Then, v is guaranteed to receive all messages that NP received at $t < t_2$ from OP and no messages are missed. A complete proof is given in Section 5.

To preserve the causal order while the switch is taking place, node v does not process messages received directly from NP until the **drop** has been received. This ensures that

Node v

send join_request($View_m[1]$) to NP;
wait for acknowledgment from NP;
on receiving ack(V_{NP}, $View_{NP}[1]$) from NP,
set old_View = $View_m[2]$; old_V^P = V_m^P;
set $View_m[2]$ = $View_{NP}[1]$; V_m^P = V_{NP}; $V_m.U$ = $V_{NP}.W$;
on receiving drop from OP,
discard old_View and old_V^P;

NP on receiving join_request($View_m[1]$)

send the special control message drop(v) ;
add v to its list of child nodes: set $View_{NP}[1, n+2] = m$;
set $View_{NP}[n+3]$ = $View_m[1]$; $V_{NP}.D_{n+1} = 0$; $V_{NP}^C.L_{n+1} = 0$;
send add_child(v) to its correspondents;
send v an acknowledgment ack(V_{NP}, $View_{NP}[1]$);

Correspondent k of NP on receiving add_child(v)

update $View_k$ to include v as a child of NP;
If k is a child of NP then add an entry $V_k^P.D_{n+1}$;

OP upon receiving drop(v)

remove v from $View_{OP}[1]$, V_{OP} and V_{OP}^C ;
remove row r+2 from $View_{OP}$ and rearrange the rows ;
send del_child(v) to its correspondents;

Correspondent k of OP on receiving del_child(v)

remove v from $View_k$;
If k is a child of OP then remove the entry $V_k^P.D_r$;

Figure 3: Reorganization algorithm

v processes all messages sent to v before the move (and received from OP) first, then it processes the new messages sent after the move. Further, since join_request propagates through the network like normal messages before changing correspondents, then NP and OP receive join_request *after* receiving all messages propagated by v before the move. Since NP starts receiving new messages directly from v after v receives the ack, then the ordering is preserved. The following theorem shows that the causal order is not violated while the change is taking place.

Theorem 2 *If no node failures occur, then the causal order is preserved while reorganization takes place.*

Proof: The causal order may be violated due to the fact that while the change is taking place, messages sent from node v may follow different paths to reach NP or OP and vice versa. The proof is based on showing that even if messages follow different paths:
1) nodes NP and OP will still receive messages sent by v in their correct order, and
2) node v will still receive messages sent by NP or OP in their correct order.

Case 1: For messages received by NP and OP from v:
Assume node v sends a message m_1 to OP for propagation at t_0, and then sends the control message join_request at t_1. Subsequently, it sends a message m_2 to NP for propagation at t_2 ($t_0 < t_1 < t_2$). NP receives m_1 and m_2 through different paths: m_1 is received through OP while m_2 is received directly from v. Similarly, OP receives m_1 directly from v while m_2 is received through NP. We shall show that both NP and OP receive m_1 before m_2. Since the join_request propagates through the network like a normal message, from Theorem 1, both NP and OP must receive m_1 before the join_request. Since v will not send m_2 except after receiving ack from NP, this means that v sends m_2 to NP only after NP and OP have received m_1.

Case 2: For messages received by v from NP or OP:
Assume NP sends a message m_1 for propagation at t_0, then it receives a join_request and sends a drop at t_1. Subsequently, it sends a message m_2 for propagation at t_2 ($t_0 < t_1 < t_2$). We shall show that although v receives the messages via different paths (i.e., m_1 from OP and m_2 from NP), it will still process m_2 after m_1. Since v receives both m_1 and drop from OP, from Theorem 1, v must receive m_1 before the drop. Since v does not process messages received directly from NP (i.e. m_2) until v has received the drop, then v will first process m_1, then m_2. The same argument holds for messages sent to v by OP while the change is taking place. □

5 Failures

In this section we describe how the protocol circumvents failures, i.e. messages can be propagated past a failed node in both directions. This is important because in a logical

hierarchy even one failed node can prevent messages from being propagated across it and result in a partition. Our protocol can bypass one failure per correspondent group, i.e., if node i fails, and none of its correspondents are down, it is possible to propagate messages past i in spite of the failure. We call this process by which messages propagate past a failed node as *diffusion*. The basic idea behind this algorithm is that when a node fails, all its child nodes would elect a coordinator CO, and CO would take over all the functions of the failed node until the failed node recovers. When the failed node recovers, it would take back its normal functions from the coordinator CO. The information in the state vectors kept by the various nodes is adequate for a *transition* to be made which enables a coordinator to take over the functions of a failed parent. Failures are detected by a standard failure detection mechanism such as timeouts. The various steps required to be performed after a failure have been grouped into four phases. Assume the failed node is f. Once all its child nodes agree that f has failed, they elect a coordinator *(election phase)*. The coordinator then performs actions to take over the role of f and bring all correspondents of f up-to-date as of the time f failed *(transition phase)*. In the next phase, CO assumes the role of the failed node *(diffusion phase)*, and finally when f recovers, it takes back its function from CO *(recovery phase)*. These four phases are summarized below:

- *Phase 1: Election*
  Child nodes of f elect one of them to act as a co-ordinator CO.
- *Phase 2: Transition*
  CO contacts all correspondents of f and a *transition algorithm* is run which is responsible for propagating any message f failed to successfully send to all its correspondents before failing.
- *Phase 3: Diffusion*
  CO takes over the responsibilities of node f and acts as a temporary parent for each child of f, $C_j(f)$, and as a temporary child for $P(f)$ to ensure the continuity of propagation flow while f is down.
- *Phase 4: Recovery*
  When f comes up, f runs a *recovery algorithm* to bring itself up-to-date and resumes its functions.

In the first phase, it is assumed that a standard election algorithm is run (see [GM82]). In the following, details of the next three phases are described.

5.1 Transition

Assume f was in the process of sending a message to its correspondents when it failed. The *transition algorithm* ensures that even if these messages were sent to a subset of the

(1)Ask node $P(f)$ for $V_{P(f)}.W$, $V^C_{P(f)}.L_j$ and $V_{P(f)}.D_j$;
(2) Ask nodes $C_i(f)$ for $V^P_{C_i(f)}$, $i = 1 \ldots n$;
(3) Let $Max\_L_f = Max\{V^C_{P(f)}.L_j, Max_i\{V^P_{C_i(f)}.L\}\}$;
If $(Max\_L_f = V^C_{P(f)}.L_j)$ **Then** $i^* = 0$;
Else $i^* = i$ s.t. $Max\_L_f = V^P_{C_i(f)}.L$;
(4) Construct the vectors $V^{P_{MAX}}$ and $V^{P_{MIN}}$ s.t.,
$V^{P_{MAX}}[i] = V^P_{C_i(f)}.D_i$, $\forall i$;
$V^{P_{MIN}}[i] = Min_j\{V^P_{C_j(f)}.D_i\}$, $\forall i$;
(5)Ask $P(f)$ to send $C_i(f)$, $\forall i$, the last $(V_{P(f)}.W - V^P_{C_i(f)}.U)$ messages sent downwards;
(6) **If** $(i^* = 0)$ **Then**
Ask $P(f)$ to send $C_i(f)$, $\forall i$, the last $(Max\_L_f - V^P_{C_i(f)}.L)$ messages received with $m\_orig = f$;
Else {
Ask node i^* to send $P(f)$ the last $(Max\_L_f - V^C_{P(f)}.L_j)$ messages received with $m\_orig = f$;
$\forall k \neq i^*$,
Ask node i^* to send $C_k(f)$ the last $(Max\_L_f - V^P_{C_k(f)}.L)$ messages received with $m\_orig = f$;
}
(7) Ask nodes $C_i(f)$, $\forall i$, to send to their *neighbors* their last $(V^{P_{MAX}}[i] - V^{P_{MIN}}[i])$ messages sent upwards;
(8) Let $y = \Sigma_k V^P_{C_k(f)}.D_k - (V_{P(f)}.D_j - Max\_L_f)$;
If $y > 0$ **Then**
Ask $C_i(f)$, $\forall i$, to send the last y messages sent upwards to $P(f)$;

Figure 4: Transition algorithm

correspondents (in the worst case to only one of them), still they will be delivered reliably everywhere. The algorithm is initiated by CO after being elected.

Without loss of generality, assume that f is the j^{th} child of its parent $P(f)$. The algorithm is run by CO and the steps in the algorithm are listed in Figure 4. In the first four steps, CO gathers the state information it needs. In the next four steps, it instructs some nodes to send to other nodes messages that the latter have missed. The objectives of this exercise are: (1) to find out which node has the most current information as of the time node f failed, and (2) to arrange for that node to send messages to other nodes which are behind. At the end of this phase, all nodes are current as of the time node f failed, and then CO is ready to assume the functions of its parent node f.

In steps 1 and 2, CO gathers the relevant state information from the parent and child

nodes of f. In step 3, it determines which correspondent of f (among f's parent and child nodes) has received the largest number of messages that were generated locally at f and keeps this number in Max_L_f. Since each child node maintains a view of its parent's V vector, and these views could be different, step 4 compares these vectors maintained by the child nodes of f. The objective is to determine the highest numbered message received by f from each of its child nodes until failure, and also to determine for all messages received by f from each child node, how far behind the other child nodes are (since f must propagate messages received from one child node to all other child nodes). Consequently, $V^{P_{MAX}}[i]$ is the highest numbered message f has received from its child node i; $V^{P_{MIN}}[i]$ is the number of messages out of these that have reached the child node that is most behind.

The subsequent steps of the algorithm can be explained better by examining all possible ways in which f might send a message to, or receive a message from, one of its correspondents and then fail before propagating the message to *all* its other correspondents. In all such cases, our transition algorithm must ensure that such a message does reach *all* correspondents of f. We divide this problem into four cases depending upon whether the correspondent of f is a parent node or a child node, and also depending upon whether f has sent a message to, or received the message from, the correspondent. These four cases are:

- f receives a message from $P(f)$, and then f dies.
- f receives a message from a child node $C_j(f)$, and then f dies.
- f sends a message to $P(f)$, and then f dies.
- f sends a message to a child node $C_j(f)$, and then f dies.

Each of these four cases is discussed separately below, and in each case we explain how the appropriate step from our algorithm ensures that a message that falls in that case is propagated to all other correspondents of f (the step numbers below refer to Figure 4).[2]

- **Case 1:** $P(f)$ generates a message itself (or receives a message from a child or a parent node), sends it to f, and then f dies (denoted $P(f) \rightarrow f$, f dies)
  Since $P(f)$ has the message, then it will get propagated upwards in spite of the failure of f. However, to ensure that it will also be propagated downwards, CO must compare $V_{P(f)}.W$ with $V^P_{C_i(f)}.U$: if $V_{P(f)}.W > V^P_{C_i(f)}.U$, then, $V_{P(f)}.W - V^P_{C_i(f)}.U$ is the number of messages missed by $C_i(f)$, and CO asks $P(f)$ to send those messages to $C_i(f)$ (*see Step 5*).

[2]The parameter T which influences the deletion policy of a node (as described in Section 3) is large enough to ensure that a correspondent of f has not already deleted a message that it received from f, and may need to diffuse.

- **Case 2:** f has generated a message, sent it to $P(f)$, and then died (denoted by $f \rightarrow P(f)$, f dies)
  Since $P(f)$ has received the message, it will again get propagated upwards in spite of the failure of f. To ensure that it is propagated downwards, CO compares $V^C_{P(f)}.L_j$ with $V^P_{C_i(f)}.L$: if $V^C_{P(f)}.L_j > V^P_{C_i(f)}.L$, then $C_i(f)$ has missed some messages and CO asks $P(f)$ to send $C_i(f)$ the missing messages (*see Step 6*).

- **Case 3:** f has generated a message, sent it to one of its child nodes (say, $C_i(f)$), and then died (denoted $f \rightarrow C_i(f)$, f dies)
  In order to ensure that such messages are propagated upwards, CO compares $V^P_{C_i(f)}.L$ with $V^C_{P(f)}.L_j$: if $V^P_{C_i(f)}.L > V^C_{P(f)}.L_j$, then $P(f)$ has missed one or more messages and CO must ask $C_i(f)$ to send the missing messages to $P(f)$. In order to ensure that the messages are propagated downwards, CO compares $V^P_{C_i(f)}.L$ with $V^P_{C_k(f)}.L$: if $V^P_{C_i(f)}.L > V^P_{C_k(f)}.L$, then $C_k(f)$ has missed some messages and CO asks $C_i(f)$ to send the missing messages to $C_k(f)$ (*see Step 6*).

- **Case 4:** $C_i(f)$ generated a message locally (or received a message from below), sent it to f, and then f died (denoted $C_i(f) \rightarrow f$, f dies).

  To ensure that such messages are propagated downwards, CO compares $V^P_{C_i(f)}.D_i$ with $V^P_{C_j(f)}.D_i$: if $V^P_{C_i(f)}.D_i > V^P_{C_j(f)}.D_i$, then $C_j(f)$ has missed some messages sent by $C_i(f)$ and CO asks $C_i(f)$ to send those messages to $C_j(f)$ (*see Step 7*). To ensure that such messages are propagated upwards, CO checks if: $(V_{P(f)}.D_j - Max\_L_f) < \Sigma_k V^P_{C_k(f)}.D_k$. If so, then $P(f)$ is missing some messages sent by child nodes and CO asks them to send those messages to $P(f)$ (*see Step 8*).

The different cases along with the conditions used to detect the need for upwards and downwards propagation are summarized in Table 2. A "–" in Table 2 means that the message has already been propagated in that direction and no action is required.

Case	Upwards	downwards
$P(f) \rightarrow f$, f dies	–	$V_{P(f)}.W > V^P_{C_i(f)}.U$
$f \rightarrow C_i(f)$, f dies	$V^P_{C_i(f)}.L > V^C_{P(f)}.L_j$	$V^P_{C_i(f)}.L > V^P_{C_k(f)}.L$
$f \rightarrow P(f)$, f dies	–	$V^C_{P(f)}.L_j > V^P_{C_i(f)}.L$
$C_i(f) \rightarrow f$, f dies	$V_{P(f)}.D_j - Max\_L_f < \Sigma_k V^P_{C_k(f)}.D_k$	$V^P_{C_i(f)}.D_i > V^P_{C_k(f)}.D_i$

Table 2: A summary of failure scenarios and conditions to detect upwards and downwards propagation.

Since missing messages can fall in only one of the four cases described above, and since all of them are detected and propagated, it follows that: *if a node fails and it has sent a*

message to at least one of its correspondents before failing, then this message will be reliably propagated to all its other correspondents and consequently, will be reliably propagated to every other node in the network.

CO assembles all requests to $P(f)$ or $C_i(f)$ into one message including the total number of messages that each of them is supposed to receive from others. $P(f)$ or $C_i(f)$ receiving diffused messages treat them exactly as if they were coming from f. The only change is that the child nodes $C_i(f)$ $(i = 1 \ldots n)$ need not update $V^P_{C_i(f)}$, since it represents f's state, and that is frozen because f is down. Further, the batch of diffused messages should be sorted such that *normal* messages are processed first, followed by *reply* messages, and, lastly, *control* messages. This sorting step is necessary because now it is possible that a message and its follow-up might be received by a node through *different* paths such that the follow-up reaches before the original message, thus disturbing the causal order. In such a case, sorting will ensure that the ordering requirements are still satisfied.

5.2 Diffusion

While node f is down, CO takes over the responsibilities of f temporarily until f comes up again. That is, $C_i(f)$'s and $P(f)$ send messages to CO, and CO will diffuse them to the other correspondents of f. Therefore, it is ensured that the flow of propagation will continue while f is down. Messages are transmitted in FIFO order between CO and correspondents of f. This phase starts once the transition phase is terminated, i.e., when $P(f)$ or $C_i(f)$'s have received and processed all messages they were supposed to receive during the transition phase. Then, each node can move into the *diffusion phase* independently.

CO, on receiving a message m from a correspondent of f, treats it as if it is coming from a parent node, and performs the algorithm described in Section 3 (see Figure 2), i.e. CO must update MV_{CO}, increment $V_{CO}.W$ and $V_{CO}.U$, send m to its own correspondents, etc. Additionally, it must also send the message to correspondents of f, other than the one the message is coming from.

If CO generates a message locally, or receives a message from one of its own correspondents, it treats it normally (see Figure 2), but, in addition, sends it to all correspondents of f also.

A correspondent of f, on receiving a message from CO, acts as follows: if the correspondent is $P(f)$, then it treats the message as if it is coming from a child (see Figure 2); if the correspondent is a child node of f, then it treats the message as if it is coming from a parent (again see Figure 2), but does not update $V^P_{C_i(f)}$.

The following theorem shows that no messages are lost if a reorganization occurs even if a failure occurs while the reorganization is taking place.

Theorem 3 *If a node v changes its parent from OP to NP, then it is guaranteed that no*

messages are lost as a result of the move, even if a failure occurs along the path from OP to NP while the reorganization takes place.

Proof: The proof is based on arguing that OP and NP will receive all messages v receives from below, and that v will receive all messages that NP has received before the move or will receive after the move.

Node NP: Assume v initiates the change at time t_1.

Every message received by v from below at time $t \geq t_1$ will be received by NP directly from v. All messages received by v at $t < t_1$ from below, OP receives them directly from v and OP will further propagate them to NP by the normal propagation mechanism. If a failure occurs along the path from OP to NP while messages are propagating, the transition and diffusion phases guarantee that no messages will be missed by NP. Therefore, NP receives *all* messages received by v from below.

Node OP: Similarly,

Every message received by v from below at $t < t_1$ will be received by OP directly from v. All messages received by v from below at $t \geq t_1$ will be sent to NP and NP will further propagate them to OP by the normal propagation mechanism. If a failure occurs along the path from NP to OP while messages are propagating, the transition and diffusion phases guarantee that no messages will be missed by OP. Therefore, OP receives *all* messages received by v from below.

Node v: Assume NP receives v's join request at time t_2. r.t.p:

(1) v receives every message received by NP at $t \geq t_2$

(2) v receives every message received by NP at $t < t_2$

The proof of (1) is trivial since NP will consider v as a correspondent at $t = t_2$ and will start sending it any message generated or received. To prove (2), we need to prove that any message that NP has received at $t < t_2$, v will receive it from OP. Assume that NP has received a message m at $t < t_2$. Since NP generates the **drop** at $t = t_2$ and since OP continues sending messages to v until it receives the **drop**, then we need to show that OP will receive m before receiving the **drop**. Three cases can occur while m and **drop** are propagating:

Case 1: while m and **drop** are propagating, there were no failures nor reorganization all along the path to OP. Then, from Theorem 1, OP will receive m before receiving the **drop**.

Case 2: while m and **drop** are propagating, a failure occurs in their path of propagation towards OP.

If the failure occurs after m reaches OP, then whether **drop** is sent during the transition phase or the diffusion phase, it will be received by OP after m anyway.

If the failure happens before m reaches OP, there are four cases:

Case 2.1: m and drop are diffused by the *transition algorithm*

If OP was one of the failed node's correspondents, then sorting m and drop -such that drop comes last- guarantees that OP will process m first, send it to v then process the drop.

If OP was not one of the failed node's correspondents, then OP will receive m and drop through normal propagation. Since each correspondent of the failed node sort messages, such that drop comes last, before processing and further propagating them and since they maintain the FIFO order in propagation, then OP will receive m first then the drop.

Case 2.2: m and drop are sent during the *diffusion phase.*

Since the coordinator CO becomes a correspondent of the failed node's correspondents and each node maintains FIFO order then Theorem 1 applies and OP will receive m before the drop.

Case 2.3: m is diffused by the *transition algorithm* while drop is sent during the *diffusion phase.*

Since the diffusion phase starts only after the transition phase is over, then m will precede drop in propagation.

Case 2.4: drop is diffused by the *transition algorithm* while m is sent during the *diffusion phase*

This case cannot occur. Assume that the failure occurs at time t_f. For this case to occur, then at time $t < t_f$, one or more correspondents of the failed node should have received the drop but not m. Since Theorem 1 applies for $t \leq t_f$, this is a contradiction and this situation cannot occur.

Case 3: while m and drop are propagating, a reorganization takes place.

From Theorem 2, since the causal order is preserved while a reorganization occurs, then OP receives m before receiving the drop. □

5.3 Recovery

When node f comes up, two functions should be performed:

1. f must check if there is any message in its log which has not been acknowledged by a correspondent i, and if so, send this message to i. Node i might have already received this message during the transition algorithm. Therefore, on receiving such a message, node i checks whether it is a duplicate, and, if so, discards it; otherwise, it accepts the message and updates V_i. Node i must also check whether it is holding any pending messages in the FIFO queue for f and flush the queue. This function ensures that the diffusion process is completed e.g. if a node had received a message from f but had deleted it from its log before the transition algorithm starts and hence the message was not diffused, or, if f has generated a message but did not send it to any other node before failing. Further, this function is essential for correspondents of f, who

might or might not have received these messages, to flush any pending messages in the FIFO queue each of them keeps for f.

2. f should bring itself up-to-date and resume its functions. While f is down, each correspondent of f, on generating a new message, or receiving a message from its own correspondents, inserts it in the log, marks it as not acknowledged by f and sends it to CO. When f comes up, each correspondent extracts from its log all messages not acknowledged by f and sends them to f. This ensures that f will receive *all* messages it missed during the failure. Node f, on receiving such messages from a correspondent i, accepts non-duplicate messages but does not forward them to its other correspondents as they already have them. It increments its $V_f.U$, if i is a parent, or $V_f.D_j$, if i is the j^{th} child, by the number of non-duplicate messages received from correspondent i. At the end of this process, f increments $V_f.W$ by the total number of non-duplicate message received from all correspondents.

   Recall that, while f was down, its child nodes were not required to update their $V^P_{C_i(f)}$ vector. Therefore, on recovering, f sends its V_f vector to all its child nodes $C_i(f)$ so that they can use the incoming vector as their new $V^P_{C_i(f)}$. Further, f needs to update its record of its parent's state; so, f requests $P(f)$ for its $V_{P(f)}$, and calls it V^P_f.

   Afterwards, normal operation of f is resumed. (If, CO receives a message from a correspondent of f after f comes up, then CO returns the message back to the sender noting that f is alive and the sender must resend the message directly to f.)

5.4 Partitions

In large scale systems, link failures can occur frequently. A sequence of link failures leads to partitions. It is assumed that an external process is responsible for maintaining link status, i.e., detecting when partitions occur and are restored.

When partitions are detected, correspondents in one partition mark in their view their correspondents in the other partition as being isolated in order not to attempt sending them messages. However, messages generated or received by correspondents in one partition are kept in the log for the isolated correspondents. When the partition heals, each pair of previously isolated correspondents exchanges messages that were kept for each other, update their state vectors and propagate received messages as usual. This will ensure that any messages that were missed during the partition will be received. Afterwards, normal operation is restored.

6 Discussion

Three major advantages of the HPP protocol over the Usenet are higher reliability, ability to restructure without messages being lost and minimum redundancy. Restructuring has implications for performance. For instance, if a node is not receiving good response time from its parent node, it could move to another location (i.e., find another parent). After a series of such moves, the overall performance of the network would improve. One performance criterion could be the average delay in receiving messages from other nodes. If this measure is continuously above a threshold for a given period of time, then a node would consider relocating. Minimizing redundancy leads to lower communication overhead and efficient use of network bandwidth. In the Usenet there is trade-off between redundancy on the one hand, and lost messages and delay on the other. By having contact with several "upstream" nodes, a node could minimize the number of lost messages and delay, but only at the cost of greater redundancy. In our model, due to the hierarchical pattern of propagation that updates follow, a node receives each message only once, and so there is no redundancy during normal operation, and yet reliable delivery is assured.

As mentioned earlier, basing the propagation scheme on a logical hierarchy allows the system to scale well, since each node communicates with only its correspondent nodes. This leads to lower communications overhead and a more even distribution of the burden of propagation among all nodes. One unique feature of the HPP protocol is that it allows aggregation and encapsulation of the important aspects of the state into a few state vectors containing minimal information. These aggregate state vectors provide enough information for the determination of missing messages. In the absence of such aggregation, any pair of nodes would have to compare very long vectors in order to determine what messages either node is missing.

The limitation of our protocol lies in the fact that messages can be diffused past a failed node only if its parent and all child nodes are alive. This means that "successive" failures (i.e., where a pair consisting of a parent node and a child node are down) will cause the diffusion to stop. Of course, no messages will still be lost, and all messages will be delivered when the failures are restored. Moreover, messages can also be diffused in spite of other kinds of multiple failures that do not involve a parent-child pair. However, to handle successive failures, more information needs to be kept in the state vectors by each node. The state information kept at each node at present is able to handle one level of failure; if additional state information is maintained at each node, then the solution can be extended to handle successive failures. In this case, there is then a trade-off between the probability and associated cost of such successive failures and the cost of maintaining the additional information. However, the same general approach can be extended to cover such scenarios also.

In [Adl93, ANB93], a propagation scheme based on hierarchies is described. HARP adopts a more general hierarchical structure, where nodes are grouped into clusters and

clusters are organized into a tree. A node communicates with the members of its clusters as well as its parent and child nodes. This allows messages to propagate slightly faster, since a node sends messages to its neighbors also. Having multiple nodes at the root of the hierarchy offers better availability and reliability than the scheme described here where there is only one node at the root. But, reorganizing the hierarchy incurs more overhead since the structure is more complex. The most important difference between HARP and HPP, however, is that HARP maintains global state information and does not encapsulate the state information as in HPP. By encapsulating the state information, the HPP protocol requires each node to maintain only local information and this is an important feature in a wide area network. HARP, on the other hand, consumes more storage by keeping global state information. Also, it produces more communications traffic because large vectors which maintain the global state have to be exchanged in case failures or reorganization occur. On the other hand, HARP tolerates any pattern of failures (including successive failures), and provides several orders of delivery (unordered, FIFO, causal and total order), from which an application may choose one depending on its requirements. Therefore, other than both being based on the notion of a logical hierarchy, HARP and HPP have little else in common.

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 towards small systems and ensures strong consistency of group views at the expense of latency and communication overhead. Causal order is maintained by timestamping each message with ordering information, of size N, representing message ids already seen by the sender. When a message m arrives at a destination, if one or more of m's predecessors' messages have not arrived, then m's delivery is delayed until the appropriate messages arrive. However, the overhead of piggybacking with each message a timestamp of size proportional to the total number of nodes in the network can be quite expensive, especially for a large number of nodes.

7 Conclusions

This paper has described a weak-consistency replica control protocol called HPP which efficiently and reliably propagates messages in wide area networks and can be used to implement a bulletin board service on a large network such as the Internet. This is already an important service on the Internet and is becoming increasingly popular on commercial networks such as Prodigy and Compuserve. The current implementation of Usenet News on the internet is not very reliable. The HPP protocol is based on organizing the nodes in the network into a logical hierarchy, and is both scalable and efficient. It ensures reliable eventual delivery of messages in spite of failures or partitions. Further, it minimizes redundancy which makes efficient use of network bandwidth. It also allows nodes to dynamically

change their position in the hierarchy to improve performance while ensuring that messages are not lost. Finally, each node maintains only local state information by encapsulating information about itself and its parent and child nodes into state vectors. This reduces communication traffic and makes the scheme more scalable.

Our protocol is able to tolerate various failures of one or more nodes in the network in that it can *diffuse* messages past them in the hierarchy. However, this diffusion is possible only if the parent and child nodes of the failed node are alive. If a parent-child pair of nodes fails, then we call it a successive failure, and in this case messages would not be diffused past the failed nodes until at least one of them recovers. On the other hand, if failures are such that the failed nodes are at alternate levels and a parent-child pair is not involved, then the algorithm is able to bypass the failed nodes and diffuse messages in spite of multiple such failures. It would be possible to extend the protocol to withstand more failures provided additional state information is kept. Clearly there is a trade-off between the amount of additional information stored and the ability of the protocol to tolerate failures. We expect to study this issue in future research. A more detailed performance comparison of the HPP algorithm with some of the other techniques described in this paper is also planned.

Acknowledgment

Noha Adly has been supported by an FCO grant from the British Council.

References

[Adl93] N. Adly. HARP: a hierarchical asynchronous replication protocol for massively replicated data. Technical Report TR-310, Computer Laboratory, University of Cambridge, UK, August 1993.

[ANB93] N. Adly, M. Nagi, and J. Bacon. A hierarchical asynchronous replication protocol for large scale systems. In *Proceedings of the IEEE Workshop on Parallel and Distributed Systems*, Princeton, NJ, October 1993.

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

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

[CAA90] S. Cheung, M. Ammar, and M. Ahamad. The Grid protocol: a high performance scheme for maintaining replicated data. In *Proceedings of the IEEE 6th Int. Conf. on Data Engineering*, pages 438–445, 1990.

[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.

[DGP90] 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.

[ED92] Alan Emtage and P. Deutsch. archie – An Electronic Directory Service for the Internet. In *Conference Proceedings Usenix*, San Francisco, CA, January 1992.

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

[GM82] Hector Garcia-Molina. Elections in a Distributed Computing System. *IEEE Trans. on Computers*, C-31(1):48–59, January 1982.

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

[Kum91] A. Kumar. Hierarchical quorum consensus: a new algorithm for managing replicated data. *IEEE transactions on Computers*, 40(9):996–1004, September 1991.

[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.

[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*, Quebec City, CA, 1990.

[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.

[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*, August 1984.