Management of Replicated Data in Large Scale Systems

Noha Adly

Corpus Christi College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

August 1995

Abstract

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 scale and autonomy issues adequately. Further, current applications require different degrees of consistency, and therefore they should be given the ability to choose the level of consistency that is appropriate for their particular semantics. This dissertation presents a new scalable replication protocol (HARP) that is based on organising the replicas into a logical hierarchy. It is argued that adopting a hierarchical structure allows for exploiting localised communication, which is taken as the key to achieve scalability. Moreover, it gives the ability to provide different degrees of consistency.

HARP provides an efficient and scalable propagation scheme where each node needs to communicate with a few nodes only while ensuring reliable delivery. A new service interface is proposed that gives the application the flexibility to choose between strong and weak consistency. Further, the scheme provides the ability to offer different levels of staleness, depending upon the needs of various applications. Dynamic restructuring operations are presented which allow the hierarchy to be built and reconfigured, including the restarting of failed nodes and re-merging partitioned networks. The operations produce low message traffic by exploiting localised communication, and do not disturb normal operations. This is achieved while ensuring no loss of messages.

Reconciliation methods based on delivery order mechanisms are provided to resolve temporary inconsistencies and an application may choose from them. A new algorithm that supports causal order delivery is proposed. The desirable characteristic of the algorithm is that, by relying on the hierarchical propagation of HARP, it cuts down the size of the timestamp required to verify causality significantly, and thus enhances scalability.

A detailed simulation study was carried out to evaluate the performance of HARP and to quantify the benefits and losses resulting from moving from strong consistency to weak consistency under different system configurations and load mixes. Further, a simulation study was conducted to compare the performance of HARP to another weak consistency replication protocol, the *Time Stamped Anti Entropy*.

An alternative hierarchical propagation protocol is proposed as an optimisation of HARP, called HPP. The main difference between HPP and HARP is that HPP avoids the exchange of global state information when reconfiguration or failures occur. Therefore, HPP is more scalable; however, it can tolerate only special patterns of failure. The protocol is presented in detail and its strengths and limitations are analysed.

To my parents

Acknowledgements

I would like to thank my supervisor, Dr. Jean Bacon, for her unceasing encouragement, support and practical assistance during the course of this research. My thanks and appreciations also go to Prof. Roger Needham who, at different stages of this project, has provided moral and technical supports.

I am deeply indebted to Prof. Magdy Nagi of Alexandria University for his valuable advice and technical assistance throughout this work. His unlimited help and constant support are greatly appreciated. My gratitude also goes to Prof. Akhil Kumar of Colorado University for giving me the chance to benefit from his experience in the field of replicated data and for his assistance while developing the HPP protocol, presented in Chapter 8. I would like also to thank Benjamin Bacarisse of University College of London for his helpful suggestions at the initial stage of this project and practical advice while designing the performance models.

My profound gratitude goes to all of the members of the System Research Group at the University of Cambridge, from whom I have learned so much, in particular Dr. Ken Moody, Sai Lai Lo, Subir Biswas, Simon Crosby, David Greaves, Matthew Doar, Shaw Chuang, Scarlet Schwiderski, Richard Hayton and Robert Sultana. I would like to acknowledge, with thanks, the useful comments on initial drafts of this dissertation made by Dr. Jean Bacon, Dr. Ken Moody, Prof. Magdy Nagi, Prof. Akhil Kumar and Sai Lai Lo. I am also grateful to Chris Hadley, Graham Titmus, Martin Johnson and Piete Brooks for their excellent team work, yielding a stupendous computing infrastructure in the Computer Laboratory. Lewis Tiffany was ever helpful in my searches for relevant literature.

I will always remember Subir, Simon, Narmhee, Scarlet, Jenniffer and Robert for their help and friendship during my four years in the Computer Laboratory. Special thanks go to my dear friends Amal Esawi, Ruth Frost and Sally Bower for their endless support, care and love. I am also grateful to all my professors and colleagues of Alexandria University for their encouragement and support.

Words cannot express my gratitude to my parents, who have overwhelmed me with love, encouragement and support which have been indispensable. I am deeply grateful to Tony and Christine Adams, who have treated me like one of their own children during my stay in the UK. Their love and care is greatly appreciated.

This work was supported by an FCO Scholarship from the British Council, to which I am extremely grateful. Thanks are also due to the following bodies for generously assisting me to attend several conferences: Corpus Christi College, the British Council, the Computer Laboratory and Cambridge Philosophical Society.

Declaration

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography.

Publications

Some of the work described in this dissertation has been published, and is described in references [Adly 93a, Adly 93b, Adly 95a, Adly 95b, Adly 95c] and [Adly 95d].

Trademarks

Ethernet is a trademark of the Xerox Corporation.
MIPS is a trademark of MIPS Technologies Inc.

Contents

List of Figures

List of Tables

Glossary of Terms

CENA	Centre d'Etudes de la Navigation Aérienne
CPU	Central Processing Unit
CV	Compact Vector
ESR	Epsilon Serializability
ET	Escrow Transaction
FIFO	Fist In First Out
FTP	File Transfer Program
GMT	Greenwich Mean Time
GNS	Global Name Service
GSE	Generalised Site Escrow
HARP	Hierarchical Asynchronous Replication Protocol
HPP	Hierarchical Propagation Protocol
HQC	Hierarchical Quorum Consensus
I/O	Input and Output
JANET	Joint Academic Network
LAN	Local Area Network
NOC	Network Operating Centre
OSCAR	Open System for Consistency and Replication
QC	Quorum Consensus
ROWA	Read One Write All

SE	Site Escrow
STV	Summary Timestamp Vector
2PC	Two-Phase Commit
TQP	Tree Quorum Protocol
TSAE	Time Stamped Anti Entropy
UNS	Universal Name Service
VC	Vector Clock
VV	Version Vector
WAIS	Wide Area Information Service
WAN	Wide Area Network
WWW	World Wide Web

Chapter 1

Introduction

1.1 Replication in Distributed Systems

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. The Internet news systems (Usenet) [Kantor 86] is one example of such systems. It is replicated at thousands of hosts and manages a large database while responding to queries in seconds. In contrast, Archie [Emtage 92], a directory service for Internet FTP archives, which is replicated at only 12 sites, can take 15 minutes to answer a simple query against a much smaller database. As Internet is growing significantly beyond its present two million nodes, and as WWW, Archie and many other information services [Obraczka 93] become more popular, the database of these services must be massively replicated for a reasonable performance.

Traditional approaches for managing replicated data are synchronous; that is, they require that read and write operations be synchronised in order to ensure that replicas are mutually consistent. Protocols requiring synchronisation 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 lock or restrain access to resources during protocol execution and reduce the system availability when one or more nodes fail, or when the network is partitioned.

In contrast, weak consistency protocols allow updates and queries to occur asynchronously at any replica. They operate under the optimistic assumption that concurrent updates will rarely conflict and therefore synchronisation at each step is unnecessary. Updates commit at the local replica, then they are propagated to other replicas and eventually, all replicas observe the updates. During the propagation, the data is in transient inconsistency and the value returned by a read request depends on whether that replica has observed the update or not yet. 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 a guarantee. 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. Therefore, asynchronous approaches provide higher availability and better response time than synchronous approaches. 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 naming systems, information services, air traffic control and stock exchanges.

1.2 Motivation

Although extensive research has been done for managing replicated data, the issues of scalability and autonomy have not been addressed adequately. Typical replication systems, which were designed for local area networks, are not well suited for wide area networks. Internetworks introduce new problems because they are geographically dispersed and they often contain slow and unreliable links. Existing protocols do not scale well because they manage replicas as one flat group of nodes. Requiring a node to communicate with all other nodes in the system might be appropriate for small networks with few replicas, but is unrealistic for wide area networks like the Internet.

With replicated systems, some applications require strong consistency but there are many applications with semantics that can tolerate some inconsistency and weaker forms of consistency are adequate and acceptable. Therefore, it is desired to design a flexible protocol that integrates both strong consistency and weak consistency into the same framework and gives the application the ability to select between them depending on its requirements. When strong consistency is provided, synchronisation should be limited to a small number of nodes to achieve acceptable performance.

Weak consistency protocols provide the users with the guarantee that the data will eventually reach all replicas and become consistent. This guarantee might not be sufficient for the needs of some applications which would like to read a more up-to-date version than the local replica at a certain point of time. Therefore, it is desirable to provide the users with different levels of staleness, by giving access to replicas that are more up-to-date than others.

Reconfiguration schemes are necessary to allow a node to join or leave the set of replicas and the ability to recover from failures. Contemporary schemes ensure a consistent view of all replicas, using two- or three-phase commit protocols or atomic broadcast protocols, at the expense of latency and communication overhead. Again, such protocols can be very inefficient and entirely impractical across internetworks.

1.3 Scope

Based on the above considerations, this dissertation proposes a protocol for managing replicated data that is suitable for large scale systems. It is believed that a weak consistency (asynchronous) approach is more appropriate for applications that need to massively replicate their data provided they can tolerate some inconsistency, and therefore the dissertation focusses on this approach.

The protocol, called Hierarchical Asynchronous Replication Protocol (HARP), is based on organising the replicas into a logical multilevel hierarchy to exploit localised communication. It provides an efficient propagation scheme where each node has to communicate with a few nodes only, typically those in its neighbourhood, thus reducing communication overhead and making the system scalable, while ensuring reliable delivery. Since different applications require different degrees of consistency, a new service interface is proposed that provides different levels of asynchrony allowing strong and weak consistency to be integrated into the same framework. Strong consistency is achieved by assembling a quorum from a small number of nodes, namely, nodes in the top level of the hierarchy. Further, the scheme provides the ability to access replicas that are more up-to-date than others, while using asynchronous queries, by reading from different levels of the hierarchy.

Reconfiguration schemes and methods for handling failures are presented. They are different from contemporary approaches because they rely on a weak consistency semantic, that allows temporary inconsistencies to develop in the view of each node but guarantees that all replicas eventually converge to a single consistent view when the changes cease. They exploit localised communication to reduce network traffic and do not suspend normal operation while ensuring no loss of information.

Reconciliation methods based on delivery order mechanisms are provided to resolve temporary inconsistencies resulting from asynchronous updates. One type of ordering that is of special interest is causal ordering. A new algorithm for maintaining causal ordering is described which, unlike most existing protocols, imposes a little space overhead appended to each message. The algorithm takes advantage of the hierarchical propagation scheme, where each node sends and receives messages from a few nodes only, and cuts down the size of the timestamp required to verify causality significantly. This low cost of timestamp size results in reduced communication overhead and increased performance and scalability.

A simulation study is carried out to evaluate the performance of HARP and to quantify the benefits in performance and losses in consistency resulting from moving from strong consistency to weak consistency under different load mixes and system configurations. Further, a quantitative comparison is performed between HARP and one of the most recent and widely used weak consistency replication protocols, *Time Stamped Anti Entropy*.

Within HARP, global state information is exchanged while reconfiguration takes place, which might impose a high overhead in very large scale systems. Therefore, an alternative hierarchical protocol, called HPP, is proposed as an optimisation. HPP encapsulates the state information such that each node keeps local information which is exchanged in case of failures or reorganisation. Therefore, HPP is more scalable than HARP; however, it can tolerate only special patterns of failure and its support for ordering of updates is weaker; hence, it represents a tradeoff.

1.4 Dissertation Outline

Chapter 2 reviews the development of replication protocols, both synchronous and asynchronous and points out the limitations of these protocols. The chapter also discusses some applications where a weak consistency approach is adequate.

Chapter 3 presents the propagation algorithm used to disseminate updates to all replicas within HARP. Next, the new service interface that integrates synchronous and asynchronous operations is defined. Also, the different levels of staleness and the reconciliation methods provided by the protocol are discussed.

Chapter 4 presents the restructuring operations which allow the hierarchy to be built and reconfigured. Also, methods for handling failures and partitions are described.

Chapter 5 describes a simple and efficient protocol that provides causal order delivery. The protocol manages to reduce significantly the size of the causal order information associated with each message.

Chapter 6 presents the simulation model used to evaluate the proposed protocol. This chapter also outlines the performance results and analysis, obtained experimentally from the implemented simulator.

Chapter 7 compares the performance of HARP with the *Time Stamped Anti Entropy* protocol through simulation. The simulation model is described, then the experiments and the results are discussed.

Chapter 8 presents the Hierarchical Propagation Protocol (HPP) which encapsulates the state information such that each node keeps local information. Methods for handling failures and reconfiguring the hierarchy are described based on these local states. Then, the strengths and limitations of the suggested protocol are analysed and compared to HARP.

Chapter 9 concludes the dissertation with a summary of the presented work and some suggestions for future work.

Chapter 2

Background

This chapter reviews some of the existing replication protocols. Since synchronous approaches have been addressed extensively in the literature, only a few protocols are outlined and published surveys are indicated. Section 2.2 gives an overview of the well known asynchronous replication protocols and points out the limitations of these protocols. It is believed that the asynchronous approach is more appropriate for applications that need to massively replicate their data and require high performance, as long as they can accept a weaker form of consistency. Section 2.3 gives an overview of some of those applications.

2.1 Synchronous Replication Protocols

Most of the schemes developed so far for managing replicated data synchronously are based on one of two basic principles: Primary Copy and Quorum Consensus. These two schemes are first described, then several variations are presented. Surveys of synchronous protocols can be found in [Ceri 91, Son 88].

2.1.1 Primary Copy

In the *Primary Copy* approach [Stonebraker 79], one replica is designated as the primary, all the other replicas are secondaries. Update requests are sent to the primary copy, which acquires locks on all secondaries, performs the update, broadcasts the change to all secondaries and then releases the locks. The primary copy method maintains consistency in the face of network partitions since updates are allowed only at the primary, but it does not tolerate the failure of the primary copy and suffers from bottlenecks at that site.

2.1.2 Quorum Consensus (QC)

Quorum Consensus is a general class of synchronisation protocols for distributed systems. An operation proceeds to completion only if it can obtain permission from nodes that constitute a *quorum group*. Quorum groups used by conflicting operations have non-empty intersection to guarantee proper synchronisation. A well known method for defining quorum sets is Gifford's scheme [Gifford 79] which is a generalisation of the majority consensus method [Thomas 79]: consider n replicas of an object, each replica is assigned some number of votes that can be used in gathering a quorum in order to execute an operation. A read operation needs to assemble a *read quorum*, an arbitrary collection of any r replicas or more. Similarly, to update the object, a *write quorum* of at least w replicas is required. Each replica holds a version number, a write operation reads the version numbers of w replicas, generates a number higher than any it has observed and stores the data at these replicas. The values of r and w are subject to two constraints: (1) $r + w > n$, and (2) $w > n/2$. The first constraint prevents reads and writes from conflicting and ensures that the most recent copy is read while the second constraint prevents two write operations from occurring at the same time.

These techniques are robust and they remain consistent in the face of node and communication failures including network partitions. Different quorums for read and write operations can be defined and different weights, including zero, can be assigned to every copy. This form is called *weighted voting*. Quorum consensus has received a great deal of attention and many protocols present variations of it.

2.1.3 ROWA and Available Copies

The *Read One Write All* (ROWA) protocol [Bernstein 84] requires a write operation to be executed on all copies synchronously, while a read operation can be executed at any copy. This method can be viewed as QC with $w = n$ and $r = 1$. It improves read availability by reducing write availability. This approach does not tolerate node crashes for updates, hence, it offers low reliability compared to other techniques.

The *Available Copy* protocol [Chan 86, Goodman 83] is a variation of ROWA, where a write operation updates only copies that are available, ignoring copies that are down. Therefore, it increases write availability. However, it cannot tolerate network partitions since different partitions can update different copies leaving the system in an inconsistent state.

2.1.4 Virtual Partitions and Dynamic Voting

Algorithms have been devised for increasing availability in the face of site failures and network partitions.

In the *Virtual Partitions* technique [El-Abbadi 85, El-Abbadi 86, El-Abbadi 89], the basic idea is for each site to maintain a *view* consisting of the sites it believes it can communicate with. Operations can proceed only at nodes belonging to the *majority partition*, which is the partition that contains more than half of the nodes in the system. Views of sites change as site and communication failures occur. View management protocols are devised, based on two- or three-phase commit protocols, to ensure consistency of the views at all replicas. Each object is assigned read and write *quorums* which are redefined based on the current view to maximise availability, while ensuring that quorums can be formed in only one of the partitions, namely, the majority partition.

Dynamic voting schemes [Davcev 85, Jajodia 87] redefine the *majority partition* to be the partition containing more than half of the *current replicas* rather than the whole system. Algorithms are presented so that a node can decide whether it is in the majority partition or not based on how many sites it can communicate with and the new quorums are computed accordingly. Operations are allowed to proceed only in the majority partition. In [Davcev 89, Jajodia 90], dynamic voting has been extended by assigning weights to replicas and allowing operations to proceed even if only one copy is available while guaranteeing that quorums cannot be formed in more than one partition.

2.1.5 Logical Structured Protocols

Protocols have been proposed that exploit a logically structured set of replicas with the objective of reducing quorum sizes. More precisely, the Hierarchical Quorum Consensus, the Tree Quorum and the Grid protocols have been proposed. In the following is a brief description of each.

The *Hierarchical Quorum Consensus* protocol (HQC) [Kumar 91] is based on constructing a logical multilevel tree, of depth m, where the physical copies are stored only at the leaves and the higher level nodes correspond to logical nodes. A read (write) quorum is formed by assembling r_i (w_i) quorums from each level of the tree such that $r_i + w_i > l_i$ and $2w_i > l_i$ for all levels i, where l_i= number of nodes at level i. An m-level tree with degree equal to three results in read and write quorums of size $2^{log_3 n}$ versus $\lceil \frac{n+1}{2} \rceil$ in the case of majority voting. However, in terms of availability the algorithm does not perform very well since it can block if specific nodes are down.

In the *Tree Quorum Protocol* TQP [Agrawal 90], replicas are organised into a tree of height h and degree $2d+1$. A write quorum is formed by selecting the root and a majority of its children; for each selected node the majority of the children is also selected, until reaching the leaves. A read quorum is formed by selecting the root of the tree; if it is inaccessible due to failure, a majority of the root's children forms a read quorum; if any of the selected nodes fail, a majority of its children is also required to form a read quorum, and this occurs recursively. The read quorum has a size of one, namely the root, under a failure free system, and increases up to $(d+1)^h$ under failures. The write quorum size is $\frac{(d+1)^{h+1}-1}{d}$, which is generally less than the majority of copies. However, if more than a majority of the copies in any level of the tree become unavailable, write operations cannot be executed. For example, if the root of the tree is down, no write operations can proceed. Therefore, the protocol provides low cost for read operations, however, there is an imbalance between the cost and the availability of reads and writes. Further, it may produce a bottleneck at the root and finally it is vulnerable to failure of the root.

In the *Grid Protocol* [Cheung 90], the nodes are arranged in a rectangular grid. A read quorum is defined to be any set of nodes that includes a representative from every column of the grid, and a write quorum is defined to include any read quorum plus an entire column of the grid. For square grids, the size of read quorums is $\sqrt{n}$ and the size of the write quorums is $2\sqrt{n} - 1$. This protocol is vulnerable to the failure of an entire column or row in the grid, hence, it incurs a penalty of reduced availability.

In [Rabinovich 92], a mechanism is presented that allows the reconstruction of the logical structure if failures occur and the new read and write quorum sets to be recomputed. It relies on performing a special operation periodically *epoch checking*, initiated by an elected site, which polls all replicas and, from the responding nodes, it forms a list of alive replicas, referred to as *new epoch*. The list of the new epoch members along with the epoch number are then sent synchronously to every member of the new epoch.

2.2 Asynchronous Replication Protocols

Many asynchronous protocols have been suggested. In this section, the well known protocols are reviewed. Their propagation schemes and their policies to reconcile conflicts are outlined and their limitations are discussed.

2.2.1 Grapevine, Clearinghouse and GNS

Grapevine [Schroeder 84] and Clearinghouse [Oppen 83] are early examples of replicated directory systems that achieve eventual consistency. A node originating an update, executes it then propagates it to all other nodes using an unreliable multicast. Each update associates the item with a unique timestamp and during propagation the most recent version, as determined by timestamps, is the one retained. To ensure reliable delivery, copies of the databases are exchanged and merged periodically. Each comparison must span *all* copies in order to ensure reliability. However, the periodic updates impose a large load on the network since each comparison involves sending a complete copy of the database to every other node.

In the Global Name Service (GNS) [Lampson 86], a replicated naming database, 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. The sweep operation is expensive, since it has to collect all updates from all replicas, and cannot be executed frequently, hence, the speed of propagation is slow. Further, in case of failures, partitions or addition of a new replica, the ring has to be reformed from scratch. The reformation process must be controlled by an administrator during which normal operation is suspended. That is, dynamic reconfiguration is not supported.

2.2.2 Epidemic Replication

In [Demers 87], several randomised 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, rumour mongering* and *anti-entropy.* Direct mail propagates an update to other replicas using a single unreliable multicast datagram. A replica can use rumour mongering by selecting another replica randomly and sending it one or more *hot rumours*, again using unreliable datagrams. Hot rumours are recent update messages that the replica believes the other is unlikely to have observed. Several stopping rules are suggested to ensure that a message does not continue propagating forever, but none of the rules can ensure that a message has been propagated to every replica before stopping. 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.

Resolving conflicts is solved by associating each data item with a global unique timestamp (denoting the GMT) and the value with the latest timestamp is kept. The techniques can be combined, using direct mail or rumour 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, which would not be acceptable for large databases nor for large scale systems. Further, combining several techniques means that a node can receive a message more than once, hence placing more load on the network.

2.2.3 Wuu's Algorithm and Two Phase Gossip

Wuu and Bernstein in [Wuu 84], present a propagation scheme for maintaining a replicated dictionary using logs. Each replica keeps a two dimensional time table (2DTT) recording what messages have been received by other replicas. The 2DTT allows the exchange of missing updates only rather than the whole copy of the database. Periodically, each node i sends to every node j a message containing its 2DTT and all messages in its log that i believes j does not have. Node j receiving this message, extracts the new messages and updates its 2DTT by merging it with the incoming one. Although an attempt is made to send to a node messages that this node has not received, redundant messages are still sent as 2DTT is just an approximate view of what other nodes have. Another deficiency is that the 2DTT, of size n^2, is sent as part of each message, which incurs a high communication overhead especially for large scale systems. Further, the speed of propagation is very sensitive to the activation period.

In [Heddaya 89], the *Two Phase Gossip* protocol is presented for propagating messages and controlling the size of logs used to represent replicated data. Messages are propagated in the same way as in [Wuu 84] but the status stored and sent on every message is reduced by a factor of n, based on the assumption that clocks are synchronised. More precisely, instead of storing n^2 timestamps, each replica i has to store two vectors o_i and w_i, each of size n. Entry $o_i[j] = t_1$ means that node i has received all messages that j had at $t \leq t_1$, and entry $w_i[j] = t_2$ means that node i knows that node j has received every message with timestamp $\leq t_2$. Periodically, node i sends a gossip message to every other node j including o_i and w_i and every message with timestamp $> w_i[j]$. Node j receiving the message, extracts the new messages and merges o_j with o_i and w_j with w_i taking the element-wise maximum. Therefore, each message carries a stamp of size $2n$ rather than n^2. However, this optimisation is feasible only if clocks are synchronised. Further, the vector w contains a lower bound on what other nodes have observed. Therefore, in comparison with [Wuu 84], the Two Phase Gossip causes more duplicate messages to be sent and also it purges the log less quickly.

2.2.4 TSAE

In [Golding 92c], a weak consistency replication protocol is proposed, called *Time Stamped Anti Entropy* (TSAE), which is based on the epidemic anti-entropy protocol [Demers 87] and uses a set of data structures much like those in [Wuu 84] and [Heddaya 89] to exchange updates. 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. The protocol ensures that replicas eventually converge to a consistent state during normal operation and when recovering from failures. Reconciliation methods based on FIFO and total delivery orderings are provided and group membership protocols are described to allow a node to join and leave the group of replicas dynamically. TSAE has been used to implement the replicated bibliographic database *Refdbms* [Wilkes 91] developed at Hewlett-Packard Laboratories, as well as the *Tattler* system [Long 92], a distributed monitor for the Internet. Both applications will be discussed in Section 2.3.

Although TSAE attempts to reduce sending duplicate messages by exchanging states and then deciding which messages to send, still redundant messages are exchanged during normal operations. This is because a node can participate in several sessions concurrently and also because the status messages are out-of-date due to network delays. The amount of redundancy increases when TSAE is combined with the best-effort multicast. Also, the status messages exchanged place an extra load on the resources. Further, like other replication mechanisms, the scheme cannot scale to a large number of replicas since it is assumed that any node can communicate with any other node. This protocol will be compared to our proposed protocol and will be described in more detail in Chapter 7.

2.2.5 Lazy Replication

Liskov and Ladin in [Liskov 86] 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 observation vector o_i used in [Heddaya 89]. Periodically, each replica sends a gossip message containing its timestamp and its message log to *all* other replicas and replicas are made consistent by merging the states and timestamps. Total and causal orders are supported by relying on a central node generating the sequence order. In [Ladin 90, Ladin 92], an extension is made that allows clients to define exactly what causal relations should be enforced between messages, so weaker orderings can be specified and the service is responsible for scheduling operation execution to respect that order. This method has been used by applications such as garbage collection in a distributed heap [Ladin 89], and locating movable objects in distributed systems [Hwang 88].

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 well as every node has to communicate with *all* nodes. Further, it needs an extra log to prevent executing duplicate updates, hence, using excessive storage. Finally, the scheme suffers from a slow speed of propagation due to the periodic updates.

2.2.6 OSCAR

OSCAR - Open System for Consistency and Replication - [Downing 90b, Downing 90a] implements weak consistency replication using a mixture of distributed and centralised 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. A replicator uses this vector to detect messages it has missed and ask other replicators for them. This vector is also used to decide when to purge a message from the log. Reconciliation methods based on FIFO order and overwrite order (latest timestamp wins) are supported to resolve conflicts. A negative side of the scheme is that the mediator must contact *all* replicators, then, each replicator has to contact other replicators to extract the missing messages, which causes a large network traffic. Further, the mediator may become a bottleneck and prohibit scalability.

2.2.7 Quasi-copies

In [Alonso 88, Alonso 90], a weak consistency replication protocol for information retrieval systems is presented. The notion of *quasi-copies* is introduced, which 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. For instance, a user can specify that the value of a copy should differ from the *real* value by no more than a constant, or that it should not be out-of-date by more than some period or number of versions. The scheme relies on a central location, 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 [Barbara 90], the model is extended to involve multiple central sites where the database is divided into segments, each one controlled by a single node.

Although this approach allows for controlling the inconsistencies of the replicated data, it relies on centralised components which cause bottlenecks in the system, limit the update availability and do not scale for large numbers of replicas. 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 is not taken into account.

2.2.8 Epsilon Serializability

Pu and Leff [Pu 91a] present an asynchronous replication approach that applies an extension of serializability called *epsilon-serializability* (ESR) [Pu 91b]. ESR is a correctness criterion which allows temporary and bounded inconsistency in replicas to be seen by queries. In contrast to our protocol and to the other mechanisms described in this section, this model is concerned with transactions where several operations must be executed as a group.

Four replica control methods are presented that use operation semantics to increase concurrency. The *ordered updates* method executes update transactions in the same order at every replica. However, a reliable and ordered message delivery is not provided and is 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. Finally, 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. These four methods are supported by our proposed protocol.

Methods for bounding the amount of inconsistency seen by queries are presented based on

ESR. The divergence is measured by the number of concurrent update transactions with which the queries interleave. For ordered updates, each query is given a global order number as if it is an update. Each time a query overlaps an update, an inconsistency counter is incremented. When the counter reaches a predefined limit, the queries are forced to run in total order. However, the details of the algorithm are left to the global ordering mechanism adopted. For commutative updates, transactions can acquire a certain number of read-write and write-write locks on objects - locks that are disallowed under strict serializability. If a transaction attempts to acquire more conflicting locks than the limit, the transaction is blocked. However, locking is not a good technique for a large number of replicas.

Although updates occur at one replica and propagate asynchronously to other replicas, it is not specified how propagation occurs nor how eventual reliability is reached. However, the definitions of operation compatibility presented can be used to build conflict reduction mechanisms in weak consistency systems.

2.2.9 Escrow Techniques

Escrow techniques have been proposed to increase concurrency and throughput in replicated databases for a class of applications involving resource allocation. In such applications, it is required to allocate a number of resources such that the total number of allocated resources is less than the total number of available resources. Typical applications are inventory control systems controlling the quantity of an item in stock, airline reservation system managing the number of seats available on an airplane, and so on. These techniques exploit the commutativity property since transactions access aggregate fields which are updated by positive or negative incremental changes.

The *Escrow Transaction* (ET) algorithm presented in [O'Neil 86], requires locking a quorum of a majority of sites. Although locks are not held for the entire duration of the transaction, still this approach lacks site autonomy. The Site Escrow (SE) techniques proposed in [Kumar 88, Kumar 90] consist of allocating the amount of available resources in an aggregate field across all replicas. A withdrawal transaction can successfully complete at a site only if the number of resources allocated at that site exceeds the number of resources that the transaction requires. If the amount available in the local escrow pool is not sufficient, the transaction would attempt to borrow from the escrow pools at other sites. This approach results in more site autonomy than ET, since each site can deplete its allocated resources independently without having to consult any other site. However, the global state of the system should be computed periodically by executing a global snapshot algorithm such that the escrow quantities are adjusted at each site to reflect the consumption of the resources. Transactions cannot proceed while the global algorithm is being executed, hence, its frequent execution may itself degrade performance. Further, it requires the synchronisation of *all* nodes which is a handicap for scalability.

The *Generalised Site Escrow* (GSE) algorithm [Krishnakumar 92], modifies the notion of SE and eliminates the need for a global state algorithm. It employs a combination of quorum locking [Herlihy 87] to control the number of transactions occurring simultaneously, with broadcasting updates through gossip messages using the algorithm of Wuu and Bernstein [Wuu 84] (see Section 2.2.3). The idea is to use the 2DTT to decide which transactions are not yet known to other sites. So, a site i makes a conservative estimate of its escrow by estimating an upper bound on the escrow values of other sites, taking into account the resources allocated by transactions known to i but not yet known to these sites, in order not to violate the allocation constraint. If the escrow at site i is not sufficient then i attempts to enlarge it by locking additional sites. In the worst case, all sites are in the quorum, at which point the transaction can be initiated. GSE requires gossip messages to be piggybacked frequently so that new allocations become known to other sites more quickly, thereby reducing the quorum size. However, these messages place an overhead on the network, especially that they carry redundant updates.

The *demarcation protocol* proposed in [Barbara 92] addresses the same problem, but the computation of escrow is based on agreement between two nodes rather than on estimation. A limit L is associated with each item and a transaction can withdraw units as long as the final value is above L. Two kinds of operations were introduced: *safe operations* which a node uses to raise its limit and *unsafe operations* to decrease the limit. A node can perform *safe operations* locally then inform the other nodes about the change. Whenever a node wishes to perform an *unsafe operation*, it requests another node to perform a corresponding safe operation, waits for notification, and then, it can lower its limit. The protocol for changing limits is more complex than the GSE protocol, especially for more than two nodes, but it yields less conservative escrow values.

2.2.10 Currency Tokens

In [Tait 92], Tait and Duchamp introduce a method for managing a replicated file system to address the problem faced by mobile clients. They attempt to use asynchronous operations and let the clients determine the level of consistency desired. Their model is that of a primary-secondary server organisation. To perform an update, a client writes the update in its cache and continues execution. The primary node performs periodic *pickups* from the client's cache and propagates the updates to the secondaries. A reliable communication mechanism is assumed to be available. To read a file, the interface provides a *loose-read* which reads from any copy with no guarantee concerning the value returned, and a *strict-read* which returns "the most recent value". However, the *strict-read* requires synchronisation with at least the majority of the secondaries as well as with all caches, hence, it results in mediocre performance. Further, to ensure consistency of the read value, *every* write must be preceded by a *strict-read* to update a fresh copy of the file and to declare itself as a potential writer. Therefore, although updates are asynchronous, they have to be preceded by all the synchronised access of the *strict-read* which incurs a large overhead and the benefits of the asynchrony are lost. To reduce the cost of *strict-read*, an optimisation is given by offering a *currency token* to replicas that have just performed a *strict-read*. A client possessing a currency token, can read consistent data from its local cache provided that there are no potential writers. However, this optimisation is beneficial only if writes are rare since any write attempt will revoke the currency tokens and everybody is forced to *strict-read* again.

2.2.11 Coda

The Coda distributed file system [Satyanarayanan 90, Kistler 92] uses an optimistic replication protocol to manage distributed file services and support disconnected operations. It uses the synchronous *Read One Write All* strategy during normal operation and allows operations to occur asynchronously only if partitions occur. Synchronisation 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 [Kumar 91a]. 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.

2.2.12 Isis

Isis [Birman 87, Birman 91] is a distributed programming toolkit that provides atomic, interactive delivery with total or causal message ordering. It has been used to develop many applications, including replicated file systems. Processes use a group membership service to join and leave process groups, and a process can belong to more than one group. Group multicast is provided using either the ABCAST totally-ordered multicast protocol or the CBCAST causally-ordered protocol. Isis is based on virtual synchronous process groups and ensures strong consistency of group views at the expense of latency and communication overhead. It is therefore, unlike weak consistency mechanisms, aimed towards small systems that must provide consistent, interactive service.

2.3 Applications using Weak Consistency

Asynchronous approaches trade availability and response time for consistency. This approach is suitable for applications that need to be massively replicated and where high availability and performance are more crucial than strong consistency. In general, in these applications the semantics of the operations can be used to tolerate inconsistency and it is not necessary for every replica to observe updates immediately as long as updates are propagated without too much delay. Many applications fall into this category and in this section some of them are discussed.

As mentioned in Section 2.2.1 naming systems have been using asynchronous protocols since it is more important to get a fast answer than to be guaranteed to get the absolutely latest answer. It would not be tolerable for a user to be unable to lookup someone in MIT to find their mailbox because not all the replicas in South America were known to be up-to-date. Most changes to the naming database are non-conflicting or if they conflict they can easily be reconciled. For instance, while updating the password or the mailbox site of a user, the update with the highest timestamp is retained. Adding two users with the same name, although this rarely occurs, can be solved by appending a hidden timestamp to make all names unique. Further, the data is self-validating and can be treated as *hints*, i.e using stale naming data at the application level is acceptable. For example, if an enquiry about the Internet address of a server returns an obsolete value, the user can detect the error when he or she tries to connect to the server.

Distributed information services is one of the applications that must scale to the vast number of users that can access a shared information system. For example, Archie [Emtage 92], a directory service for Internet FTP archives, receives in the order of 100 000 queries per day [Bowman 94]. Archie is a specialised service with a limited audience as compared to widely-used services such as library cards and catalogues which receive around 100 queries per second. Another example is the WAIS (Wide Area Information Service) [Kahle 91], a text retrieval system which covers over 255 publicly registered databases available by Internet sites from around the world including: full-text papers of periodicals and conferences, weather satellite pictures, poetry archives, an index to journals periodicals, etc. Such systems must be highly replicated to meet their scalability and performance requirements. For instance, Archie with a dozen replicas, responds to queries in seconds on a Saturday night, but it can take five minutes to several hours to answer simple queries during a weekday afternoon. To yield a five-second response times during peak hours, an estimation of 1800 replicas is needed [Bowman 94]. In order to update an entry in the database of those services, it is impractical and unnecessary to freeze large number of copies. Further, if a user contacts a stale replica, a correct answer will be obtained, but may not be the most up-to-date.

A typical information service that is implemented using a weak consistency protocol is the *Refdbms*, a distributed bibliographic database developed at Hewlett-Packard laboratories [Wilkes 91, Golding 92b]. A user enters a reference in one replica and the reference is propagated to all other replicas using the TSAE protocol [Golding 92c]. Users can search databases by keywords, locate copies of papers, and add, change, or delete references. Fields of different entries are grouped together, and a separate reconciliation policy is used for each group's fields. These policies are based on message ordering delivery. For example, changes to author-related fields overwrite old ones, while adding or deleting locations of copies can be done in any order since they are commutative. Some operations can result in conflict, such as adding two different references with the same tag, or changing one reference in two different ways. These conflicts are handled by processing update messages in the same order at every replica, i.e. in total order. While users search for references, searches need not return completely current information; that is, eventual message delivery is acceptable for such an application.

The *Tattler* system [Long 92], a distributed monitor for the Internet, adopts a weak consistency replication protocol. It monitors a set of Internet hosts, measuring how often they are rebooted and for what fraction of the time they are available. Only one operation updates a Tattler database: merging a set of samples. When a site obtains a new sample, it logically multicasts the sample to the other sites. Each sample represents an interval when the host was known to be available. A sample that is being merged into a database will be either disjoint from every other sample recorded for the same host, or it will overlap with another sample. If it overlaps, the two samples are combined. Otherwise, the host has been rebooted and a new interval has begun. Sample merging is commutative and idempotent, so message ordering is unimportant as long as messages are delivered reliably.

One of the most obvious services that uses massive replication and weak consistency is Usenet, the Internet news systems [Kantor 86]. In bulletin board applications, the main consistency requirements are that messages generated by a node must be seen by all other nodes in the 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 causal ordering of messages.

The French Civil Aviation Administration has developed a new flight plan service for air traffic control systems in CENA [1] [Queinnec 93]. The systems are intrinsically geographically distributed and impose strict constraints on availability: a failure should never interrupt the service and each operational site should be able to provide service even in stand alone mode. The flight plan service is not involved in keeping planes from colliding. Instead, it maintains the description of the flights, such as model of the aircraft, the current position of the flight, the current speed, and it is responsible for planning arrivals, scheduling gates and so on. The large size of the system and the high requirements of availability call for replication. However, up-to-date information is not necessary as the fluctuation of performance is more crucial. Controllers would like to know information about the current progress of flights but it is useless to get the accurate position of a plane after a long delay. Therefore, weak consistency is adopted and rumour mongering techniques [Demers 87] are used to propagate flight data to controllers. Further, updating flight data is limited to one or few sites at a time, and moves with time. Therefore, the degree of write-sharing is small and a new update overwrites an old update according to a timestamp associated with the message. The implemented system involves eight replicas but it is aimed for hundreds.

Weak consistency replication is also adequate for a mobile computing environment. If replicas reside on mobile computers, the user can perform updates locally during disconnection. When reconnected, mobile replicas will exchange updates with the other replicas. Other applications include stock exchanges, railroad scheduling, keeping routeing tables for mobile computing and so on.

[1] Centre d'Etudes de la Navigation Aérienne

2.4 Summary

Many replication protocols have been proposed. Synchronous protocols provide strong consistency, but they require synchronisation among a large number of replicas which imposes limitations on performance. Recently, several asynchronous protocols have been suggested that trade performance and availability for consistency. This approach is suitable for applications that can tolerate inconsistent or out-of-date information.

In the proposed asynchronous protocols, several limitations were found. First, only few of them provide eventual delivery, i.e. a guarantee that a message will be received by all replicas despite failures. Many of them rely on the availability of reliable communication delivery. Second, all protocols assumed that a node can communicate with all other nodes in the system, which is unrealistic in internetworks. Third, most of the protocols involve varying degrees of redundancy, which increases communication overhead and wastes network bandwidth. Fourth, most of them experience a slow speed of propagation because they rely on periodic updates. Finally, they cannot provide the user with more up-to-date information if the results returned from the local replica are unsatisfactory.

The protocol presented in this dissertation improves on these systems by overcoming these limitations. In particular, it provides an efficient propagation scheme that is based on a logical hierarchy, where a node communicates with few nodes only, while ensuring reliable delivery. Further, it does not involve duplicate messages to be sent and relies on immediate propagation to speed up dissemination of updates. Also, it allows locating replicas that are more up-to-date than others, if desired.

Chapter 3

Hierarchical Replication Protocol (HARP)

This chapter presents a Hierarchical Asynchronous Replication Protocol (HARP) that scales well for thousands of replicas while ensuring reliable delivery [Adly 93b]. It provides an efficient and scalable propagation scheme that is based on replicas organised into a logical hierarchy. The logical hierarchy and the motivation behind choosing a hierarchical structure are presented in Section 3.2. The propagation scheme is described in Section 3.3. A new service interface is proposed in Section 3.4 that provides asynchronous operations, that commit at any replica then propagate to other replicas, as well as 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 offer different levels of staleness, depending upon the needs of various applications. This property is discussed in Section 3.5. Reconciliation methods based on delivery order mechanisms are provided to resolve temporary inconsistencies, resulting from asynchronous updates, and an application may choose from them (Section 3.6). Finally, Section 3.7 discusses some issues regarding the design of the hierarchy.

3.1 The System Model

The system consists of n nodes connected by an internetwork. The set of nodes is denoted by N. Nodes communicate by exchanging messages; there is no global shared memory. Processors may fail then restart, however, fail-stop processors [Schlichting 83] only are assumed and Byzantine failures [Lamport 82] do not occur. Nodes communicate using point-to-point and multicast messages; the latter may be transmitted using multiple point-to-point messages if no more efficient alternative is available. The underlying communication network is

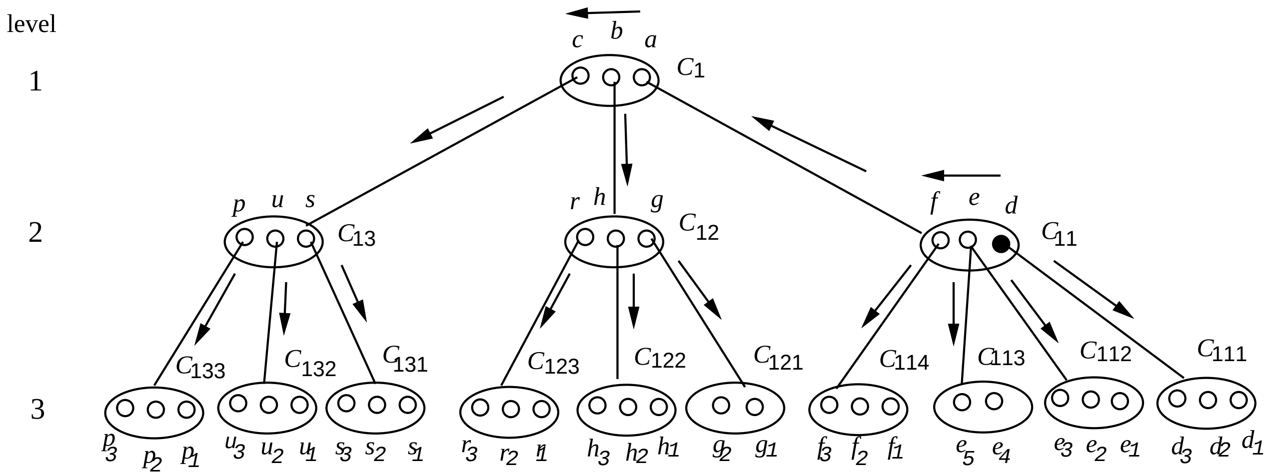

Figure 3.1: 40 nodes organised in a hierarchy of three levels

unreliable: it may lose or duplicate messages and does not guarantee any order of delivery. However, it is assumed that messages are received incorrupted. Link failures can cause the network to be partitioned. These partitions are eventually merged again. 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 acknowledgement after a certain timeout period. Each node has a local clock, but clocks are not necessarily synchronised.

3.2 The Logical Hierarchical Structure

Replicas are organised 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 parent node in its parent cluster. Figure 3.1 shows 40 nodes organised into a hierarchy of three levels [1]. Nodes in the same cluster, referred to as *neighbours*, have fast and efficient communication between them as well as with their parent. The parent, neighbours and children of a node are together referred to as its *correspondents*.

The idea of adopting a logical hierarchical structure has been based on the following grounds:

- Existing replication protocols do not scale well because they manage replicas as a single, flat group of nodes requiring a node to communicate with all other nodes, which is unrealistic for wide area networks where replicas are geographically dispersed. On the other hand, within a hierarchy, a node communicates with few nodes only which

[1]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

reduces message traffic, improves latency, reduces the amount of information kept, balances the load among the nodes and enhances scalability of the system.

- It is desirable to provide strong consistency while limiting synchronisation to a small number of nodes to achieve an acceptable performance. This can be realised by designating the nodes in the top cluster of the hierarchy as the nodes from which synchronous operations assemble quorums.
- It is desirable to design an asynchronous replication protocol that can provide the user with different levels of staleness. This cannot be done if propagation relies on random patterns since there will be no way to distinguish copies that are more up-to-date than others.
- By fixing the pattern of propagation, the scheme can ensure reliable delivery with no redundancy which makes efficient use of the network bandwidth. Further, this can be achieved while relying on immediate propagation, rather than delayed propagation, which speeds up the diffusion of updates.
- While restructuring the network, due to a node joining or leaving the group of replicas, it is possible to limit the network traffic and the general overhead associated with the reconfiguration to the nodes belonging to the cluster the node is joining or leaving, rather than all nodes in the network. This issue will be addressed in detail in Chapter 4.

This idea has been supported by the observation that large scale networks (internetworks) are usually connected in a hierarchical fashion. For instance, the JANET network in the UK, has a mesh of 8 NOCs (Network Operation Centres) densely connected, forming the backbone. Sites of regional networks are connected radially (as a star) to their NOC and they feed local nodes connected in local area networks. The same applies to the Internet. In a recent study measuring the performance of transaction message delivery in the Internet [Zhang 94], it has been shown that communication delays follow a hierarchical behaviour. By measuring the message round-trip times over 2000 hosts on the Internet, the results have shown the *clustering effect* in the Internet topology: hosts in the same local network have similar round-trip time (around 100 μs to 10 msec) and variation between any two hosts is very small. Further, communication between one site to many different sites in another local network have similar performance (around 100 msec to 10 sec) which could be represented by a major host of that network. Similar results were reported in [Golding 91]. Therefore, the logical hierarchy should be built to reflect the physical topology, grouping nodes that have fast and efficient communication into one cluster in order to exploit *localised communication*. For instance, nodes connected through the same LAN would belong to the same cluster at a low level of the tree and nodes in the backbone would form the top cluster of the hierarchy.

3.3 The Propagation Scheme

Classic propagation schemes fall into two categories: *flooding techniques*, and *probabilistic techniques*. The Internet news (Usenet) [Kantor 86] uses the flooding approach to distribute updates among its thousands of replicas. The propagation is based on the notion of "upstream" and "downstream" sites. A site can attache itself to one or more sites and start receiving news feeds from them. The receiving site becomes a "downstream" site for the sending "upstream" sites. To ensure message delivery if failures occur, a node attach itself to more than one "upstream" sites, receiving simultaneous feeds from them, and then eliminates duplicates. The flooding approach suffers from incurring a large number of messages and lots of redundant traffic which wastes network bandwidth.

The *probabilistic technique* has been used by most of the previous weak consistency protocols. The technique relies on *delayed propagation*, where nodes exchange status and new messages periodically in sessions. Partners in these sessions are selected from a probabilistic function. These protocols assume that a node can communicate with any other node. This would be acceptable for small networks with a few replicas, but it will incur a large communication overhead for wide area networks. Further, since a node sends not only messages it originates, but also messages it has received from other replicas, there is redundancy with varying degrees.

In HARP, the propagation scheme is based on the logical hierarchy described in Section 3.2 and is very simple: a node i, originating a message, sends it to its neighbours, parent and children. Each receiving node j passes the message to the next level as follows: if the message comes from a neighbour or from the parent, then j sends the message to its children; else, if it comes from a child then j sends it to its neighbours, its parent and to its children 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 3.1 shows the path that a message would follow while propagating if it originated at node d.

In the following, the data structures used are described. Then, the details of the propagation algorithm are presented.

3.3.1 Data Structures

The protocol requires each node to maintain a set of data structures which are assumed to be kept on stable storage to survive failures. The set of data structures kept at node i are summarised in Table 3.1.

- A *Log* keeping the set of messages received. When a message m 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	Log of all messages received
VV_i	Version vector
LL_i	Vector of linked lists
$AckM_i$	Acknowledgement matrix
$View_i$	Structure describing the hierarchy

Table 3.1: Data structures kept at node i in HARP

$$Log_i = \text{list of } (uid, m, p_1, p_2), \qquad \text{where}$$

- uid is a unique identifier assigned to the message m by its originator,

  $uid{=}\langle m\_org, m\_seq\rangle$, where

  - m_org is the node id that originated m, and
  - m_seq is the value of a counter kept by node m_org and incremented by 1 to generate a new uid
- p_1 is a flag set to 1 if the message has been applied to the local database, and
- p_2 is a flag set to 1 if the message has been already delivered to all nodes.

A message can be purged from the log only when both flags are set to 1.

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

  – VV_i, a *Version Vector* of dimension n, where $VV_i[j] = k$ means that node i has received all messages which originated at j up to message number k.

  – LL_i, a *Vector of Linked Lists* of dimension n, where $LL_i[j]$ is a linked list keeping the uid of messages received from node j out of FIFO order. LL_i will be empty if all messages are received in FIFO order.

  VV and LL are updated upon the generation or receipt of any message. When a node i generates a new message, $VV_i[i]$ is incremented by 1. When a node i receives a message m that originated at node j, it increments $VV_i[j]$ if m is received in FIFO order. Otherwise, the uid of m is inserted in $LL_i[j]$ until the missing messages arrive. VV and LL represent the history of what messages the replica has received. They are used by the reconciliation methods as well as by the reconfiguration algorithms to determine which messages are missing from a node. Also they are used to detect duplicate messages.

- $AckM$, an *Acknowledgement Matrix*, 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 $AckM_i[j, r] = l$ this means that node i knows that node j has learned of all the messages that have originated at node r up to message number l. Basically, row j in $AckM_i$ is a copy of VV_j as seen by i. $AckM$ is similar to the 2DTT used in [Wuu 84].

- $View_i$, 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,
  - P, the node id of j's parent,
  - $Child\_Cids$, the set of cluster ids that are children of j,
  - up, a flag set to 1 if the node j is up; 0 otherwise, and
  - r_seq, a sequence number assigned by j.

$$View = \text{list of (row)}$$
$$\text{row} = \text{list of } (Cid, P, Child\_Cids, up, r\_seq)$$
$$Child\_Cids = \text{list of (cluster ids)}$$

Table 3.2 shows part of $View$ for the logical hierarchy shown in Figure 3.1 as it should appear at each node. In the rest of the dissertation, $View_i.a$ will be used to denote the row of node a in the $View$ of node i and fields of $View$ will be referred to as follows: $View_i.a.Cid = C_1$, $View_i.e.Child\_Cids = \{C_{112}, C_{113}\}$, and so on.

Node	Cid	P	$Child\_Cids$
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 3.2: A part of the $View$ data structure

The hierarchy and the locality of propagation allows for optimisations to reduce the size of $AckM$ from $O(n^2)$ to $O(n)$, to purge the Log more quickly and to reduce the size of the $View$ structure. This issue will be discussed in detail in Section 4.6.

It should be noticed that vector and matrix notations have been adopted to describe the data structures used for simplicity of exposition, and for consistency with the literature. Since the size of these structures is dynamic, and since each entry needs to be associated with the corresponding global node identifier, a functional notation would have described the structures more precisely. However, in order to retain the terminology of the literature, for example TSAE, OSCAR, Isis, etc, it was decided to use vector and matrix notation.

3.3.2 The 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 i, originating a message m, increments its entry in VV_i and assigns m the next *uid*. Next, i stores the message in Log_i, sends the message to its neighbours, parent and children and waits for acknowledgements (timeouts and retransmissions are used to overcome lost messages). Also, it applies the update operation to the database, when the reconciliation method allows it, and sets p_1 for m in Log_i to 1. Each node j receiving a message checks if it is a duplicate, and if so, discards it. Then, it updates its VV_j and LL_j from the *uid* of the message, inserts the message into Log_j, and sends an acknowledgement to the sender. Node j proceeds in propagating the message such that, if the message comes from a neighbour or the parent, then j sends the message to its children; else, if it comes from a child then j sends the message to its neighbours, its parent and to its children of clusters other than the one the message comes from. Finally, it applies the update to the database according to the reconciliation method associated with the data item. This works recursively and a message originating at any site will eventually propagate everywhere. Steps taken by a node originating or receiving a message for propagation are described in Figure 3.2.

From the fixed hierarchical pattern of propagation that messages obey, it follows that: *If any node originates a message m and if no failures or reconfiguration occur, then following the above protocol, m will be reliably delivered to every node in the network and every node will receive m only once.*

Periodically, each node 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 $AckM_i[j,k] = \mathsf{Max}\ \{AckM_i[j,k], VV_j[k]\}$ for $k = 1$ to n.

To purge unneeded messages from the log, each node i summarises what other nodes know by forming a vector min_ack such that $min\_ack[k] = \mathsf{Min}_j\ \{AckM_i[j,k]\}$ for $k = 1$ to n. So, if $min\_ack[k] = q$ this means that all nodes have received and acknowledged all messages initiated from node k up to message number q. Therefore, node i sets p_2 to 1 for all messages in Log_i with $uid = \langle k, m\_seq \rangle$, where $m\_seq \leq min\_ack[k]$. A message is purged from the *Log* when both p_1 and p_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. As mentioned before, messages can be purged from the *Log* more quickly, relying on the fact that a node communicates with a few nodes in the hierarchy, as will be discussed in Section 4.6.

The described propagation 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 neighbours, parent and children). 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 message only

When a node i initiates an update m

```
{
increment VV_i[i];
stamp m with uid = ⟨i, VV_i[i]⟩;
insert m into Log_i (with p_1 = p_2 = 0);
perform the operation on the database and set p_1 = 1;
send m to j, where j = View_i.i.P ∧ View_i.j.up = 1;          /* parent */
send m to j, ∀j:  View_i.j.Cid = View_i.i.Cid ≠ null ∧ View_i.j.up = 1;     /* neighbours */
send m to j, ∀j:  View_i.j.Cid ∈ View_i.i.Child_Cids ∧ View_i.j.up = 1;     /* children */
set timeouts to retransmit the message if acknowledgements are not received;
}
```

When a node j receives a message m with $uid = \langle m\_org, m\_seq \rangle$ from node i:

```
{
If VV_j[m_org] ≥ m_seq or m_seq ∈ LL_j[m_org] Then
  discard m and stop;                /* duplicate message */
If VV_j[m_org] + 1 = m_seq Then {          /* m is in FIFO order, update VV_j[m_org] */
  increment VV_j[m_org];
  /* check LL_j[m_org] for messages that are now in the right order to increment VV_j[m_org] */
  While m_seq in the head of LL_j[m_org] = VV_j[m_org] + 1 {
    increment VV_j[m_org];
    delete first entry in LL_j[m_org];
    }
  }
Else                  /* m is out of FIFO order */
  insert m_seq into LL_j[m_org] in ascending order;
insert m into Log_j (with p_1 = p_2 = 0);
send acknowledgement to i;
If i = View_j.j.P or View_j.i.Cid = View_j.j.Cid Then
  /* i is a parent or a neighbour, send m to all children */
  send m to k, ∀k:  View_j.k.Cid ∈ View_j.j.Child_Cids ∧ View_j.k.up = 1;
If View_j.i.Cid ∈ View_j.j.Child_Cids Then {
  /* i is a child, send m to neighbours, parent and children of other clusters */
  send m to k, where k = View_j.j.P ∧ View_j.k.up = 1;          /* parent */
  send m to k, ∀k:View_j.k.Cid = View_j.j.Cid ≠ null ∧ View_j.k.up = 1;     /* neighbours */
  send m to k,             /* children not in i's cluster */
    ∀k:(View_j.k.Cid ∈ View_j.j.Child_Cids) ∧ (View_j.k.Cid ≠ View_j.i.Cid) ∧ View_j.k.up = 1;
  }
set timeouts to retransmit the message if acknowledgement are not received;
perform the operation on the database and set p_1 = 1 ;
}
```

Figure 3.2: Steps taken by node i upon originating or receiving a message for propagation

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 increases the speed of propagation. Finally, the system is fully asynchronous; that is, it does not block the sender until remote delivery occurs.

3.4 Levels of Asynchrony

Different applications require different degrees of consistency. Therefore, applications should be given the ability to choose the level of consistency that is appropriate for their particular semantics. As it has been pointed out recently in [Terry 95], current replicated data models lack the ability to provide the applications with some degree of control over the tradeoffs between consistency and performance.

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. More specifically, the protocol supports the following operations:

Fast_Write: a Fast_Write can be initiated and committed locally at any replica (the site of origin), and then propagated to other replicas using the propagation scheme.

Fast_Read: a Fast_Read returns the value read from any replica. It is associated with a parameter *r_node* denoting the node to read from. The default of *r_node* is the local node. If *r_node* is not the local node, then the local node sends the read request to *r_node* which performs the read and returns the results to the local node.

Slow_Write: a Slow_Write 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 i to perform a Slow_Write, it sends the message to its ancestor node j in the top cluster, which acts as a *coordinator*. The coordinator assembles a quorum from its neighbours (using majority quorum consensus or any other standard method). Once the coordinator decides to commit, it assigns the message the next *uid* and sends commit to its neighbours and to the origin node i. Each node in the top cluster starts propagating the message by passing it to its children, excluding node i. Node i, receiving commit from the coordinator, updates its VV and LL, inserts the message in Log_i, applies the operation on the database then proceeds in propagating the message to its children.

Slow_Read: a Slow_Read returns the value read from a quorum of nodes of the top cluster. A node i, wishing to perform a Slow_Read, sends the request to its ancestor node j in the top cluster. Node j assembles a quorum from its neighbours and returns the read value to i.

Opt_Write: an Opt_Write is similar to Slow_Write, but it is applied to the database of the site of origin and, optionally, to some other selected replicas (called *optimistic nodes*) before commitment. However, these updates are 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.

This update is associated with a parameter *opt_nodes* denoting the set of optimistic nodes that apply the update before committing. These nodes usually are nodes in the neighbourhood of the origin node that like to observe the update as soon as it originates.

A node i wishing to perform an Opt_Write, applies the update on the database and returns control to the user. Then, it sends the message to the set of nodes *opt_nodes* as well as to its ancestor node j in the top cluster. An optimistic node receiving the message, executes the update. Node j, acting as a coordinator, assembles a quorum from its neighbours. If j decides to commit, then it assigns the message the next *uid* and sends commit to its neighbours, to the origin node i and to the optimistic nodes. Each node in the top cluster starts propagating the message by passing it to its children, excluding the origin and the optimistic nodes. The origin node or an optimistic node receiving a commit, updates its VV and LL, inserts the update in its *Log* and proceeds in propagation. Any other node will receive the update message through normal propagation; it updates its VV and LL, inserts the message in the *Log*, applies the operation on the database and proceeds in propagation. If j decides to abort the update, it sends an abort message to the origin node and optimistic nodes which then undo the update.

Slow_Write should be used by operations that are likely to result in conflict and where no conflict reconciliation is acceptable, such as some banking applications. Slow_Read, if used in conjunction with Slow_Write, will return the most up-to-date value. Opt_Write can be used by operations that require fast response and that conflict with very low probability; however, if conflicts occur their semantics can accept an abort. For instance, in air line reservation, the semantics can be used to provide site autonomy by associating an integrity constraint with the reservation operation; e.g. :

If *number of available seats* > 10 **Then** `use Opt_write`
    Else `use Slow_Write`

The probability of abort could be reduced by increasing the bound on the constraint. Other applications can use Opt_Write such as stock ordering, adding a user name in a naming service and so on. Also, it can be used instead of Slow_Write 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_Write 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 n, this will be equivalent to the conventional majority quorum consensus method, or whatever standard technique is 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 synchronise with is small. Applications that require a lower degree of consistency i.e. *weak consistency* can use Fast_Write, Fast_Read and Opt_Write which offer better performance since they are executed locally.

Combining strong and weak consistency has been introduced in the Cambridge Universal Name Service (UNS) [Ma 92]. A two-class name service infrastructure is proposed, where replicas are divided into first class servers and secondary servers. The first class replicas run a synchronous replication protocol based on quorum consensus [Lamport 89], while the secondary servers run the epidemic anti-entropy protocol [Demers 87]. Only the first class servers are allowed to carry out updates. Secondary servers are updated asynchronously by the first class servers through a two-way communication: the firsts use call-back to *push* recent updates to the seconds; and the seconds contact the firsts periodically to *pull* new updates and refresh their copies. Secondary servers are used as read-only copies, which may return out-of-date data. The system provides the client with the ability to get the most recent data, if desired, by reading a quorum from the first class servers. These guarantees are similar to the ones provided in HARP by the Slow_Write, Slow_Read and Fast_Read operations.

3.5 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 levels in the hierarchy.

More specifically, assume L is the number of levels in the hierarchy and D is the estimated delay (in time units) required to send a message from a node to its neighbours, parent or children. If a Fast_Read from a node at level l is used in conjunction with Slow_Write, 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, performed at a node in level l, is used in conjunction with a Fast_Write or an Opt_Write, then the answer will be at most $(L+l-1)D$ 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. Also, the information read at the nodes in the top level will be at most LD stale. Further, each node will be at most D time units in divergence from its neighbours, parent and children. 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 observe 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 a flight plan service of an air traffic control system such as the one developed at CENA [Queinnec 93] (see Section 2.3), controllers need to know information about the current progress of flights which are still many hours away so that they can plan arrivals, schedule gates and so on. However, the further (in time) controllers are from the current position of a plane, the less precision they need. Therefore, in order to carry out short-term planning, the needed flight plan data are those of the planes in their local sectors and they may also need to consult information about the neighbouring 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, scheduling trains and so on. Also, it could be useful for operations that are associated with a condition, such as requiring to read a replica updated to a certain timestamp, or if the data is self-checking e.g. looking for the Internet address of a server or searching for a specific reference in a bibliographic database. In these cases, if the results obtained locally are not satisfactory, then replicas in higher levels could be contacted to get the desired data.

3.6 Reconciliation Methods

Fast_Write 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, based on delivery order mechanisms, and an application developer may choose from them. Each data item (or parts of the database) 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, in the Tattler system [Long 92], a distributed monitor for the Internet, operations are commutative and do not conflict (see Section 2.3). Similarly, for some systems, certain data items can only be incremented or decremented. In this case, a node can apply the update to the database as soon as it is received as the order is irrelevant.

Latest-wins order: In this method, a new update overwrites an old value and an update trying to overwrite a more recent version is ignored. The *View* structure, used to describe the logical hierarchy, is one example of replicated data that uses this method. As will be seen in Chapter 4, when the row of node j changes in the *View* due to reorganisation, node j propagates its row in an *update_view* message, to inform every other node of the change. An *update_view* message is delivered only if it is a new update. Since by the nature of the algorithm only node j can propagate its own row then the *r_seq* field, which is a counter incremented by j for each change, can be used to decide whether the incoming message is new or old. This method is also appropriate for other applications such as updating a mail-box of a user, or updating the current location of a plane and so on. In the case where any node can update the data item, global timestamps based on synchronised clocks should be used to decide whether the update is new or old. Grapevine [Schroeder 84], the Global Name Service [Lampson 86], and the CENA air traffic control system [Queinnec 93] use 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 j should deliver a message originating from node i only if all previous messages originating from node i have been delivered. This method is useful for applications where a replica has autonomous control over its data items; that is, it is the only one updating this particular data item, however, these items should be read by other replicas in order.

Guaranteeing FIFO order is relatively simple. The basic idea is to number the messages at the source and to have destination sites deliver the messages in that order. 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, when the missing messages arrive. That is, a received message is not delivered immediately, but its delivery is delayed until the condition is satisfied.

Causal order: If two messages are causally related (one causes the other) [Lamport 78] 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 [Liskov 91, Ladin 90], some applications are suggested that can use causal order such as garbage collection in a distributed heap and locating movable objects in distributed systems. This ordering is of particular interest and it will be addressed separately in Chapter 5.

Total order: Causal order places no constraints on the relative delivery ordering of concurrent events while total order does. With total ordering, messages are delivered in the same order at every node and causal relations between messages are preserved. This is the strongest order; it is more costly than causal ordering, thereby requiring synchronous solutions.

Many methods have been suggested in the literature for supporting total order. Chang and Maxemchuck's [Chang 84] approach is that all sources transmit messages to a central site, which assigns sequence numbers to messages and forwards them to the destination sites, which deliver them following the order assigned by the central site. The central site is identified with a token, and this token circulates through the system. However, this approach does not scale. A similar approach has been used in [Ladin 90]. In Isis [Birman 91], the protocol relies on a coordinator (token-holder) and uses two causal messages: one to send the messages and the second is sent by the token-holder to indicate the order in which messages should be delivered. This protocol is quite costly as it requires an extra message (order message) to be broadcast for each totally ordered message sent. Further, these protocols suffer from the drawbacks of the centralised approach.

In TSAE [Golding 92c], the approach relies on delaying the message delivery until the replica guarantees that no messages with lesser timestamp will be received, which happens when the timestamp associated with the message is less than or equal the minimum timestamp in the summary vector. The disadvantage of this scheme is that messages suffer from a long delay until delivered: if a node does not send any messages for a while then all messages generated during this period at all other nodes will not be delivered. Further, it relies on synchronised clocks which is hard to achieve in a large scale system. Finally, it does not respect the potential causal relations. Not all solutions are reviewed here; related work can be found in [Cheriton 85, Kaashoek 89, Melliar-Smith 90, Garcia-Molina 91].

Total order delivery is supported in HARP by treating the message as a Slow_Write: a node requiring to send a totally ordered message, sends the message to its ancestor node in the top cluster, nodes of the top level vote to agree on the sequence number (order) to give to the message. A majority of votes is enough to select a unique number. 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. This method is similar to the one suggested by Birman and Joseph in [Birman 87]. However, in their method all nodes have to vote to agree on a unique sequence, as opposed to nodes in the top cluster only, which results in a large number of messages and increased delay which degrades performance.

3.7 Discussion

Designing the logical hierarchy is an optimisation problem that should take several factors into account, such as minimising the delay and cost of communication; taking into consideration the underlying topology, minimising the number of levels in the hierarchy to limit the propagation delays, balancing the load evenly among the nodes and selecting an adequate number of nodes for the top cluster. Also, nodes with higher capacity and reliability are better placed in higher levels of the hierarchy since they may be assigned more load. Another factor that might need to be considered is the volume of the traffic generated by the applications. One should aim at producing a *balanced* tree in the sense that the volume of messages generated from any sub-branch at the same level is roughly the same. Further, if there are a set of nodes that are expected to generate a higher rate of transactions than others, then those nodes might be better placed in higher levels of the hierarchy. In this way, they will get better response while using synchronous operations, will provide higher speed of propagation while using Fast_Write and will read more up-to-date data while using Fast_Read.

This is a complex problem since it depends on many factors, some of them are contradictory. Algorithms studying possible solutions need to be devised and evaluated. This topic is not addressed here as it is beyond the scope of the dissertation. In Chapter 6, different physical and logical topologies are evaluated just to illustrate their impact on performance and to give broad guidelines on the construction of the hierarchy. In the rest of the dissertation, it is assumed that an *external service* exists that is responsible for determining which nodes are grouped into which clusters as well as deciding which changes are to be made to the hierarchy to cope with the dynamic characteristics of the network. The output of this service can be fed to the reconfiguration protocols presented in Chapter 4 to build and restructure the hierarchy.

A similar service is studied by Katia Obraczka as her Ph.D. topic [Obraczka 95]. She proposes extending the TSAE protocol by organising replicas into hierarchical replication groups imitating the Internet hierarchy and replicas select their anti-entropy session peers from their logical neighbours. The architecture proposes a *physical topology estimator* that probes the underlying physical network and estimates its characteristics. Based on the estimated physical topology, the architecture builds automatically *hierarchical logical update topologies* that are k-connected for resilience, minimise the communication costs, have a limited diameter and reduce duplicate updates. The solutions rely on heuristic methods based on graph theory. Changes in the physical topology (such as node or link failures, network partitions, link congestion and so on) are detected periodically and a new logical topology is recomputed accordingly. Experiments are carried out to validate and evaluate the physical estimator for 12 Internet sites. Also, the logical calculator is evaluated and tuned over several physical topologies for different objective functions.

The idea of adopting a logical hierarchical structure has been suggested recently in [Bowman 94], where problems facing scalable resource discovery in the Internet have been discussed. It has been pointed out that a major problem is that existing replication protocols ignore network topology and that it is necessary to manage replication in groups that exploit the hierarchical structure of the Internet. The same idea has been used recently in [Dine 94, Renesse 95] for scalability reasons but for addressing a different problem, namely, multicasting and broadcasting in internetworks.

3.8 Summary

This chapter has presented a new protocol for managing massively replicated data based on a logical hierarchy. It provides an efficient propagation scheme where each node has to communicate with only a few other nodes, typically those in its physical proximity, thus reducing communications overhead, improving latency and making the scheme scalable. Further, it ensures reliable delivery with no redundancy.

A new service interface is proposed that provides different levels of asynchrony allowing strong and weak consistency to be integrated into the same framework and gives the application the ability to select between them. Further, the service provides the facility to offer different levels of staleness, depending upon the needs of various applications, by querying from different levels of the hierarchy. Finally, it allows the application to select from a number of reconciliation methods based on delivery order mechanisms to resolve conflicts. Therefore, the protocol provides a choice of service guarantees and gives the application developer the flexibility to tailor the service to achieve the required degree of consistency by choosing the appropriate operations to manage the data. It is then up to the application developer to present the user with a simple interface, while hiding from the user the details of the HARP's service interface.

Chapter 4

Restructuring the Hierarchy

4.1 Introduction

Restructuring operations provide mechanisms for a node to, dynamically, join or leave a cluster, to move from one cluster to another, to create or destroy clusters, to merge or split clusters and so on. These operations are necessary to build the tree, expand it, reorganise it to cope with performance problems such as node crashes, link congestion, partitions, and so on. The decision by which a node joins or leaves a cluster is not of our concern. But the manner in which the protocol deals with membership information of a node as well as ensuring that messages are not lost as a result of a reorganisation are relevant.

HARP provides four basic operations to reorganise the logical hierarchy, namely for a node to join, leave, initiate or destroy a cluster, while ensuring that no messages are lost and that the data structures are updated correctly to reflect the change [Adly 93a]. The operations are completely distributed, with no centralised components. Further, they do not rely on synchronisation with all nodes, nor on two- or three-phase commit protocols to perform the change. Instead, they rely on a weak consistency approach, that allows temporary inconsistencies to develop in the view of each node but guarantees that all replicas eventually converge to a single consistent view when the system stabilises and changes cease. This is achieved while incurring a low communication overhead and ensuring no loss of information.

The basic idea behind the protocol, is that a change in one cluster does not affect the operation of other clusters and that the $View$ structure -which reflects a node's view of the hierarchy- can be considered as a replica that does not always need to be seen consistently by all nodes. For instance, if node i needs to make a change (e.g. leave or 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, i.e. the parent and the members of the cluster that i is leaving or joining. Each node j involved in the change alters its $View$ to discard or include i and sends an acknowledgement to i. When these nodes have acknowledged the change, the operation is completed. Nodes in other clusters need not know about the change immediately since they are not communicating directly with i. Therefore, i can propagate its updated row $View_i.i$, 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.

The problem of joining and leaving a set of nodes has been addressed in the literature as *communication group membership* for multicasting and broadcasting. Most of these protocols provide *consistent* views of group membership; that is *all* nodes observe changes to the membership in the same order. For instance, group membership in Isis [Birman 87, Birman 91] as well as in [Cristian 91] are built on top of atomic broadcast protocols imposing total ordering. In [Ricciardi 91], two- and three-phase commit protocols are used to maintain consistent group views. The Arjuna system [Little 90] maintains a logically centralised group view via atomic transactions. These protocols suffer from high latency and communication overhead. Further, they suppress sending of new messages during a significant portion of the protocol.

In [Golding 92c], weak consistency membership protocols are presented that rely on chains of anti-entropy sessions to inform all nodes that a replica is joining or leaving the set of replicas. However, since sessions are performed periodically, this approach suffers from a long delay until nodes are informed about a certain change, including nodes in the local neighbourhood. Further, the leave operation is completed only when the leave request has been observed by *all* nodes, which results in a significant delay. Meanwhile, the leaving node cannot send any new messages.

Our protocol adopts a weak consistency approach and relies on the logical hierarchy to limit the reconfiguration overhead by confining the changes to the nodes grouped in the same cluster. Hence, it reduces the network traffic and the general overhead associated with reconfiguration. In Section 4.2, the four *basic operations* will be described. Their proofs of correctness are presented in Appendix A. Section 4.3 shows how these operations can be used to form *composite operations* and to reconfigure the hierarchy. Methods for handling failures and partitions are discussed in Section 4.4. Section 4.5 presents optimisations that reduce the overhead of some of the composite operations. Finally, optimisations concerning space and storage are discussed in Section 4.6.

4.2 Basic Operations

HARP provides four basic operations that allow the hierarchy to be reconfigured:

- leave: a node leaves its present cluster; keeping all its descendants,
- join: a node joins a cluster; keeping all its descendants,
- initiate: creates a new empty cluster, and
- destroy: destroys an existing cluster.

Figure 4.1 shows a part of the hierarchy if node d leaves C_{11} then joins again, and vice versa.

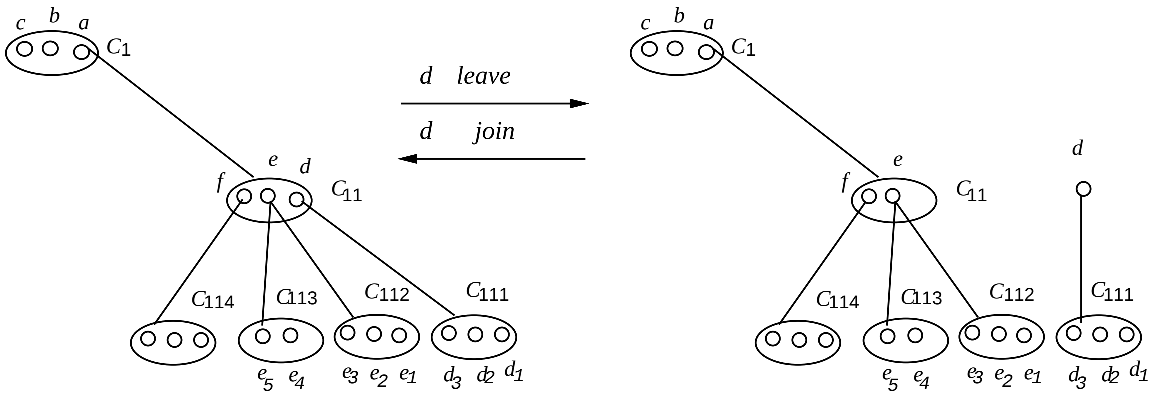

Figure 4.1: The hierarchical structure when node d leaves C_{11} then joins, or vice versa

Each node i keeps two variables:

- $status_i$ describing the status of node i, which is set to
  - normal while i is not performing any restructuring operations
  - leaving while i is performing a leave operation
  - joining while i is performing a join operation
- old_cl_i denoting the cluster id that i is leaving. It is set to $null$ while i is not performing a leave operation.

The following definitions will be used along this chapter:

- A node p is the *parent* of cluster C_x if $C_x \in View_p.p.Child\_Cids$
- A node i is a *current* member of C_x if $View_i.i.Cid = C_x$ and $status_i =$ normal
- A node i is a *leaving* member of C_x if $old\_cl_i = C_x$ and $status_i =$ leaving
- A node i is a *joining* member of C_x if $View_i.i.Cid = C_x$ and $status_i =$ joining

In general, when a node j changes its row in $View_j$ (because it has performed one of the restructuring operations), it propagates an *update_view* message containing its updated row $View_j.j$. Since messages can arrive to node i from node j in different orders, j increments $View_j.j.r\_seq$ before propagating the *update_view* and this will enable the latest update from j to be kept. That is, when node i receives *update_view* containing $View_j.j$, then i checks: if $View_i.j.r\_seq > View_j.j.r\_seq$, then it discards the message as it is an old update. Otherwise, it replaces $View_i.j$ by the new row as received from the message.

Node i needs to keep some of the following (temporary) lists while it is performing a restructuring operation:

- $join\_list_i$ contains the set of nodes i has to join with.

$$join\_list_i = \text{list of } (j, flag_1, flag_2), \qquad \text{where}$$

  - j is the node to join with,
  - $flag_1$ is a flag set to 1 if j acknowledged the change; 0 otherwise, and
  - $flag_2$ is a flag set to 1 if i has received from j the messages i asked for; 0 otherwise.

- $leave\_list_i$ contains the set of nodes i is leaving. They are kept in the list until they acknowledge that i is no longer in their $View$.

$$leave\_list_i = \text{list of } (j, flag), \qquad \text{where}$$

  - j is the node i is leaving, and
  - $flag$ is a flag set to 1 if j acknowledged the change; 0 otherwise.

- msg_list is a list of *uid*s of the messages i has asked other nodes for.

4.2.1 The LEAVE Operation

A node i leaves its cluster by executing a leave operation. The node keeps all its descendants when leaving its cluster. The leaving node changes its status to leaving, stops sending messages to its neighbours and parent -by discarding them from $View_i.i$-, waits until they have received and acknowledged every message sent by i previously, then, propagates an *update_view* message containing its updated row and declaring that it is leaving. Each correspondent j of node i receiving *update_view* from i, proceeds to propagate the message and updates $View_j.i$ as received from the message. Further, the parent and every neighbour j, sends i an *ack_leave* message. If j is leaving the cluster as well then it discards i from $leave\_list_j$. When i gets an *ack_leave* from the parent and all neighbours, then the operation is considered complete.

By the end of the operation, it is ensured that all members and the parent of the cluster have received every message i had before leaving. Further, they will not receive any messages from i after it leaves and vice versa. If while the operation is in progress, i receives any message for propagation from j such that $j \in leave\_list_i$, i sends an acknowledgement to j and it may either process the message and send it to its children or just reject it. This situation can occur if j sends a message to i before receiving the *update_view*. The leave protocol is described in Figure 4.2. The function Get_View(), shown in Figure 4.3, returns a list of the members and the parent of a certain cluster.

When a node leaves the top cluster, the quorum size of Slow_Write, Slow_Read and Opt_Write decreases dynamically. That is, when a node in the top cluster receives an *update_view* from a neighbour declaring it is leaving, it decreases the number of nodes of the cluster by one and computes the new quorum size. If a node while leaving receives a vote request, it does not send its vote in order to ensure quorum intersection.

```
When a node i decides to leave its cluster C_x:
{
  status_i = leaving;
  old_cl_i = C_x;
  leave_list_i = Get_View(i, C_x) - {i};          /* insert neighbours and parent in leave_list_i */
  View_i.i.Cid = null; View_i.i.P = null;      /* stop sending messages to neighbours and parent */
  ∀j ∈ leave_list_i
    wait until all messages sent to j have been acknowledged;
  /* propagate an update_view message */
  increment View_i.i.r_seq;
  propagate update_view(leaving, old_cl_i, View_i.i);          (see Figure 3.2)
  ∀j ∈ leave_list_i
    send to j update_view(leaving, old_cl_i, View_i.i) for propagation;
  when ack_leave is received from j
    If j ∈ leave_list_i Then set flag in leave_list_i to 1;
  when flag = 1,   ∀j ∈ leave_list_i {
    discard leave_list_i;
    old_cl_i = null;
    status_i = normal;
  }
}

A node j receiving update_view(stat, C_x, row_i) from i
{
  proceed in propagating the update_view message;          (see Figure 3.2)
  If View_j.i.r_seq > row_i.r_seq Then /* old update */
    discard the message and stop;
  replace View_j.i by row_i;
  If stat = leaving Then {           /* i is leaving C_x */
    If (View_j.j.Cid = C_x ∧ status_j = normal) ∨C_x ∈ View_j.j.Child_Cids Then
      /* j is a current member or the parent of C_x */
      send ack_leave to i;
    If old_cl_j = C_x ∧ status_j = leaving Then {
      /* j is a leaving member of C_x */
      If i ∈ leave_list_j Then remove i from leave_list_j;
      send ack_leave to i;
    }
  }
}
```

Figure 4.2: The leave protocol

Function Get_View(i, C_x)
{
/* returns a list of members of cluster C_x and its parent as seen by node i */
  $Cl\_view = \phi$;
  $\forall k \in N$
    If $(View_i.k.Cid = C_x \vee C_x \in View_i.k.Child\_Cids) \wedge View_i.k.up = 1$ **Then**
      $Cl\_view = Cl\_view \cup \{k\}$;
  If $View_i.i.Cid = C_x \wedge$ $status_i$ = joining **Then**
    /* add members of C_x as seen by i while joining */
    $Cl\_view = Cl\_view \cup \{k\}, \quad \forall k \in join\_list_i$;
  `return`(Cl_view);
}

Figure 4.3: The function Get_View()

4.2.2 The JOIN Operation

A node i joins a cluster C_x by contacting all members of C_x and the parent of C_x to inform them of its desire to join. Further, it exchanges with them missing messages to ensure that they have the same set of messages before normal propagation begins. The joining node keeps all its descendants when joining a cluster. For node i to join C_x, it performs join(C_x, q), where q is either the parent of C_x or any member of C_x. Node q is referred to as the *connect node* and is responsible for supplying i with a view of cluster C_x, i.e. a list of members and the parent of C_x. Apart from the first node to join the hierarchy where $q = null$, the following conditions should hold:

(Cond 1): C_x should have a *parent* or at least one *current* member while i is joining

(Cond 2): q should be either the *parent* of C_x or a *current* or a *leaving* member of C_x. If q is a *joining* member and is joining through the *connect* node r, then r should be either the *parent* of C_x or a *current* or a *leaving* member of C_x. This condition works recursively if r is a *joining* member.

The join protocol is described in Figures 4.4 and 4.5. The joining node changes its status to joining, and clears its previous view of C_x in $View_i$ so that it does not start sending messages to members of C_x before they observe i as a member. Then, i inserts q in $join\_list_i$ and sends q a $join\_req$. Node q replies by sending i its view of C_x (members and parent) and i updates $join\_list_i$ to include additional members and sends them $join\_req$. The nodes reply by sending i their view of C_x, i merges their views with $join\_list_i$ and sends additional members, if any, a $join\_req$. This process occurs iteratively until i sends $join\_req$ to all members of C_x.

```
Node i executing join(C_x, q)
{
  status_i = joining;
  ∀j ∈ N
    If View_i.j.Cid = C_x Then View_i.j.Cid = null;      /* clear previous view of C_x */
  /* initialise View_i.i */
  View_i.i.Cid = C_x;     View_i.i.P = null;     View_i.i.up = 1;
  If q = null Then                    /* i is the only member */
    change status_i to normal and stop;
    Else {
      msg_list = φ;      increment View_i.i.r_seq;
      insert q in join_list_i and send q join_req(C_x, View_i.i);
    }
  when node i receives a reply to the join_req from a node j {
    If ack_join(VV_j, LL_j, row_j, View_cl) is received Then {
      ∀k ∈ View_cl                /* update join_list_i */
        If k ∉ join_list_i ∧ View_i.k.Cid ≠ C_x ∧ View_i.i.P ≠ k Then
          add k to join_list_i and send k a join_req(C_x, View_i.i);
      /* update View_i */
      If row_j.r_seq ≥ View_i.j.r_seq Then replace View_i.j by row_j;
      If C_x ∈ View_i.j.Child_Cids Then View_i.i.P = j;
      Exchange(i, j);               /* exchange missing messages with j */
      set flag_1 for node j in join_list_i to 1;
      }
      Else {           /* a nak(View_cl) is received from node j; j has already left C_x */
        ∀k ∈ View_cl                /* update join_list_i */
          If k ∉ join_list_i ∧ View_i.k.Cid ≠ C_x ∧ View_i.i.P ≠ k Then
            add k to join_list_i and send k a join_req(C_x, View_i.i);
        /* i should ensure to receive messages that j had before leaving */
        choose any other node k ∈ join_list_i with flag_1 =1;
        reset flag_1 and flag_2 for k in join_list_i to 0;
        send join_req(C_x, View_i.i) to k;
        remove j from join_list_i;
        }
    }
  when flag_1 =1,      ∀j ∈ join_list_i
    /* all members have included i as a member */
    increment View_i.i.r_seq and propagate update_view(joining, C_x, View_i.i);
  when i receives extracted messages from j {
    set flag_2 for j in join_list_i to 1;
    If View_i.j.Cid = null Then send j an ack_leave;      /* j is a leaving member */
    }
  when flag_2 = 1,     ∀j ∈ join_list_i
    discard join_list_i;     msg_list = φ;     status_i = normal;
}
```

Figure 4.4: The join protocol

Node j receiving a $join\_req(C_x, row_i)$ from i

```
{
  If View_j.j.Cid = C_x ∨ old_cl_j = C_x ∨ C_x ∈ View_j.j.Child_Cids Then {
    /* j is a current, joining or leaving member of C_x or is the parent of C_x */
    If View_j.i.r_seq ≤ row_i.r_seq Then
      replace View_j.i by row_i;          /* update row View_j.i */
    If View_j.j.Cid = null ∧ old_cl_j = C_x Then         /* j is leaving C_x */
      insert i in leave_list_j;          /* wait until i extracts missing messages from j */
    send i ack_join(VV_j, LL_j, View_j.j, Get_View(j, C_x));
    }
    Else
      send i a nak(Get_View(j, C_x));              /* j has already left C_x */
}
```

Figure 4.5: The join protocol (Continued)

Node j, a member or the parent of C_x, receiving $join\_req$ accepts i's membership by including i in $View_j$ as a neighbour or as a child and sends i *ack_join* including its state VV_j and LL_j. When i receives *ack_join* from j, it includes j in $View_i$ as a neighbour or as a parent and it exchanges missing messages with j by sending to j what i has and j has not, and by asking j for messages that i is missing. Missing messages are determined by comparing VV_j and LL_j with VV_i and LL_i and they are sent in one batch.

When i receives *ack_join* from the parent and every member of C_x, then it propagates an *update_view* message, including its new row $View_i.i$, informing every node that it became a member of C_x. By exchanging missing messages with every member j, it is ensured that i has received all messages other members have and that every member has all messages that i has. When i has done so with every node in $join\_list_i$ then the operation is over.

If a node j receives $join\_req$ when it has already left C_x then it sends i a negative acknowledgement (nak). In this case, node i re-exchanges missing messages with any other *current* member to ensure that it receives all messages j had before leaving. If, while joining, i receives messages from a node k such that $k \in join\_list_i$ but i does not see k yet as a neighbour or a parent in $View_i$, then i should keep these messages in a list to be processed when k replies with *ack_join*. This can happen if k receives $join\_req$ from i, replies with *ack_join* and starts propagating messages to i, however, i receives these messages before receiving the *ack_join*.

It should be noted that i does not have to wait for *ack_join* from all members to start propagating messages. When i receives *ack_join* from a node, then i can start sending it messages immediately. However, if i is in the top cluster, it does not start assembling quorums for Slow_Write, Slow_Read and Opt_Write until all members have sent *ack_join* so

Function Exchange(i, j)
{

/* node i exchanges missing messages with node j */
  $msg\_req = 0$;
  If $View_i.j.Cid = C_x \vee View_i.i.P = j$ **Then** {
    /* j is a *current* or *joining* member of C_x or is the *parent* of C_x */
    /* i sends j what j is missing */
    $\forall k \in N$, $\forall$ *seq* s.t. $(seq \leq VV_i[k] \vee seq \in LL_i[k]) \wedge (seq > VV_j[k] \wedge seq \notin LL_j[k])$
        `retrieve the message with` $uid = \langle k, seq \rangle$ `from` Log_i `and send it to` j;
    /* ask j for what i is missing */
    $\forall k \in N$, $\forall$ *seq* s.t. $(seq \leq VV_j[k] \vee seq \in LL_j[k]) \wedge (seq > VV_i[k] \wedge seq \notin LL_i[k])$
        If $\langle k, seq \rangle \notin msg\_list$ **Then** {
          `extract the message with` $uid = \langle k, seq \rangle$ `from` j;
          `add` $\langle k, seq \rangle$ `to` msg_list;
          $msg\_req = 1$;
          }
    }
    Else {   /* j is a *leaving* member of C_x */
      /* ask j for what i is missing */
      $\forall k \in N$, $\forall$ *seq* s.t. $(seq \leq VV_j[k] \vee seq \in LL_j[k]) \wedge (seq > VV_i[k] \wedge seq \notin LL_i[k])$
          If $\langle k, seq \rangle \notin msg\_list$ **Then** {
            `extract the message with` $uid = \langle k, seq \rangle$ `from` j;
            `add` $\langle k, seq \rangle$ `to` msg_list;
            $msg\_req = 1$;
            }
      }
  If $msg\_req = 0$ **Then** {   /* i did not ask j for any messages */
    `set` $flag_2$ `for node` j `in` $join\_list_i$ `to 1`;
    If $View_i.j.Cid = null$ **Then**   /* j is a *leaving* member */
      `send` j `an` ack_leave;
  }
}

Figure 4.6: The function Exchange()

that it can compute the quorum size correctly. When a node in the top cluster receives a $join\_req$ from a node joining the cluster, it increases the number of nodes of the cluster by one and computes the new quorum size. If a node while joining receives a vote request, it sends its vote only if all members have sent ack_join ($flag_1$ is set $\forall j \in join\_list_i$) in order to ensure quorum intersection.

4.2.3 The INITIATE Operation

The initiate operation is performed by a parent to create an empty cluster as a child. A node i executes the operation initiate(C_x) by adding the cluster C_x to its sets of children. That is, it sets $View_i.i.Child\_Cids = View_i.i.Child\_Cids \cup \{C_x\}$. Then, it propagates an *update_view* message containing its new row $View_i.i$.

4.2.4 The DESTROY Operation

The destroy operation is performed by a parent to remove a cluster. The cluster should be empty for a deletion to be permissible. Assume node i executes the operation destroy(C_x), then it checks:

If $\exists\ j$ s.t.$View_i.j.Cid = C_x \wedge View_i.j.up = 1$ **Then**
    error /* cluster cannot be removed, it is not empty */
    Else {
        $View_i.i.Child\_Cids = View_i.i.Child\_Cids - \{C_x\}$;
        `increment` $View_i.i.r\_seq$;
        `propagate` $update\_view(status_i, C_x, View_i.i)$;
    }

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. For correctness, a sequence of join and leave operations is executed at a certain node serially. That is, a node has to be in the normal state before starting to execute any of the join or leave operations. If it is in a leaving or joining state, then it queues any new operations until it becomes normal. The correctness of the basic operations is established in Section A.1.

4.3 Composite Operations

The four basic operations allow the hierarchy to be built and reconfigured. For instance, a possible sequence of operations that different nodes would execute to build part of the subtree shown in the right-hand side of Figure 4.7 is as follows:

(1) a: join(C_1,-)	(2) a: initiate(C_{11})	(3) d: join(C_{11}, a)
(4) e: join(C_{11}, a)	(5) d: initiate(C_{111})	(6) d_1: join(C_{111}, d)
(7) d_2: join(C_{111}, d)	(8) d_3: join(C_{111}, d)	etc

Further, any reorganisation in the logical hierarchy can be done as a sequence of these operations. In the following, several examples of composite operations that allow the hierarchy to be reorganised are described. Each operation consists of a sequence of the basic operations.

move a node i may need to move from cluster C_x to another cluster C_y due to performance reasons e.g. if the links between i and nodes of C_x became congested while the communication between i and nodes of C_y 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 can be done as a sequence of leave and join operations. For example, for node f to move from C_{11} to C_{12}, as shown in Figure 4.7, it performs the following operations:

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

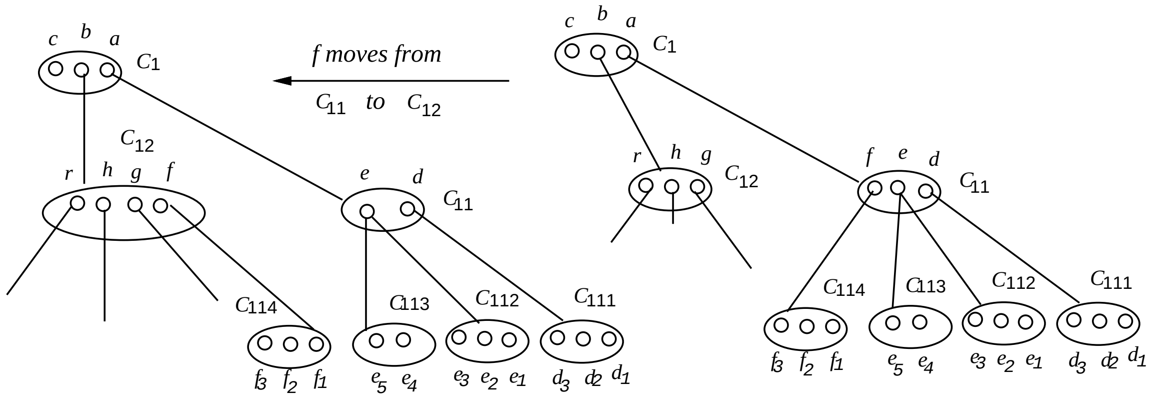

Figure 4.7: A node f moves from C_{11} to C_{12}

join-new a new node joins the set of replicas by a simple join operation. If it is joining through the *connect* node k, then k sends the new node a copy of the database, the message log Log_k, the view of the hierarchy $View_k$, as well as VV_k and LL_k.

change-parent changing the parent of a cluster is needed if the parent needs to shutdown or if the communication between the parent and children becomes congested. Assume C_{112} needs to change its parent from e to d, as shown in Figure 4.8. 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)

shutdown assume node f needs to shutdown, say for repair, then f's children change their parent from f to another node, say e, by performing change-parent. Then, f performs a simple leave operation.

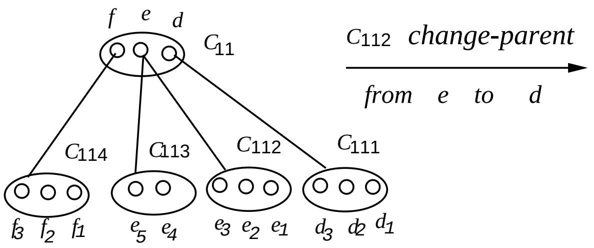

Figure 4.8: C_{112} changes parent from e to d

merge it merges two (or more) clusters into one. For example, to merge C_{113} into C_{114}, as shown in Figure 4.9, 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) or join(C_{114}, f_2) or join(C_{114}, f_3))
- e: destroy(C_{113})

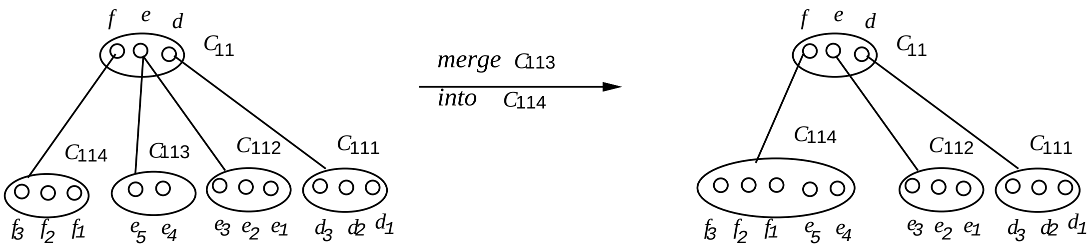

Figure 4.9: C_{113} merged into C_{114}

split it splits one cluster into two. For example, if it is required to split C_{112} such that e_1 and e_2 form another cluster C^*_{112} with d as a parent, as shown in Figure 4.10, 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)

Therefore, we can see that any operation needed to restructure the tree can be composed as a sequence of the basic operations, while ensuring correctness. That is, no messages are lost due to the change and the data structures are updated correctly to reflect the change. However, it involves some excessive overhead which could be optimised as will be discussed in Section 4.5.

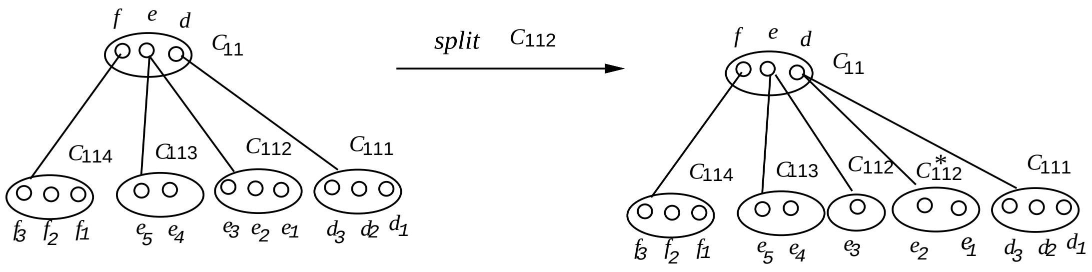

Figure 4.10: Split C_{112} to C_{112} and C^*_{112}

4.4 Failures and Partitions

While the protocol is operational, nodes may fail and communication links may fail and cause the network to be partitioned. When site or link failures occur, HARP continues to propagate information between connected operational 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.

4.4.1 Failures

It is assumed that a standard failure detection mechanism is available that decides when a particular node has failed. In the special case where a node loses its communication with all other nodes in the network, it is treated like a node failure. When a node f fails and a node i is informed that f has failed, i removes f from its view by resetting $View_i.f.up$ to 0 in order to mark f as down. Therefore, i will stop sending messages to f and will ignore any subsequent messages sent from f. This is necessary to avoid a situation where f starts to send messages without performing a recovery protocol. If $f \in join\_list_i$ or $leave\_list_i$, then i deletes f from the list so that i does not block indefinitely waiting for replies from f. Further, if f is in the top cluster, then nodes in the top cluster decrease the number of nodes in the cluster by one and compute the new quorum size accordingly.

The failure of f causes the hierarchy to be partitioned, where the children of the failed node and their descendants get isolated from the rest of the tree. Further, it might block the propagation of some messages. For instance, if f originates a message m, sends it to only a subset of its correspondents then dies, then m will not propagate everywhere. Similarly, if f receives a message m from say a neighbour or the parent but it fails before sending m to all its children, then all descendants of f will not receive m. Therefore, the protocol should guarantee that, despite failures, the following is true:

1. All running nodes can continue exchanging messages. In order to ensure flow of propagation while f is down, children of the failed node should be attached to a new parent

so that they are linked to the tree.

2. Any message that f failed to successfully send to all its correspondents before failing, and at least one of its correspondents has it, will be propagated everywhere.

These two conditions are guaranteed by the following steps, taken when the failure is detected. Assume node f, shown in the right-hand side subtree of Figure 4.7, fails and one of its neighbours, say e, will be the new parent for f's children. This node is elected by f's correspondents once they agree that f has failed. It is assumed that a standard election algorithm is run (see [Garcia-Molina 82]).

1. e initiate a new child cluster C_x. (e: initiate(C_x))

2. Children of f join C_x (f_1, f_2, f_3: join(C_x, e)). This will ensure that children of f will be linked to the tree and will receive from e any messages they have missed. Further, if a child f_1 of f has a message that another child f_2 of f does not have, then f_2 will receive the message. Finally, if a child f_1 of f has a message that neighbours or the parent of f do not have, then e will receive it and send it to other neighbours and the parent and consequently it will propagate everywhere.

3. Neighbours and parent of f exchange missing messages with each other. This will ensure that if one or more of f's neighbours or the parent have got a message then all f's correspondents will receive it and it will be propagated everywhere. This step can be achieved by having neighbours of f *rejoining* f's cluster. (e, d: join(C_{11}, a))

It is assumed that the instructions for performing these operations are given to the correspondents of f by the elected node. This method can be optimised, reducing some overhead, as will be described in Section 4.5.2.

When a node recovers after failures, there are generally two tasks that must be accomplished before normal processing can proceed. First, the operational nodes must be notified that the failed node needs to be reintegrated and update their $View$ consequently. Second, the state of the failed node must be restored. This includes the Log, VV, LL, $View$ and so on. These two tasks can be achieved by having the recovered node performing a simple join operation, to be reintegrated into the network. Hence, from Theorem 2, it will receive any messages it has missed during the failure, will propagate any message it had and did not send before it failed, and everybody will be informed of its recovery as the node will propagate an *update_view* message.

Note that in practice, it is not always possible to determine with absolute certainty that a particular node has failed; it may be slow or the network may be congested. Therefore, it is possible that the failure detection mechanism decides that a node f has failed while it has not. Further, when f comes up after a failure, its correspondents may be in the process of isolating it, or even it may have not yet been detected that f has failed. The recovery procedure copes with these situations since performing a join operation allows f to be reintegrated to the hierarchy, regardless of its previous position and of how other nodes were viewing f before it joins.

4.4.2 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 maintaining link status, and detecting when partitions occur and are restored.

When a partition occurs, the *affected nodes* are those which have been isolated from some of their correspondents due to the partitioning. *Affected nodes* in each partition mark in their *View* their correspondents in the other partition as being isolated in order not to attempt sending them messages. Further, *affected nodes* in one partition exchange missing messages with each other (i.e. with correspondents in the same partition). This ensures that if a message m has reached at least one node in a partition then every other node in this partition will receive m. 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 4.11 . When partitions are detected, nodes in P1 (i.e. a, f) remove e and d from their $View$, by resetting their *up* field to 0, hence stopping sending them messages and delete them from their *join_list* and *leave_list*. Further, a and f exchange missing messages between them. Similarly, nodes in P2 (i.e. e, d) remove a and f from their $View$ and exchange missing messages between them. After healing, nodes e and d 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.

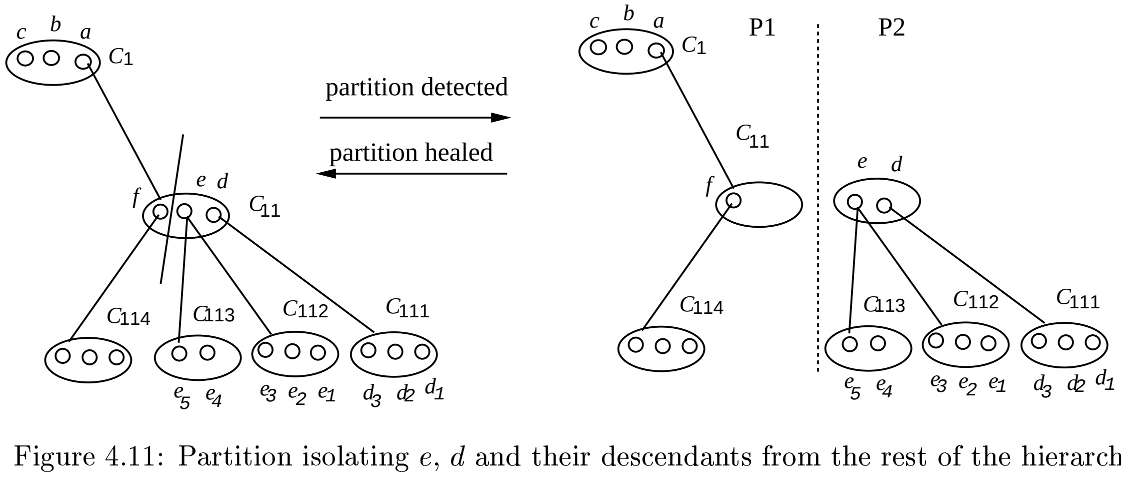

Figure 4.11: Partition isolating e, d and their descendants from the rest of the hierarchy

During partitioning, Fast_Write 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_Write 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). Opt_Write 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* 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 [Davcev 89, Jajodia 90] can be applied to the nodes of the top cluster.

4.5 Optimisation of Composite Operations

The composite restructuring operations are correct since they only consist of a sequence of the four basic operations. However, some optimisations can be achieved by writing each composite operation as an independent operation to reduce the overhead. For instance, change-parent could be optimised by having the children negotiating with the old and new parent only to update the *View* and exchange missing messages rather than leaving then joining the same cluster. Another example is merging a cluster C_x into C_y. The nodes of the merging cluster C_x 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_y only.

Similarly, the method for handling failures as described in Section 4.4.1, although correct as it includes only basic operations, it could be optimised by having a single node responsible for adopting the children of the failed node as well as exchanging missing messages with the neighbours, parent and children of the failed node. This would save the neighbours and the children from the overhead of rejoining the cluster and exchanging messages with each other.

In this section, two operations are selected and described in details, namely, change-parent, changing the parent of a node and take-over, for a node to take-over the responsibilities of a node that failed. They were selected to be studied because it is believed that they might be used more frequently than other operations.

4.5.1 The CHANGE-PARENT Operation

The change-parent operation is initiated by the old parent of a cluster and performed by the new parent. Assume a parent *old_p* needs to change the parent of its child cluster C_x

from node old_p to node new_p. The basic idea of the protocol is for old_p to inform new_p, which then contacts all members of C_x and exchanges with them any missing messages to ensure that they have the same set of messages before propagation begins.

The protocol is presented in Figures 4.13 and 4.14. An illustration of a scenario of message exchanges where old_p has two children i and j is shown in Figure 4.12. The protocol starts by old_p sending a request ($change\_req$) to new_p asking it to be the parent of C_x. The request includes a list of members of C_x. The new parent new_p replies to old_p by an ack_req message and sends a $change\_par$ message to members of C_x. A child receiving a $change\_par$ message, changes its parent from old_p to new_p in its $View$, sends new_p an ack_par including its state (VV and LL) and a list of its neighbours, then it propagates its new row in $View$. When new_p receives ack_par from a child, it updates its view of C_x and sends any additional member a $change\_par$ message. Also, it exchanges missing messages with the child to ensure that they have the same set of messages, then starts sending it new messages.

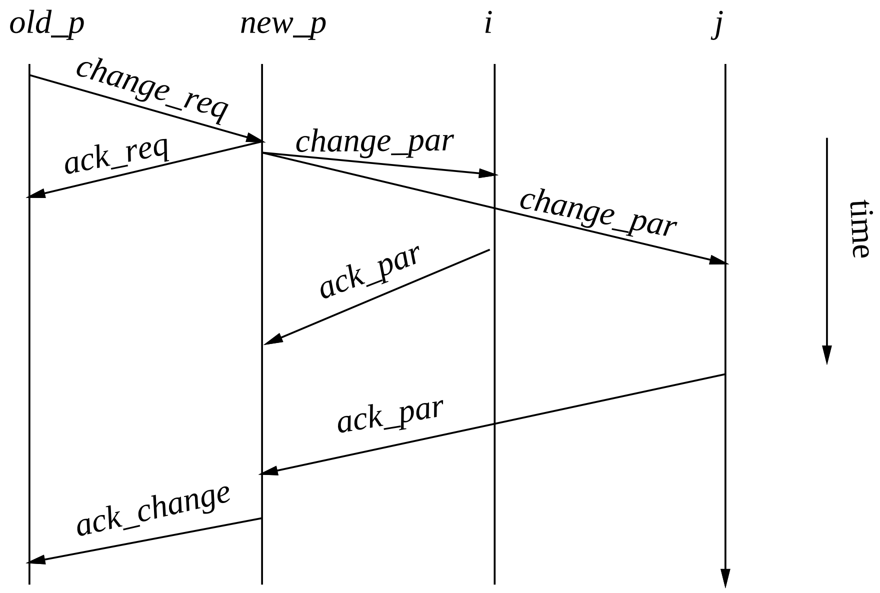

Figure 4.12: Scenario of message exchanges when old_p changes the parent of its two children i and j to the new parent new_p

When new_p has received an ack_par from every member of C_x, then it sends an ack_change to the old_p to inform it that all members have been notified. Then, it propagates an $update\_view$ message including its new row $View_{new\_p}.new\_p$ informing everybody that new_p is the new parent of C_x. When old_p receives an ack_change from new_p, then it propagates its row $View_{old\_p}.old\_p$ to inform everybody that it is not the parent of C_x anymore. When new_p has exchanged missing messages with all members of C_x then the operation is completed.

For correctness, new_p should not be in a joining or leaving status in order to be able to

The old parent old_p

{

  /* stop sending to members of C_x */

  $View_{old\_p}.old\_p.Child\_Cids = View_{old\_p}.old\_p.Child\_Cids - \{C_x\}$;

  $Children = \phi$;

  $\forall\ i \in N$

    If $View_{old\_p}.i.Cid = C_x \wedge View_{old\_p}.i.up = 1$ **Then** $Children = Children \cup \{i\}$;

  `increment` $View_{old\_p}.old\_p.r\_seq$;

  `send to` new_p $change\_req(C_x, Children, View_{old\_p}.old\_p)$;

  `when` old_p `receives` $ack\_req(row_{new\_p})$ `from` new_p

    /* update $View_{old\_p}.new\_p$ to denote that new_p is the parent of C_x */

    If $row_{new\_p}.r\_seq \geq View_{old\_p}.new\_p.r\_seq$ **Then** `replace` $View_{old\_p}.new\_p$ `by` row_{new_p};

  `when` ack_change `is received from` new_p

    /* inform everybody that old_p is not the parent of C_x anymore */

    `propagate` $update\_view(status_{old\_p}, View_{old\_p}.old\_p.Cid, View_{old\_p}.old\_p)$;

}

Node j **receiving** $change\_par(C_x, row_{old\_p}, row_{new\_p})$ **from** new_p

{

  If $row_{old\_p}.r\_seq \geq View_j.old\_p.r\_seq$ **Then** `replace` $View_j.old\_p$ `by` row_{old_p};

  If $row_{new\_p}.r\_seq \geq View_j.new\_p.r\_seq$ **Then** `replace` $View_j.new\_p$ `by` row_{new_p};

  $Neigh = \phi$;

  $\forall k \in N$

    If $View_j.k.Cid = C_x \wedge View_j.k.up = 1$ **Then** $Neigh = Neigh \cup \{k\}$;

  Case {

    $status_j =$ normal $\wedge View_j.j.Cid = C_x$

      /* j is a *current* member */

      $View_j.j.P = new\_p$;   `increment` $View_j.j.r\_seq$;

      `send` new_p $ack\_par(VV_j, LL_j, View_j.j, Neigh)$;

      `propagate` $update\_view(status_j, C_x, View_j.j)$;

    $status_j =$ joining $\wedge View_j.j.Cid = C_x$

      /* j is joining C_x */

      $View_j.j.P = new\_p$;   `increment` $View_j.j.r\_seq$;

      `send` new_p $ack\_par(VV_j, LL_j, View_j.j, Neigh)$;

      `remove` old_p `from` $join\_list_j$;

    $status_j =$ leaving $\wedge\ old\_cl_j = C_x$

      /* j is leaving C_x */

      `replace` old_p `by` new_p `in` $leave\_list_j$ `with` $flag = 0$;

      `send` new_p $ack\_par(VV_j, LL_j, View_j.j, Neigh)$;

    $View_j.j.P = new\_p$

      /* j has joined C_x and new_p is already the parent */

      `send` new_p ack_p;

    `Otherwise`   /* j has already left C_x */

      `send` new_p `a` nak;

    }

}

Figure 4.13: The change-parent protocol

The new parent new_p

{

when new_p receives $change\_req(C_x, Children, row_{old\_p})$ from old_p {

$status_{new\_p}$ = adopting;

If $row_{old\_p}.r\_seq \geq View_{new\_p}.old\_p.r\_seq$ **Then** replace $View_{new\_p}.old\_p$ by row_{old_p};

$View_{new\_p}.new\_p.Child\_Cids = View_{new\_p}.new\_p.Child\_Cids \cup \{C_x\}$;

increment $View_{new\_p}.new\_p.r\_seq$;

send old_p $ack\_req(View_{new\_p}.new\_p)$;

$join\_list_{new\_p} = Children$;

$\forall j \in join\_list_{new\_p}$

send j $change\_par(C_x, View_{new\_p}.old\_p, View_{new\_p}.new\_p)$;

}

when new_p receives reply from node j {

Case {

$ack\_par(VV_j, LL_j, row_j, Neigh)$ is received

If $row_j.r\_seq \geq View_{new\_p}.j.r\_seq$ **Then** replace $View_{new\_p}.j$ by row_j;

/* update $join\_list_{new\_p}$ from list of neighbours $Neigh$ */

$\forall k \in Neigh$

If $k \notin join\_list_{new\_p}$ **Then** {

add k to $join\_list_{new\_p}$;

send k $change\_par(C_x, View_{new\_p}.old\_p, View_{new\_p}.new\_p)$;

}

Exchange(new_p, j); /* exchange missing messages with j */

set $flag_1$ for node j in $join\_list_{new\_p}$ to 1;

ack_p is received /* j is already a child */

set $flag_1$ and $flag_2$ for j in $join\_list_{new\_p}$ to 1;

nak is received

/* j has already left C_x; new_p ensures receiving messages that j had before leaving */

choose any other node $k \in join\_list_{new\_p}$ with $flag_1 = 1$;

reset $flag_1$ and $flag_2$ for k in $join\_list_{new\_p}$ to 0;

send $change\_par(C_x, View_{new\_p}.old\_p, View_{new\_p}.new\_p)$ to k;

remove j from $join\_list_{new\_p}$;

}

}

when $flag_1 = 1$, $\forall j \in join\_list_{new\_p}$ { /* all members have changed their parent to new_p */

send ack_change to old_p;

/* inform everybody that new_p is the new parent of C_x */

propagate $update\_view(status_{new\_p}, View_{new\_p}.new\_p.Cid, View_{new\_p}.new\_p)$;

}

when new_p receives extracted messages from j {

set $flag_2$ for j in $join\_list_{new\_p}$ to 1;

If $View_{new\_p}.j.Cid = null$ **Then** send j an ack_leave; /* j is a leaving member */

}

when $flag_2 = 1$, $\forall j \in join\_list_{new\_p}$

discard $join\_list_{new\_p}$; $msg\_list = \phi$; $status_{new\_p}$ = normal;

}

Figure 4.14: The change-parent protocol (Continued)

perform change-parent. If it is, then it should queue the *change_req* until it becomes normal. However, *old_p* can be in any status since it is just initiating the change-parent operation rather than performing it. As shown, the algorithm takes care of the cases where the children of *old_p* are *current*, *joining* or *leaving* members of the cluster while the operation is in progress. The correctness of the change-parent operation is established in Section A.2 showing that no messages are lost and that the *View* is kept consistent.

A special situation can occur if a node i is joining a cluster C_x while C_x's parent has initiated change-parent operation but has not received an *ack_req* from the new parent yet. In this case, at the end of the join operation, the node i might end up having no parent. Therefore, an extra check needs to be added at the end of the join algorithm, described in Section 4.2.2; that is, if $View_i.i.P = null$, then i should enquire about the current parent of C_x by polling a neighbour for its view of C_x. Then it sends the parent a *join_req*.

The above protocol incurs less overhead than changing the parent of a cluster using the four basic operations as was explained in Section 4.3, as it saves the members of the cluster from the overhead of leaving the cluster then rejoining it again and exchanging missing messages with each other. Instead, each member of the cluster negotiates with the new parent only to update the *View* and to exchange missing messages, hence, incurring less overhead and reducing message traffic while ensuring correctness.

4.5.2 The TAKE-OVER Operation

When a node f fails, then one of its current neighbours or children will perform the take-over operation to take f's responsibilities. As mentioned before, this node is elected by f's correspondents after failure detection. If f was joining a new cluster when failed, then one of the new neighbours should take-over. The node must be in normal status before starting to take-over.

As discussed in Section 4.4.1, the failure of f causes its children to be isolated from the rest of the hierarchy. Further, it might block the propagation of some messages. Therefore, if a node i will take-over f, then it should perform two functions:

1. Adopts f's children, if any. This ensures flow of operation while the node is down.
2. Ensures that if f sent a message to, or received a message from, one of its correspondents then failed before propagating the message to *all* its other correspondents, then such a message will be propagated everywhere.

These two functions are achieved by performing change-parent; that is, i sends *change_par* requests to all f's children. So, they will assign i as the new parent, and if a child of f has a message that i does not have then it will be sent to i and vice versa. Further, i joins f's cluster. Hence, it takes exactly f's position and if a neighbour or the parent of f has a message that i does not have then it will be sent to i and vice versa. Node i receiving a missing message, sends it to *all* f's neighbours, children and parent. This avoids a situation where f sends a message m to a subset of the correspondents then fails and ensures that if any correspondent of f has m, then all correspondents of f will receive m, and consequently m will be propagated to every node in the network.

In the case where f failed while it was leaving its cluster; if i has not yet received *update_view* from f declaring that it is leaving, then f might not have completed the leave operation before failing. Therefore, i joins f's cluster and adopts f's children as usual. If i has received the *update_view* message from f, then it has completed the operation before failing. If i is a neighbour, then it just adopts f's children and there is no need to rejoin f's cluster, as f has sent every message to all neighbours before leaving. If i is a child of f, then it adopts f's children and joins *any* cluster to link with the rest of the hierarchy.

The take-over operation incurs less overhead than the method described in Section 4.4.1 using the four basic operations, as it saves the children of the failing node from the overhead of joining a new cluster and exchanging messages between each other. Also, the neighbours are exempted from rejoining f's cluster and exchanging messages with each other. This reduces message traffic and communication overhead.

The steps taken by node i while performing take-over are outlined in Figure 4.15. The rest of the scenario of join and change-parent is the same as was described before, except when i receives any extracted message, it sends it to *all* its neighbours, children and the parent that appear in $View_i$.

If f failed while it was adopting the children of a cluster C_x (i.e. while performing change-parent), then the old parent j of C_x, would not have received an *ack_change* from f. In this case, j asks members of C_x to change their parent back to j (i.e. send them *change_par*), and any message m that j extracts from a child should be sent to all neighbours, children and parent to ensure that it is propagated everywhere. When j receives *ack_par* from all members of C_x then it can ask another node to be the new parent of C_x if desired.

The correctness of take-over is presented in Section A.3 showing that every child of f will be linked to the hierarchy, regardless of the status of f or its children, and that if f fails before sending a message m and at least one of its correspondents has m, then m will be propagated everywhere.

In the case where f failed while it was taking over another node j, it is assumed that the node i that will take-over f will take-over j as well. If j and f were neighbours, then i adopts the children of j and f and joins f's cluster as before. If j was the parent of f, then i adopts the children of j and f and joins j's cluster.

Node i **executing** take-over
{
$status_i$ = taking-over;
/* join f's cluster */
$clust = View_i.f.Cid$;
If $clust \neq null$ **Then** join($clust, k$);
s.t. $k \in N \wedge ((View_i.k.Cid = clust \vee clust \in View_i.k.Child\_Cids) \wedge View_i.k.up = 1)$
If $clust = null \wedge View_i.i.P = f$ **Then**
/* f fails while leaving, and i is a child */
join any cluster;
/* adopt f's children */
increment $View_i.i.r\_seq$;
$\forall C_x \in View_i.f.Child\_Cids$ {
$View_i.i.Child\_Cids = View_i.i.Child\_Cids \cup \{C_x\}$;
$\forall k \in N$ s.t. $View_i.k.Cid = C_x$ {
add k to $join\_list_i$;
send k $change\_par(C_x, View_i.f, View_i.i)$;
}
}
}

Figure 4.15: Steps taken by a node i taking over a node f

4.6 Space Optimisations

This section discusses some optimisations that allow reduction of the size of the acknowledgement matrix, *AckM*, the *Log* and the *View* structure, hence, saving storage. Another aspect of optimisation is to reduce the amount of status information transferred during reconfiguration. This issue will be addressed separately in Chapter 8.

4.6.1 The *AckM* and the *Log*

The fact that a node discards a message from the *Log* when it has been received by all nodes in the network, and that the size of *AckM* is $O(n^2)$ may cause excessive storage, especially for a large number of replicas. Two separate optimisations can be made:

1. Since a node i sends messages to nodes in its neighbourhood only (children, neighbours and parent), 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 neighbourhood have acknowledged the message. This optimisation will enable i

to flush Log_i more quickly, hence reducing its size. Consequently, the number of rows in $AckM_i$ can be reduced to include only i's correspondents instead of $n-1$ rows.

This optimisation may impose some restrictions on reconfiguring the hierarchy. Basically, a node i cannot move to other clusters freely as nodes of the new cluster might not have in their *Log* messages that i is missing. Similarly, a node after recovering, has to join the same cluster it was affiliated with before it failed as its members are the only ones that are keeping the messages the node has missed when it was down. For instance, node e_1 in Figure 3.1 cannot move and be a child of node u, or if e fails and recovers it cannot join C_{12} instead of C_{11}. Without this optimisation it is feasible to support such reconfigurations as u will have in its Log_u messages that e_1 has not received yet and will pass them to e_1 while joining. Hence, there is a tradeoff between the cost of maintaining additional information and flexibility in the reorganisation of the hierarchy.

However, it is believed that having a complete freedom for reconfiguring the hierarchy is unlikely to be needed in most cases, especially in large scale systems connected through internetworks. It seems more likely that a reorganisation will involve a rearrangement of the nodes within a local subtree of the hierarchy, spanning two, or may be three levels. Such reconfigurations would need a node i to keep rows in *AckM* for i's correspondents and their correspondents only, and consequently to flush a message from Log_i when the message has been received by those nodes. Maintaining such information would give more flexibility in the reconfiguration while the size of *AckM* and the *Log* is kept small by storing only local information rather than a complete state of all nodes.

2. If clocks are synchronised (i.e. global timestamps can be used) then the matrix *AckM* can be optimised to a vector *AckV* of dimension $1 \times n$, where, $AckV_i[j] = ts$ means that node i knows that j has received every message from any sender with timestamp $\leq ts$. In this case, when node i initiates a message, it assigns it a timestamp from the local clock. 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, which is the minimum entry in VV_k. When i receives such a message, it updates its entry k of its vector $AckV_i$. Then, node i can purge a message from the log if its timestamp is less than or equal to the minimum entry in $AckV_i$. This idea has been used by the propagation protocols presented in [Heddaya 89, Golding 92c]. Again optimisation 1 can apply here and the number of columns in *AckV* can be reduced.

4.6.2 The *View* Structure

Although the presentation of the propagation and the reconfiguration protocols considered that each node keeps the view of the whole tree, this is not necessary for the correctness of the operations. A node needs only to keep rows for itself and its correspondents and discard rows about the other nodes. This will reduce the size of the *View* structure while the propagation protocol, all the basic restructuring operations and the change-parent operation are still correct.

For handling failures, as described in Sections 4.4.1 and 4.5.2, more information needs to be kept in the $View$ of each node. For instance, for a node to take-over a neighbour or a parent, it needs to keep rows for their correspondents. Similarly, for a node to take-over a grand-parent, if consecutive failures occur, then it needs to keep rows of its correspondents and so on. Therefore, there is a tradeoff between the amount of information kept in the $View$ of a node and the number of successive failures in different levels of the hierarchy that the node should handle. Again, it is believed that a certain node is likely to take-over a node in its local neighbourhood only. Consequently, it would suffice for a node to keep in its $View$ rows for its correspondents and their correspondents.

It should be noticed that the decisions of how many rows to keep in $View$ and $AckM$ (and consequently when to flush a message from the Log) can be made dynamically and do not have to be statically predefined. Since each node i propagates an *update_view* message whenever a change in its row in $View_i$ occurs, a node j, receiving this message can then decide whether it needs to keep a row for i or not. Similarly, since each node i propagates its VV_i periodically, then j can decide whether it wants to keep a row for i in $AckM$ or not on receiving this message. Therefore, the degree of flexibility in reconfiguring the hierarchy and the levels of failures that a node should handle can be determined dynamically without the need to alter the protocols.

In fact, it is possible to devise a propagation algorithm based on a logical hierarchical structure where each node keeps only local state information. Not only can it keep a partial view of the hierarchy, but the state vectors recording messages a node has received could be confined to its local portion of the hierarchy rather than the whole network. These local state vectors are exchanged in case of reorganisation, failures or partitions, instead of exchanging VV and LL, which could be large for very large scale networks. Such a protocol would be more scalable than HARP since it reduces communication overhead. However, it imposes some limitations in handling failures due to the minimal information kept. Such a protocol is presented in detail in Chapter 8 and its strengths and limitations are discussed.

4.7 Summary

Dynamic restructuring operations were presented which allow the hierarchy to be built and reconfigured, including the restarting of failed nodes and re-merging of partitioned networks. They rely on a weak consistency approach where a node performs a change after negotiating with a small number of nodes. This might result in temporary inconsistency in the view of other nodes, but the protocol ensures that the views of all nodes eventually converge to a single consistent view when the changes cease.

The protocol provides four basic operations for a node to join or leave a cluster or to initiate or destroy a cluster. Methods for reconfiguring the hierarchy, for handling failures and recovery as well as handling partitions using these basic operations have been presented. Another two operations have been presented in detail, namely, changing the parent of a cluster and taking over another node that failed. All the operations are distributed and do not rely on two-phase commit protocols. So, they incur low message traffic, small communication overhead and they do not suppress normal operation and the sending of new messages while the protocols are running. This is achieved while ensuring that no messages are lost. The scheme is therefore suitable for large scale systems.

Chapter 5

Causal Order

One way to guarantee consistency of replicated data is to impose a delivery order on messages when reaching their destinations. One type of delivery ordering that is of interest is *causal ordering*. A new protocol is proposed that encodes causal ordering information efficiently with each message on an unreliable communication network [Adly 95a]. It is designed for a replicated system where a message is sent to every replica in the network. It is based on the hierarchical propagation scheme of HARP. The desirable characteristic of the protocol is that it imposes little space overhead in the required timestamp appended to each message. This low cost of timestamp size results in reduced communication overhead and increased performance and scalability of the system.

This chapter describes the proposed protocol. It begins by defining causal ordering, then it describes the notion of Vector Clocks which have been widely used to implement causality. Section 5.3 reviews some of the previous proposed solutions to maintain causality and Sections 5.4 and 5.5 present two suggested algorithms to provide causal ordering within HARP. In Sections 5.6 through 5.8, the restructuring operations and the procedures to handle failures and partitions presented in Chapter 4 are revisited to introduce the necessary modifications to allow the logical hierarchy to be reconfigured while causal ordering is preserved.

5.1 What is Causal Ordering ?

Causal ordering is a generalisation of FIFO ordering, using the *happened before* relationship introduced by Lamport [Lamport 78]. When processes in a network communicate, the messages they exchange define a partial order of events. Two messages m and m' are said to be causally related, noted $m \rightarrow m'$, if the event of sending m could have caused the event of sending m'; that is, the process that sent m' had either sent m or already received m before sending m'. Causal order is satisfied if m' is delivered after m at every process in the system.

Protocols implementing causal order distinguish between *received* messages and *delivered* messages such that, a received message is not delivered to the replica immediately, but its delivery is delayed until a certain condition is verified. Consider the events $send_i(m)$ and $delv_i(m)$ to denote the transmission of message m by node i and the delivery of message m to the replica at node i, respectively. As in [Lamport 78], the *happened before relation* for the system (noted $\rightarrow$) is defined as follows:

1. for any two events e_1 and e_2 produced by the same process:
   if e_1 is produced before e_2, then $e_1 \rightarrow e_2$
2. for any message sent from node i to node j: $send_i(m) \rightarrow delv_j(m)$
3. if $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ then $e_1 \rightarrow e_3$

Such a relation characterises any distributed computation based on multicast in an asynchronous environment. In order to be a *causal order*, this relation has to satisfy the following constraint:

If $send_i(m) \rightarrow send_j(m')$ then for any node k: $delv_k(m) \rightarrow delv_k(m')$

Causal consistency has many important applications. For instance, the networks news systems can use this ordering to ensure that users will see follow-up messages only after seeing the original posting. Other applications that can benefit from causal ordering are teleconferencing, trading systems, locating movable objects in distributed systems, distributed debugging and so on. Renesse in [Renesse 93] shows the usefulness of causal ordering in a distributed system by presenting some important examples of common distributed applications where causality is a necessary and sufficient ordering of events.

5.2 Vector Clocks

The system of *Vector Clocks* was developed independently in [Fidge 88] and [Mattern 89]. In this system, each node maintains a local variable called *Vector Clock*, denoted by VC, which is an integer vector of dimension n. The component $VC_i[i]$ reflects the logical time at node i (measured in *number of past events*), while $VC_i[j]$ is the best estimate that node i was able to derive about node j's current logical clock value $VC_j[j]$.

A Vector Clock VC_i at node i is maintained according to the following rules:

(1) Initially, VC_i initialised to 0: $VC_i[j] = 0, \quad \forall j$

(2) On each internal or send event, node i increments VC_i: $VC_i[i] = VC_i[i] + 1$

(3) On sending a message m, node i attaches a timestamp $VC(m)$ to m: $VC(m) = VC_i$

(4) On receiving a message m with timestamp $VC(m)$, node i updates VC_i:
$VC_i[j] = \mathsf{Max}\ \{VC_i[j], VC(m)[j]\}, \quad \forall j.$

The following definitions are associated with Vector Clocks:

$$VC_i = VC_j \Leftrightarrow VC_i[k] = VC_j[k], \quad \forall k$$

$$VC_i < VC_j \Leftrightarrow VC_i \neq VC_j \wedge VC_i[k] \leq VC_j[k], \qquad \forall k$$

From the construction of VC, we have:

$$e \rightarrow e' \Leftrightarrow VC(e) < VC(e') \qquad \text{(the proof can be found in [Schwarz 94])}$$

This property can be exploited to efficiently determine the causal relationship between events based on their Vector Clocks.

5.3 Related Work

A number of algorithms have been developed to provide causal message ordering.

Schiper et al describe a method based on Vector Clocks [Schiper 89]. Each process maintains a Vector Clock and a buffer containing the Vector Clocks associated with the most recent message sent to each process by this process, or known to be sent by any other process. The buffer is, in essence, the Vector Clocks associated with the set of undelivered messages, as seen by the current process. Every message carries a copy of the buffer as a timestamp. From this buffer, a process can determine if a message has any undelivered causal predecessors, and can delay delivery of the message appropriately. The size of message timestamps required by this method is of order n^2, where n is the number of processes.

Isis [Birman 87, Birman 91] developed protocols for maintaining causality in a general framework where there can be several groups of processes, possibly overlapping, and each message is multicast within a group. They are communication protocols, where external events only are considered, namely, send and receive a message. Originally in [Birman 87], the protocol consists of sending a sufficient set of predecessor messages (rather than just message ids) along with each message. In [Birman 91], the protocol was improved by piggybacking the message ids only. This protocol, based on the Vector Clocks and influenced by protocols developed by Ladin [Ladin 90], may be summarised as follows:

- Each node maintains a Vector Clock VC_i
- Node i increments $VC_i[i]$ before sending a message m and timestamps m with $VC(m) = VC_i$
- On receipt of a message m originating at node i and timestamped with $VC(m)$, node j delays delivery of m until two conditions are satisfied:
  (1) $VC_j[i] + 1 = VC(m)[i]$, and
  (2) $VC_j[k] \geq VC(m)[k] \qquad \forall k \neq i$

The first condition ensures that j has received all messages that i has originated before m while the second condition ensures that j has received all messages that i has already received from other nodes before originating m. Delayed messages are placed temporarily into a *holding queue* until the appropriate messages arrive. When a message is delivered, VC_j is updated to the element-wise maximum of VC_j and $VC(m)$. Figure 5.1 shows an example where node $N1$ sends a message m_1 to nodes $N2$ and $N3$. $N2$ sends a message

m_2 after receiving m_1. However, $N3$ receives m_2 before m_1. Since the second condition is not satisfied, then $N3$ delays the delivery of m_2 until m_1 arrives. The amount of information appended to each message requires a space proportional to the total number of processes in the network. Similar propositions have been designed by several authors [Golding 92c, Macedo 92, Raynal 91] based on Vector Clocks.

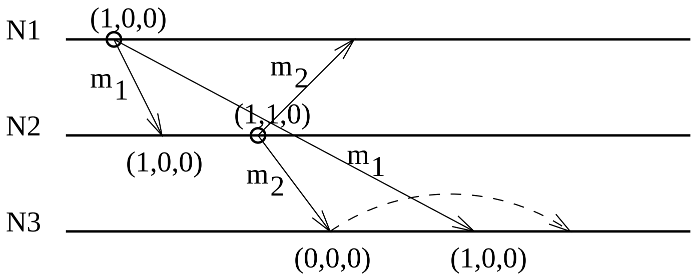

Figure 5.1: An illustration of Isis method to maintain causality

Psynch [Peterson 89, Mishra 89] and Transis [Amir 92] preserve causality by maintaining a message dependency graph rather than passing Vector Clocks. In Psynch, the causal relation is represented in the form of a directed acyclic graph, called the *context graph*. The nodes in this graph represent the multicast messages, while the edges represent the causality relation between the receipt of one message by a process and the subsequent sending of another message. When a process sends a message, it multicasts the message to each remote host and attaches to the message causal ordering information consisting of the edges that connect the message to the context graph. This information represents message ids already seen by the sending process. When a process receives a message m, if one or more of m's predecessor messages have not arrived, then m is placed temporarily into a *holding queue* until the appropriate messages arrive. The amount of information appended to each message requires a space proportional to the total number of processes in the network.

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. A number of optimisations have been suggested to reduce the size of timestamps required to record causality:

- A compression technique is proposed in [Singhal 92] that sends only entries in the Vector Clock that changed since the last transmission. This technique can reduce the size of the timestamps only if process interaction exhibits temporal or spatial localities, and in the worst case the size is of order n. Compression also requires additional storage which has a size of order n^2. Further, it requires that communication channels be FIFO.

- Fowler and Zwaenepoel [Fowler 90] have introduced methods that achieve a reduced size timestamp by encoding only direct causal dependencies. A Vector Clock for an event, that represents transitive dependencies on other processes, is constructed off-line from a recursive search of the direct dependency information at processes. However, these schemes require significant computation to calculate transitive dependencies, and the dependencies can only be calculated with access to the event streams of all processes. The methods require an order of h steps of computation, where h denotes the total number of messages sent during the computation. Therefore, this technique is not suitable for applications that require on-line computation of the Vector Clock.

Not all suggested solutions are cited here. Schwarz and Mattern in [Schwarz 94] present a good survey of the published causal ordering protocols and their relative strengths and weakness are discussed in detail. Also several methods that capture causality and based on logical clocks are reviewed in [Raynal 95].

5.4 Causal Order in HARP with Version Vectors

Since the propagation protocol involves only communication events (i.e. message send and receive) and no internal events, then the *Version Vectors* can be used as Vector Clocks and an algorithm such as the one proposed in [Birman 91] can be adopted in HARP to capture causality. It should be noticed that the model is designed for a replicated system where every message is destined for all nodes. The modifications to the propagation protocol presented in Section 3.3 would be as follows:

- Each node i originating a message m timestamps m with $VV(m) = VV_i$
- When a node j receives a message m for causal delivery, it compares VV_j with $VV(m)$ and delays delivery of m until

$$VV_j[i] + 1 = VV(m)[i] \text{ and } VV_j[k] \geq VV(m)[k] \qquad \forall k \neq i$$

The remainder of the propagation protocol is the same and the restructuring operations presented in Chapter 4 do not require any modifications to ensure correctness. Note that this method requires VV_i to be sent with every message propagated.

5.5 Causal Order in HARP with Compact Vectors

It has been shown that causal order can be implemented in HARP using VV at the expense of appending a timestamp of size $O(n)$ to every message. However, it is believed that a timestamp of size $O(n)$ imposes a high communication overhead, especially for a large number of nodes. In this section, a novel algorithm is presented to maintain causality which reduces the size of the timestamp significantly. The algorithm relies on the hierarchical propagation scheme described in Section 3.3 and is based on Vector Clocks.

The basic idea behind the algorithm is to take advantage of the hierarchical structure and of the fixed pattern of propagation that messages follow. Consider any node i belonging to cluster C_x. Following the propagation algorithm, node i receives every message -originating at any node in the network- either from a neighbour, the parent or a child. Then, to preserve causality, before delivering a message m, a node needs to check that it has received from its correspondents and delivered every message that preceded, and therefore was causally related to m. Therefore, each node needs to keep track of messages received from neighbours, parent and children, and stamp messages with this information only. The algorithm presented here cuts down the size of the timestamp appended to each message to be of order q, where q is the number of nodes in a cluster. In Section 5.5.1, Compact Vectors are introduced, which are similar to Vector Clocks in that they reflect what messages have been sent to and received from other nodes. However, the knowledge is confined to nodes in a certain cluster rather than relating to every node in the network. Section 5.5.2 presents the causal delivery condition using the Compact Vectors as well as the procedure to update these vectors.

5.5.1 Compact Vectors (CV)

Each node keeps a *Compact Vector (CV)* [1] for each cluster it communicates with directly recording messages sent and received from the cluster. Assume that node i is the r^{th} member of the cluster C_x and that i has c child clusters (ordered from 1 to c). Further, assume that each cluster contains q nodes, ordered by global node identifier. Then, node i maintains $c+1$ vectors $(CV_i^0, CV_i^1, \ldots, CV_i^c)$ and each vector is of size $q+1$: [2]

• The CV_i^0 vector summarises what messages were sent to and received from neighbours and parent. More specifically,

- $CV_i^0[0]$ = number of the last message *received* by i from the parent and delivered at i
- $CV_i^0[r]$ = number of the last message delivered at i and *sent* to neighbours and parent
- $CV_i^0[k]$ = number of the last message *received* by i from member k of C_x and delivered at i, $\forall k \neq r$

• The CV_i^y vector summarises what messages were sent to and received from children belonging to cluster C_y. More specifically,

- $CV_i^y[0]$ = number of the last message delivered at i and *sent* to children belonging to C_y
- $CV_i^y[k]$ = number of the last message *received* by i from child k belonging to C_y and delivered at i

[1] As discussed in Section 3.3.1, vector notation is adopted for simplicity of exposition, and for consistency with the literature. Again, a functional notation would have described the structures more precisely, since each entry in the Compact Vector needs to be associated with the corresponding node identifier. Also, the size of these structures changes when reconfiguration occurs. In this case, CV would be defined as $CV : N \to seq$, where seq is a set of integer sequence numbers. However, in order to keep the terminology of the literature, we refer to the data structures as Compact Vectors. A likely implementation is by means of a linked list of pairs (node id, sequence) which, after the first entry for the parent node, is ordered by global node identifier.

[2] The numbers c and q are used for ease of presentation and are not a restriction of the algorithm.

5.5.2 The Causal Delivery Condition

Assume node i receives a message m from correspondent j. Then, there are three cases:

Case 1 if j is a neighbour

Then, i needs to delay the delivery of m until it has received and delivered all messages j had delivered before m. These messages are either sent directly from j or both i and j have received them from other members of C_x or the parent of C_x. Therefore, j stamps m with its CV_j^0 and i compares it with its own CV_i^0 and applies the causal delivery condition. That is, assuming that j is the k^{th} neighbour:

- j stamps m with $CV(m) = CV_j^0$, and
- i delays delivery of m until
  $CV(m)[k] = CV_i^0[k] + 1 \wedge CV(m)[l] \leq CV_i^0[l], \quad \forall l \neq k$

Case 2 if j is a child belonging to cluster C_y

Then, i needs to delay the delivery of m until it has received and delivered all messages j had delivered before m which are either sent directly from j or both i and j have received them from other members of C_y. So, j stamps m with its CV_j^0 (which carries information about cluster C_y) and i compares it with its own CV_i^y and applies the causal delivery condition. That is, assuming that j is the k^{th} child belonging to cluster C_y:

- j stamps m with $CV(m) = CV_j^0$, and
- i delays delivery of m until
  $CV(m)[k] = CV_i^y[k] + 1 \wedge CV(m)[l] \leq CV_i^y[l], \quad \forall l \neq k$

Case 3 if j is the parent

Then, i needs to delay the delivery of m until it has received and delivered all messages j had delivered before m which are either sent directly from j or both i and j have received them from other members of C_x. So, j stamps m with its CV_j^x and i compares it with its own CV_i^0 (which contains information about cluster C_x) and applies the causal delivery condition. That is:

- j stamps m with $CV(m) = CV_j^x$, and
- i delays delivery of m until
  $CV(m)[0] = CV_i^0[0] + 1 \wedge CV(m)[l] \leq CV_i^0[l], \quad \forall l \neq 0$

Delayed messages are placed in a queue, $undelv\_q_i$, to be processed after the missing messages arrive [3]. An undelivered message is marked with which correspondent it came from so

[3] It should be noticed that a node does not propagate a message to its correspondents until it has been delivered.

When node i generates a message m:

{

`increment` $CV_i^0[r]$;

`stamp a copy of` m `with` CV_i^0 `and send it to parent and neighbours;`

$\forall\ j = 1$ `to` c {

  `increment` $CV_i^j[0]$;

  `stamp a copy of` m `with` CV_i^j `and send it to children belonging to cluster` C_j;

  }

}

When node i receives a message m from its k^{th} neighbour:

{

/* Check for causality */

If $(CV(m)[k] = CV_i^0[k] + 1 \wedge CV(m)[l] \leq CV_i^0[l]\ \ \forall l \neq k)$ **Then** {

  `deliver` m;

  `set` $CV_i^0[l]$ = Max $\{CV_i^0[l], CV(m)[l]\ \}$, $\forall l$;

  $\forall\ j = 1$ `to` c {

    `increment` $CV_i^j[0]$;

    `stamp a copy of` m `with` CV_i^j `and send it to children belonging to cluster` C_j;

    }

  `process any messages in` $undelv\_q_i$ `that are now in order ;`

  }

  Else

    `Mark` m `as non-delivered and place it in` $undelv\_q_i$ `for later processing;`

}

When a node i receives a message m from the k^{th} child belonging to cluster C_y:

{

/* Check for causality */

If $(CV(m)[k] = CV_i^y[k] + 1 \wedge CV(m)[l] \leq CV_i^y[l],\ \forall l \neq k)$ **Then** {

  `deliver` m;

  `set` $CV_i^y[l]$ = Max $\{CV_i^y[l], CV(m)[l]\ \}$, $\forall l$;

  `increment` $CV_i^0[r]$;

  `stamp a copy of` m `with` CV_i^0 `and send it to parent and neighbours;`

  $\forall\ j = 1$ `to` $c,\ j \neq y$ {

    `increment` $CV_i^j[0]$;

    `stamp a copy of` m `with` CV_i^j `and send it to children belonging to cluster` C_j;

    }

  `process any messages in` $undelv\_q_i$ `that are now in order ;`

  }

  Else

    `Mark` m `as non-delivered and place it in` $undelv\_q_i$ `for later processing;`

}

Figure 5.2: Steps a node i takes to deliver a message m and to update the Compact Vectors, upon originating m or receiving m from its neighbours or children

When node i receives a message m from its parent:

```
{
 /* Check for causality */
If (CV(m)[0] = CV_i^0[0] + 1 ∧ CV(m)[l] ≤ CV_i^0[l]  ∀l ≠ 0) Then {
  deliver m;
  set CV_i^0[l] = Max {CV_i^0[l], CV(m)[l] }, ∀l;
  ∀ j = 1 to c {
    increment CV_i^j[0] ;
    stamp a copy of m with CV_i^j and send it to children belonging to cluster C_j;
    }
  process any messages in undelv_q_i that are now in order ;
  }
  Else
    Mark m as non-delivered and place it in undelv_q_i for later processing;
}
```

Figure 5.3: Steps a node i takes to deliver a message m and to update the Compact Vectors, upon receiving m from its parent

that when delivered, the node knows to which correspondents it should be sent to. The steps node i takes to deliver a message m and to update the Compact Vectors, upon originating m or receiving m are described in Figures 5.2 and 5.3.

It is obvious from the protocol that each message is stamped by one Compact Vector which is of size $O(q)$, where q is the number of members in a cluster. This reduces significantly the size of the message especially for a large number of nodes. This reduction in the size of the timestamp results in a lower communication overhead and enhances scalability of the system. Using the above protocol, it is guaranteed that causal order is preserved. The proof of correctness is established in Section B.1.

5.6 Modifications to the Restructuring Operations

Unlike Version Vectors, a Compact Vector cannot be freely compared to any other Compact Vector. Two Compact Vectors can be compared to each other only if both of them hold message numbers sent or received by the members and the parent of the same cluster.

Definition 1 *Two Compact Vectors, CV_1 and CV_2 are compatible if their entries hold message numbers sent or received by the same set of nodes.*

Only compatible Compact Vectors can be compared to each other. Therefore, if a node i receives a message m timestamped with a $CV(m)$ incompatible with i's Compact Vectors, then the check of causality for m cannot be carried out and m will never get delivered at i. During normal operation, a node i is ensured to receive all messages with timestamps compatible with CV_i.

When a reconfiguration occurs, since it involves nodes changing clusters and correspondents, the Compact Vectors of each node i participating in the change should be updated to reflect the new configuration by adding (removing) entries for new (old) correspondents. While updating them, it may happen that they become incompatible with the timestamp $CV(m)$ of an undelivered message m received by i before or during the reconfiguration. These situations should be handled so that such a message m is eventually delivered at i. Further, following the restructuring operations presented in Chapter 4, two new correspondents i and j exchange missing messages during the reconfiguration. If i and j were belonging to different clusters, then the Compact Vectors of i are incompatible with the timestamps associated with messages in Log_j and vice versa. Consequently, the messages received during the exchange will remain blocked indefinitely at both i and j.

Therefore, some modifications are needed to the restructuring operations. Mainly, the procedure by which messages are exchanged needs to be altered so that received messages can be delivered in causal order. Further, a procedure for updating the Compact Vectors needs to be added for each operation while ensuring that messages received before, during, and after the reorganisation are delivered in causal order and that no undelivered messages are blocked indefinitely.

5.6.1 The Basic Operations

The modifications to the basic operations require adding a flag to the $View$ structure so that a node can distinguish between the different states of other nodes. The flag $View_j.i.state$ is two bits long and is set to:

'00' if i is in the process of joining j's cluster. It is set after j receives *join_req* from i.

'01' if i has joined j's cluster. It is set after j receives *update_view* from i with $status_i =$joining and $View_i.i.Cid = View_j.j.Cid$.

'10' if i has left j's cluster. It is set after j receives *update_view* from i with $status_i =$leaving.

'11' if i belongs to another cluster. It is set after j receives *update_view* from i with $View_i.i.Cid \neq View_j.j.Cid$.

For the leave operation presented in Section 4.2.1, the modifications are simple. When a node j receives the *update_view* message from a neighbour or a child i declaring that it is leaving, it waits until all undelivered messages that have been received from i before the *update_view* have been delivered. Then, j discards the corresponding entry of i from its Compact Vectors. The waiting step ensures that if j had a message m received from i that is still undelivered, then j's Compact Vector will remain compatible with $CV(m)$ until m is delivered. Further, since a node does not leave except when it makes sure that all messages it had to send to its neighbours and parent have been received and acknowledged, then the neighbours and the parent of the leaving node will be able to deliver any message sent by the leaving node before it leaves as well as any message that is causally related to messages sent via the leaving node and no undelivered messages are blocked indefinitely.

A node i can initiate or destroy a cluster C_x as described in Sections 4.2.3 and 4.2.4 respectively. Also, it needs to create or destroy the corresponding Compact Vector CV_i^x.

The main modifications to the join operation presented in Section 4.2.2 are that nodes exchange only messages that have been delivered rather than exchanging all messages that have been received. Those messages are delivered at the recipient in the same order they were delivered at the sender. Since the sender has already delivered them in causal order, then the order is preserved at the recipient. Nodes need to stamp messages stored in the log with the order in which they were delivered to pass on this order with each message sent during the exchange. Further, the set of the missing messages are stamped with the corresponding Compact Vectors so that the recipient updates its Compact Vectors accordingly before normal propagation begins. The scenario of message exchange with the modifications is outlined in Figure 5.4 and an explanation follows.

- Assume node i is joining a cluster C_x. Node i starts by initialising its CV_i^0 to 0.
- When node i sends *join_req* to node j, a member or the parent of C_x, it associates the message with VV_i^u (rather than VV_i), which is equivalent to VV_i but omitting the uids of messages that are undelivered; that is, messages queued in $undelv\_q_i$ [4].
- When a node j, a member or the parent of C_x, receives *join_req* from i, it creates a new entry for i in its Compact Vector and initialises it to 0. Further, it compares VV_j^u with VV_i^u and sends i messages that j has delivered and i has not along with the *ack_join*. The *ack_join* message is also stamped with VV_j^u, and CV_j^0, if j is a member of C_x, or CV_j^x if j is the parent of C_x. Then, j can start propagating messages to i.
- On receiving *ack_join* from j, node i sends j messages that i has delivered and j has not stamped with CV_i^0. Further, it delivers the messages received from j in order, then it updates CV_i^0 as follows:

  If j is a neighbour then $CV_i^0[l] = \mathsf{Max}\ \{CV_i^0[l], CV_j^0[l]\},\ \forall l$

  If j is the parent, then $CV_i^0[l] = \mathsf{Max}\ \{CV_i^0[l], CV_j^x[l]\},\ \forall l$

  Therefore, CV_i^0 is updated to reflect messages that have been received and delivered from members and the parent of C_x and CV_i^0 becomes compatible with CV_j^0 of every neighbour j and with CV_p^x of the parent p. Consequently, i is ready to accept messages stamped from its new neighbours and parent and can start propagating messages to them.

[4] VV^u could be maintained at each node all the way through normal operation or it can be constructed from VV and *undelv_q* at the time the reorganisation takes place.

- A neighbour j, on receiving the missing messages from i, delivers them in order and it updates its CV_j^0 to the element-wise maximum of CV_i^0 and CV_j^0. This ensures that if i has sent any messages for propagation to other members or the parent of C_x after it has joined, these messages will be reflected in CV_j^0 [5]. Similarly, the parent p updates CV_p^x after delivering the received messages from i.

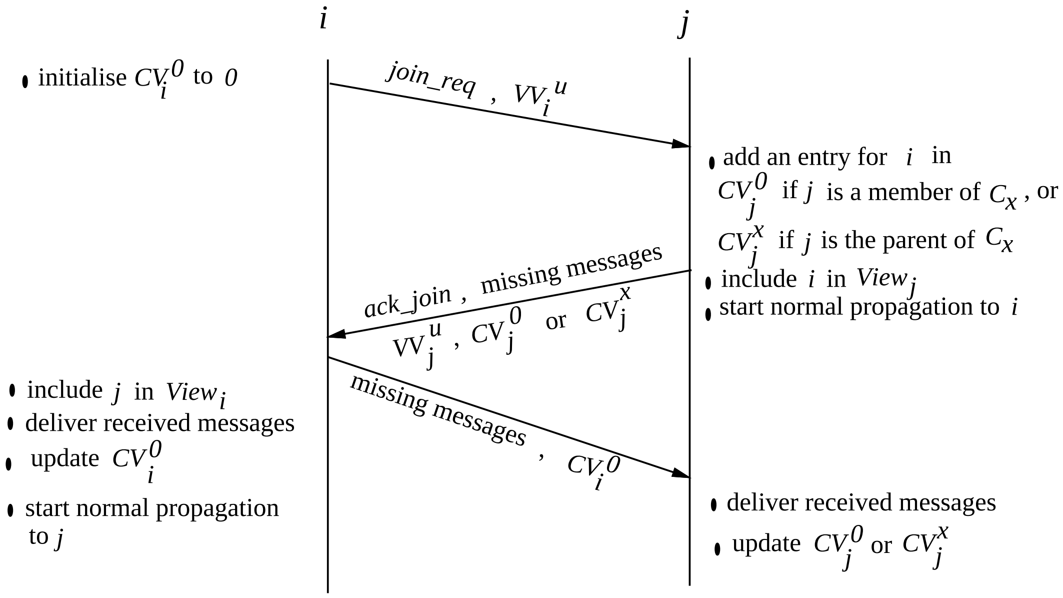

Figure 5.4: Scenario of message exchange between a node i joining a cluster C_x and node j a member or the parent of C_x

It should be noted that during or after the join operation, a node i might be receiving a message m from a correspondent j that has been delivered at j but m already exists in $undelv\_q_i$ undelivered. This can happen if m has been already received by i from another correspondent k. In this case, i delivers m as received from j and updates its CV accordingly. Further, if k is a current correspondent, then i updates its CV as if m has been received from k to ensure the flow of message delivery. If k is a previous correspondent then i just discards m.

Assume a node j, a member or the parent of C_x, receives a message m from another member k while a node i is joining C_x. The timestamp $CV(m)$ might or might not carry an entry for i, depending on whether i has or has not joined k yet. Consequently, the causality check cannot be performed if an entry for i exists in $CV(m)$ but not in CV_j^0 or vice versa, since they are incompatible and m will not be delivered at j. The same applies if node i is leaving C_x. These cases need to be handled to ensure that comparing CV_j^0 with $CV(m)$, m will not be blocked indefinitely and that it is delivered in causal order. The checks carried out

[5]Recall that i does not have to wait for *ack_join* from all members and the parent of C_x to start propagating messages. When i receives *ack_join* from a node, then i can start sending it messages immediately.

by j to detect these situations and the associated actions taken are described in Figure 5.5. In the figure, it is assumed that j is a neighbour of i, where i is joining or leaving C_x. The same modifications apply if j is the parent of C_x (substituting CV_j^0 by CV_j^x) or if j is a node joining C_x.

---

(1) **If** `an entry for` i `exists in` CV_j^0 `but not in` $CV(m)$ `and` $View_j.i.state =' 01'$ **Then**
  /* k has left i while j has not yet */
  j `places` m `in` $undelv\_q_j$ `until the entry of` i `is discarded from` CV_j^0`, then performs the causality check`

(2) **If** `an entry for` i `exists in` $CV(m)$ `but not in` CV_j^0 `and` $View_j.i.state =' 10'$ **Then**
  /* j has left i while k has not yet */
  j `discards the entry of` i `from` $CV(m)$ `before comparing` $CV(m)$ `and` CV_j^0

(3) **If** `an entry for` i `exists in` CV_j^0 `but not in` $CV(m)$ `and` $View_j.i.state =' 00'$ **Then**
  /* i has joined j but has not yet joined k */
  j `creates a entry for` i `in` $CV(m)$ `with value` 0 `before comparing` $CV(m)$ `and` CV_j^0

(4) **If** `an entry for` i `exists in` $CV(m)$ `but not in` CV_j^0 `and` $View_j.i.state =' 11'$ **Then**
  /* i has joined k but has not yet joined j */
  j `places` m `in` $undelv\_q_j$ `until` $join\_req$ `is received, then performs the causality check`

Figure 5.5: Checks and actions performed by node j on receiving a message m from node k, a neighbour or the parent of j, while a node i is joining or leaving j's cluster

Following the above modifications it is ensured that no messages are lost during the leave or join operations and that the Compact Vectors are updated such that all messages are delivered in causal order and no undelivered messages are blocked indefinitely. The proof of correctness is presented in Sections B.2 and B.3.

5.6.2 The CHANGE-PARENT Operation

Assume a node *old_p* needs to change the parenthood of its child cluster C_x to a new parent *new_p*. The change-parent operation, described in Section 4.5.1, needs some modifications which are very similar to those of the leave and join operations. Members of C_x discard *old_p* from their Compact Vectors only after delivering all messages sent by *old_p* before the change. Also, members of C_x and *new_p* exchange only delivered messages, and the recipient delivers the received messages in the same order they were delivered at the sender. Further, compatible Compact Vectors are exchanged and updated before normal propagation begins. The scenario of message exchange is outlined in Figure 5.6 and the modifications are as follows:

- *old_p* stops sending new messages to members of C_x, but it waits until all previously delivered messages that have been sent to members of C_x have been received and

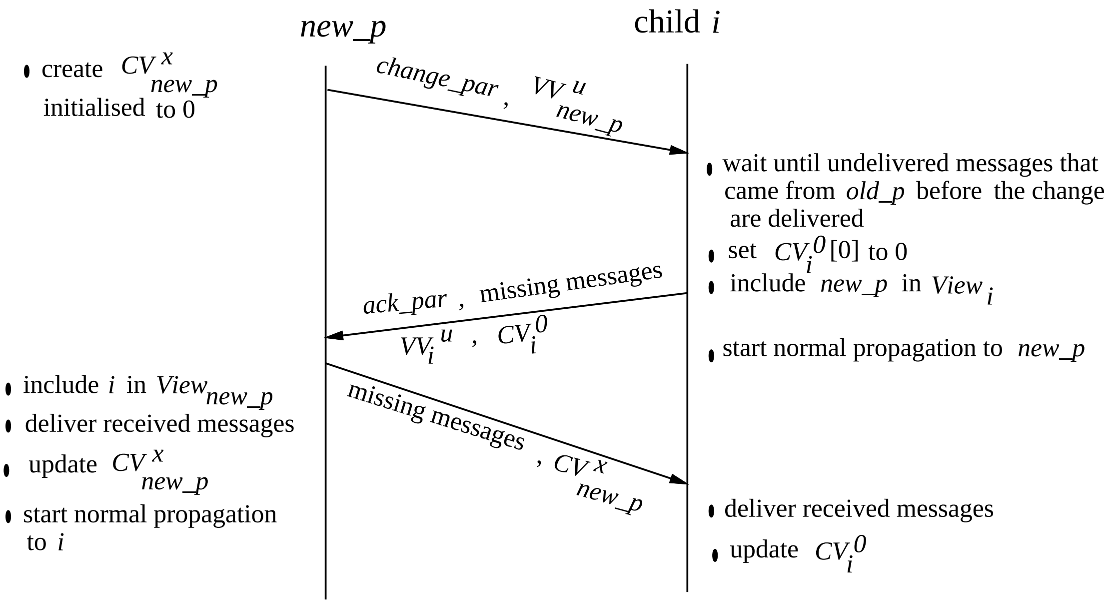

Figure 5.6: Scenario of message exchange while a node i is changing its parent to *new_p*

acknowledged before issuing *change_req*. This is to prevent undelivered messages at a child from being blocked indefinitely.

- When a node i, a member of C_x, receives *change_par* from *new_p*, it waits until all undelivered messages that have been received from *old_p* before receiving *change_par* have been delivered. Then, it changes the first entry of CV_i^0 from *old_p* to *new_p* and resets its value to 0 ($CV_i^0[0] = 0$), initialising the number of messages received from *new_p*. The waiting step is necessary to ensure that no message that previously came from a neighbour or *old_p* before the change and is causally related to messages received from *old_p* is blocked indefinitely. Further, i includes *new_p* in its $View_i$ and sends *new_p* the *ack_par* message associated with VV_i^u, the set of delivered messages that *new_p* does not have and CV_i^0.

- When *new_p* receives *ack_par* from a node i, it sends i the set of delivered messages that i does not have associated with $CV^x_{new\_p}$. Further, it delivers the messages received from i in order, then it updates $CV^x_{new\_p}$ to the element-wise maximum of $CV^x_{new\_p}$ and CV_i^0. Similarly, node i delivers the messages received from *new_p* in order and updates CV_i^0 to the element-wise maximum of $CV^x_{new\_p}$ and CV_i^0.

- When the operation is completed, if *old_p* still has an undelivered message m that came from a member of C_x, then *old_p* discards m; that is, it pretends it has not received it. Then, it will receive m and messages causally related to it through normal propagation when members of C_x propagate those messages through *new_p*.

Assume node i, a member of C_x, receives a message m from another member j while the change is taking place. The first entry of $CV(m)$ might carry the timestamp of *new_p* or *old_p* depending on whether j has received *change_par* and has reset $CV_j^0[0]$ to 0 or not. The checks needed to detect these situations and the actions taken to ensure that comparing CV_i^0 with $CV(m)$, m will be delivered in order are described in Figure 5.7.

---

(1) **If** the first entry in $CV(m)$ is for *new_p* while that of CV_i^0 is for *old_p* **Then**
  /* j changed its parent to *new_p* while i did not */
  i places m in $undelv\_q_i$ until *change_par* is received and $CV_i^0[0]$ set to 0, then performs the causality check
(2) **If** the first entry in $CV(m)$ is for *old_p* while that of CV_i^0 is for *new_p* **Then**
  /* i changed its parent to *new_p* while j did not */
  i sets $CV(m)[0]$ to 0 before comparing CV_i^0 and $CV(m)$

Figure 5.7: Checks and actions performed by node i, a member of C_x, on receiving a message m from another member j while they are changing their parent

The rest of the protocol is the same as described in Section 4.5.1. The protocol ensures that *new_p* and the members of C_x will receive the same set of messages delivered in causal order and that no undelivered messages will be blocked indefinitely. The proof of correctness is presented in Section B.4.

5.7 Failures

Either the procedure based on the basic operations (Section 4.4.1) or the take-over operation (Section 4.5.2) is used for handling failures. Since take-over is mainly a combination of change-parent and join operations performed by the node taking over the failed node, then the modifications described in Sections 5.6.1 and 5.6.2 are adopted. However, while the children of the failed node change their parent, they do not have to wait for undelivered messages that came from neighbours and the failed node to get delivered before participating in change-parent. The waiting step might block the take-over operation if the failed node has died before sending messages causally related to those messages. Since those messages are delivered at least at a neighbour, they will be received by the node taking over and will be sent to all f's correspondents.

Assume a node f has failed before sending or receiving a message m and that node i is taking over f, then the take-over operation guarantees that:

1. if at least one correspondent of f, say j, has delivered m, then j will send m to i -while i is performing join or change-parent- and i delivers it and sends it to all f's correspondents. Consequently, m will be received and delivered everywhere.

2. if j, a correspondent of f, has m as an undelivered message, then there are three cases:

   Case 1 if m came from a correspondent of j, say k, that is not a correspondent of f. Then $CV(m)$ does not depend on messages sent from f and m will be delivered at j. If m is delivered during the take-over, then j sends m to i which will deliver it and send it to all f's correspondents. If m is delivered at j after the take-over, then j sends m to its own correspondents including i. Since i takes exactly f's position, then m follows the same pattern of propagation it would have followed if f was alive and consequently will be propagated and delivered everywhere.

   Case 2 if m came from a correspondent of j, say k, that is a correspondent of f. Then k has delivered m and will send it to i during the take-over and i will send it to all f's correspondents. Consequently, m will be received and delivered by all nodes through propagation.

   Case 3 if m came from f itself. In this case there is no guarantee that m can be delivered at j. This case occurs if say f received a message m' from a child c_1, then it generates the message m (i.e $m' \to m$) but it sends m to only one neighbour j then it dies. Since the Compact Vectors of c_1 and j are not compatible then even if j receives m' from c_1 it will not be able to deliver it.

In other words, take-over ensures the flow of propagation while f is down. Further, it ensures that if f had failed before sending or receiving a message m and at least one correspondent of f has delivered m then m will be received and delivered everywhere. But if m is causally dependent on a message m' it guarantees that m and m' will be propagated and delivered everywhere provided that at least one of f's correspondents has received and delivered both m and m'. If such a correspondent does not exist then m may remain undelivered until f comes up. This restriction is due to the fact that not all correspondents of f have compatible Compact Vectors.

When the failure is repaired, the node is reintegrated through a join operation.

5.8 Partitions

The procedure described in Section 4.4.2 can be used to handle partitions. However, in order to ensure that if a message m has reached at least one node in a partition, then every node in this partition will *receive* and *deliver* m, a modification is needed in the procedure used by the *affected nodes* [6] in one partition to exchange missing messages with each other. The modification is based on the fact that delivered and undelivered messages should be treated separately to preserve the causal order. But since nodes exchanging messages are

[6] Recall that *affected nodes* are those which have been isolated from some of their correspondents due to the partitioning.

members or the parent of the same cluster, then their Compact Vectors are compatible and they can exchange undelivered messages.

Assume nodes i and j are two *affected nodes* in one partition. After exchanging VV and LL, assume node i asks j for a set of missing messages, then j should send to i the requested messages such that:

- For messages that were delivered at j, they are sent along with CV_j^0, if j is a neighbour or a child of i, and with CV_j^x, if j is the parent of i and i belongs to cluster C_x. Node i receiving those messages, processes them in the same order that they were delivered at j, hence, preserving causal order. Further, i updates its Compact Vectors to reflect the messages it has just delivered. Therefore,
  If j is a neighbour of i then, $CV_i^0[l] = \mathsf{Max}\ \{CV_i^0[l], CV_j^0[l]\}$, $\forall l$
  If j is the parent of i and i belongs to C_x then $CV_i^0[l] = \mathsf{Max}\ \{CV_i^0[l], CV_j^x[l]\}$, $\forall l$
  If j is a child of i and j belongs to C_y then $CV_i^y[l] = \mathsf{Max}\ \{CV_i^y[l], CV_j^0[l]\ \}$, $\forall l$

- For messages that have not been delivered yet at j, they are sent to i such that if i is a neighbour or the parent of j, then j sends only undelivered messages that came from other neighbours or the parent of j along with $CV(m)$ and the identity of the sender k. This information enables i to deliver the message when it is in the right order. Node i treats the message as if it is coming from node k. For other undelivered messages (sent to j from its children) they will be sent to i when they are delivered at j following the normal propagation algorithm.

  Similarly, if i is a child of j and i belongs to cluster C_y, then j should send i only undelivered messages that came to j from other children belonging to cluster C_y along with the identity of the sender and $CV(m)$. Other undelivered messages will be sent to i when they are delivered at j.

5.9 Summary

This chapter has presented an efficient protocol that supports exchange of messages among a set of nodes while preserving the causal ordering of message delivery in the presence of communication and processor failures. The protocol has the desirable characteristic that it imposes a low space overhead on the causal order information associated with each message. This has been achieved by using the hierarchical propagation algorithm of HARP, where each node sends and receives messages from a few nodes only. Consequently, a node needs to keep track of messages received from those nodes and stamp messages with this information only in order to verify the causal ordering. This reduction in the size of timestamp -as opposed to previous solutions that used a timestamp of size n- results in reduced communication overhead and makes the scheme suitable for large scale systems. The protocol is asynchronous; that is, it does not block the sender until remote delivery occurs, which is a key to high performance. Further, it makes very few assumptions about the underlying network.

Chapter 6

Performance Evaluation of HARP

6.1 Introduction

A simulation study was carried out to evaluate the performance of the proposed protocol HARP [Adly 95b, Adly 95d]. The aim of the study is to explore the following issues:

1. Synchronous operations are expected to impose limitations on performance such as response time and throughput, while asynchronous operations are expected to give better performance but suffering from accessing out-of-date information. It is desired to study the tradeoffs between performance and staleness of data, especially under different system configurations such as workload mixes, load intensity and communication overhead.
2. Reading from higher levels of the hierarchy is expected to give more up-to-date information in exchange of a higher response time. The performance of reading from different levels of the hierarchy needs to be investigated.
3. It is desired to evaluate the protocol on different hierarchical network topologies, and consequently different logical topologies, to see how they can affect performance.
4. Comparing the performance over a non-hierarchical topology versus a hierarchical one.
5. Exploring the effect of increasing the degree of replication on performance.

Previous performance studies of synchronous replication protocols were confined to evaluate the quorum size and the availability of data. However, the costs of maintaining consistency in terms of latency and communication overhead were not quantified. These costs are important to be evaluated, especially in internetworks environments. Therefore, the conducted study models several components which affect the evaluation of these metrics, such as the network delays, the overhead of running various algorithms, the quorum collection policy, handling deadlocks and so on.

This chapter presents the simulation model designed and implemented (Section 6.2). Then, the results are presented and analysed. Since a certain application cannot mix synchronous and asynchronous writes together while updating the same data item, two groups of experiments are carried out in order to evaluate the above points. The first group (Section 6.3) considers synchronous updates only (i.e. Slow_Write and Opt_Write) and evaluates the performance of mixing synchronous reads (i.e. Slow_Read) and asynchronous reads (i.e. Fast_Read) under different system configurations and network topologies. The second group (Section 6.4) performs a similar set of experiments when updates are asynchronous. Also, it compares the performance of asynchronous and synchronous updates. Section 6.5 outlines the results of a set of experiments performed with large number of replicas. The experiments presented here are a subset of the total experiments performed. The full set of the experiments and the analysis are reported in [Adly 95d].

6.2 The Simulation Model

This section presents the design of the simulation model. It describes the system and the network models, the overhead of the algorithms and the way the various operations are modelled. Then, it defines the performance metrics used for evaluation. The model is described by a set of parameters which are listed in Table 6.1.

The simulator was implemented using the simulation package Simpack [Fishwick 92] developed at the University of Florida and based on the C programming language.

6.2.1 The System Model

The model consists of a number nodes connected by a communication network. Each node is assumed to store a copy of the object being replicated. A node has nonvolatile storage (disk) and a CPU. CPUs are modelled as single queue with single server; with no preemption. Each disk has its own queue, which it serves in FIFO manner.

The system is modelled as an open system where requests arrive with arrival rate drawn from a Poisson distribution. Each request is composed of a single operation (read or write). Each operation is parametrised by *cpu_req* and *io_req* specifying the average amount of CPU and I/O time required respectively; they are exponentially distributed. The fraction of read requests is determined by the parameter $freq_r$. In this study, two Fast_Read operations are evaluated to reflect the effect of reading from different levels of the hierarchy. The first operation, denoted by Fast_Read_0, reads from the local node; and the second one, denoted by Fast_Read_1, reads from a node at one level up in the hierarchy, that is from the parent

parameter	Description	Default value
n	Number of nodes in the network	12
inter_arr_time	Inter arrival time	1000 msec
$freq_r$	Frequency of read requests	.85
$fast_{r0}$	Frequency of Fast_Read_0 requests	var [0-1]
$fast_{r1}$	Frequency of Fast_Read_1 requests	var [0-1]
$slow_r$	Frequency of Slow_Read requests	var [0-1]
$fast_w$	Frequency of Fast_Write requests	var [0-1]
$slow_w$	Frequency of Slow_Write requests	var [0-1]
opt_w	Frequency of Opt_Write requests	var [0-1]
cpu_req	Processing requirements of a request (exponential)	50 msec
io_req	I/O requirements of a request (exponential)	60 msec
cpu_msg	CPU overhead to send/receive a message	2.0 msec
lan_del	Network delay over LAN (uniform)	0.02 msec
wan_del	Network delay over WAN (exponential)	10.0 msec
prop_ovhd_const	Propagation overhead: constant part	2.0 msec
prop_ovhd_var	Propagation overhead: variable part (per destination)	1.0 msec
pc_ovhd_const	2PC overhead: constant part	2.0 msec
pc_ovhd_var	2PC overhead: variable part (per participant)	1.0 msec
cpu_lock	CPU time to acquire/release a lock	0.2 msec
deadlock_timeout	Timeout signalling a deadlock detection	adaptive
backoff_per	Time interval an aborted operation waits before restarting	adaptive
cpu_abort	Overhead of aborting an operation	2.0 msec
pr_commit	Probability an Opt_Write operation will commit	0.9
cpu_undo	CPU time to undo an aborted operation	50 msec
io_undo	I/O time to undo an aborted operation	60 msec
$power\_cpu_i$	Speed of CPU at node i	1
$power\_io_i$	Speed of I/O at node i	1
$power\_transc_i$	Speed of transceiver i	1
batch_length	Length of a single simulation batch	2000 job
num_batch	Number of batches for which simulation runs	3
warmup_per	Number of jobs after which statistics are reset	200 job

Table 6.1: System parameters of HARP

node. The parameters $fast_{r0}$, $fast_{r1}$ and $slow_r$ specify the frequency of the Fast_Read_0, Fast_Read_1 and Slow_Read operations respectively. The parameters $fast_w$, $slow_w$ and opt_w specify the frequency of Fast_Write, Slow_Write and Opt_Write operations respectively. These parameters allow the examination of a wide variation of workload mixes.

6.2.2 The Network Model

The physical network modelled is chosen to reflect a hierarchical structure, as shown in Figure 6.1. It consists of 12 nodes, three of them are fully connected through long distance links to form a backbone. The remaining nodes are grouped into three-clusters. Within a cluster, nodes are fully connected through high speed links (LAN) and experience very low delay in sending messages to each other. Each cluster is connected to a node on the backbone through a long distance link. This is similar to the structure of the Internet today: regions of high connectivity with regions weakly connected through a backbone. The logical hierarchy is chosen to match the physical hierarchy; that is it consists of two levels, nodes on the backbone form one cluster at level one and each LAN cluster forms a logical cluster at level two.

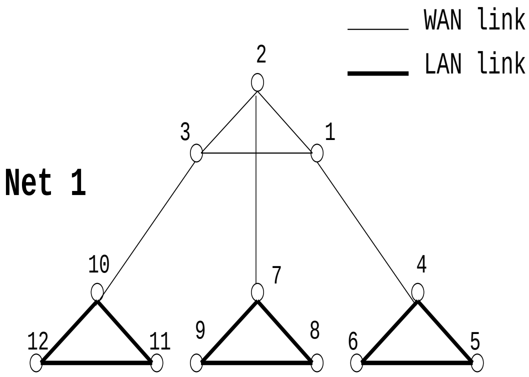

Figure 6.1: The physical network modelled, Net 1

Messages are reliably delivered and FIFO channels are assumed. The system is modelled during normal operation; that is, in the no failure case.

Sending messages between nodes involves the overhead of running the network protocols and the operating system. This overhead is modelled as cpu_msg msec processing at the CPU. A node performs this overhead when sending a message, receiving a message or relaying a message as an intermediate node between a sender and a destination node. It is assumed that there is a multicast facility rather than point-to-point communication. Also messages encounter network delays which involve transmission delays, propagation delays and media access time. Delay over a LAN is modelled as lan_del msec drawn from a uniform distribution and transceivers are modelled as single queue with an infinite number of servers. For WAN links this delay is modelled as wan_del msec, exponentially distributed, and transceivers are modelled as single server single queue. When a node i sends a message to a node j, this involves communication processing overhead (cpu_msg) at i, j and at every intermediate node on the route from i to j. Further, the message encounters network delays at each link (WAN or LAN) it crosses.

6.2.3 Overhead of the Algorithms

The **propagation algorithm** is modelled as a constant overhead (*propg_ovhd_const*), which accounts for the time to check for duplicates, update status, decide from whom a message came and to whom to send it to. Further, an overhead is added for every logical destination it is sent to (*propg_ovhd_var*), which accounts for the time to maintain timers, handle acknowledgements and so on.

For Slow_Write, Slow_Read and Opt_Write, the generic **Two-Phase-Commit algorithm** (2PC) [Bernstein 87] is adopted while collecting quorums. The overhead of 2PC at the coordinator CO (for phase 1 and 2) is modelled as a constant overhead (*pc_ovhd_const*) plus an overhead for each node participating in the quorum (*pc_ovhd_var*). The overhead of acquiring and releasing locks is modelled as *cpu_lock* msec processing at the CPU. The overhead of writing on logs is ignored. The simulator maintains lock tables and explicitly simulates lock contention.

Deadlock avoidance is implemented by using a timeout interval based on the following heuristic [Carey 87]: $deadlock\_timeout = avg(W) + k * \sigma(W)$, where $avg(W)$ is the average waiting time of lock requests, $\sigma(W)$ is the standard deviation of waiting time of lock requests and k is a weighting factor. Hence, *deadlock_timeout* is dynamically adjusted to reflect online estimation of lock request time. k is taken to be 1 as recommended by [Carey 87]. When deadlock is detected, each node releases held locks and aborts the operation. Aborting an operation is modelled as processing time overhead (*cpu_abort*) at CO and each participant involved in the quorum. A $backoff\_period = \alpha * deadlock\_timeout$ is used, then the operation is restarted.

Majority quorum consensus has been adopted to collect quorums from the top cluster for synchronous operations, although any other policy could have been used. Several **quorum collection policies** are discussed in the literature [Ammar 91, Cheung 94, Golding 92a]. Two straightforward solutions are to poll nodes serially one at a time until q nodes have responded (where q is the quorum size) or to send the request in parallel to *all* nodes. The first solution experiences excessive delay while the second incurs high traffic and causes congestion. Other policies [Golding 92a] suggested sending to q nodes in parallel then contacting more if report failures occur or to send to $q + \epsilon$ nodes in parallel. Since the extra nodes to send to are to account for failures, and since in our model it is assumed that failures do not occur, the adopted policy is to send the request to exactly q nodes in parallel. In [Cheung 94], several heuristics have been presented to access q nodes out of m with minimum costs given the topology and communication cost between each pair of nodes. However, since nodes in the top cluster are assumed to be fully connected with the same link speeds, then random selection is adopted.

6.2.4 Operation Modelling

In the following, a description of how each operation is modelled in the simulator is presented. To maintain the version of replicas, the simulator keeps a global data version counter, denoted by *global_version*, and a version counter for each node j, denoted by $version_j$.

Fast_Write : A node i originating a Fast_Write executes the request and updates versions by incrementing the *global_version* and its own version $version_i$. Then, it sends the message to its correspondents according to the propagation algorithm. This involves processing the overhead of the propagation algorithm as well as communication overhead. When the message reaches a destination j, j sends the message to its correspondents (which again involves propagation and communication overhead). Then, it executes the request and updates its own version $version_j$. The same procedure is repeated until the update reaches all leaf nodes. The updates are delivered at the replicas in any order.

Fast_Read_0 : this operation just involves executing the request at the origin node.

Fast_Read_1 : A node i originating a Fast_Read_1, sends the read request to its parent p. When the message reaches p, it executes the read, then it sends back the results to i. When i receives the message, the read is considered complete.

Slow_Write : A node i originating a Slow_Write, if i is not in the top cluster, then it sends the update request to its ancestor in the top cluster, which acts as the *coordinator* (CO) of the operation; otherwise, i acts as its own CO. CO, receiving the message, starts processing phase one of 2PC, acquires locks and sends vote_req message to (quorum size -1) nodes of its neighbours selected randomly. A *participant* receiving vote_req, acquires locks; when obtained it sends vote_yes to CO. When CO receives vote_yes from all participants, it executes the overhead of phase two. At this point the update has committed and CO sends commit message to the participants and the origin node i. Then, it executes the update, updates the *global_version* and its own version and releases locks. Next, it starts propagating the message by sending it to neighbours that did not participate in the quorum and to children other than i. A participant receiving a commit, executes the update, updates its version, releases locks and propagates the message to its children. Node i receiving commit, executes the update, updates $version_i$, returns control to the user then propagates the message to its children. Any other node receiving the message, propagates the message to its children then executes the update and updates its version. If CO times out before a quorum is assembled (a deadlock has occurred), then it sends abort message to all participants, executes the abort protocol and releases locks. It waits for *backoff_per* then restarts. A participant receiving an abort, executes the abort protocol and releases locks.

Slow_Read : The same steps as Slow_Write are followed until CO decides to commit then executes phase two of 2PC. Then, it checks if it has the highest version among the quorum. If it does, then it sends commit to all participants, executes the read request, releases locks and sends results to the origin node. A participant receiving a commit releases locks. If CO is not the most up-to-date copy, it selects one of the participants that has the highest version, sends it an exec_read message, and sends commit to the remaining participants. A participant receiving an exec_read, executes the read request, releases locks and sends back results to CO which will forward them to the origin node.

Opt_Write : Opt_Write originating at top cluster nodes, are exactly the same as Slow_Write. For Opt_Write originating at any other node i, i sends the request to CO, executes the update and returns control to the user. In this study, nodes that are on the path from i to CO are selected to be the *optimistic nodes*; that is, they execute the update as well before CO commits it. When the message reaches CO, it executes 2PC as in Slow_Write. When CO assembles the quorum, it decides whether the update will commit or abort according to an input parameter *pr_commit*. If the decision is to commit, then CO sends commit to all participants, origin node and optimistic nodes, executes the update, updates its version, releases locks and propagates updates to children excluding i and all optimistic nodes. An origin or optimistic node receiving commit, propagates the message to children, excluding any optimistic node. A participant receiving a commit, executes the update, updates its version, releases locks and propagates the message to its children. If the decision was to abort, then CO sends abort to all participants, the origin node and optimistic nodes, executes the abort protocol and releases locks. A participant receiving abort executes the abort protocol and releases locks. The origin or an optimistic node receiving abort, undo the operation.

6.2.5 Parameter Setting

Parameter settings are a difficult issue in simulating a distributed system. Our parameters for the database and requests are chosen to be comparable to previous simulation studies such as [Ciciani 90, Carey 91]. The frequencies of different classes of requests vary from experiment to experiment to generate a variety of load mixes. The values of the parameters relating to the 2PC protocol are in accordance with [Ciciani 90].

The number of nodes is kept small for the tractability of the simulation. A single point in the experiments with 12-nodes networks required a run time on MIPS machines ranging from two to five hours, depending on the load mix, while with 39 nodes it required a run time of eight to 20 hours.

The communication overhead (processing overhead and delay on links) are in keeping with [Gray 88, Agrawala 92, Zhang 94] and with a study on measurements of wide area networks presented in [Pu 91c]. It is based on a Layered Refinement methodology which measures performance characteristics on a live Internet and build an overall profile of end-to-end performance by collecting data and measurement of different layers such as application, operating system and network layers. The delay over a LAN is taken to be 0.02 msec split equally between propagation delay and transmission delay (assuming a message size of 1 Kbyte and a bandwidth of 1 Gbps). The delay over a WAN is taken to be 10 msec split equally between propagation delay and transmission delay (assuming a message size of 1 Kbyte and a bandwidth of 2 Mbps).

For many of the costs, the absolute values of the parameters are not particularly important; the key is their magnitude relative to the other parameters. The default values used for the system parameters are listed in Table 6.1.

6.2.6 Performance Metrics

The following performance metrics are considered:
- *response time*, the time from when the request is initiated until it is completed,
- *commit time*, the time it takes CO to commit a Slow_Write, Slow_Read or Opt_Write,
- *utilisation* of the resources,
- *throughput* of the system.

In order to evaluate the speed of propagation of updates, two metrics are introduced:
- *reach time*, the time at which an update reaches all replicas, and
- $coverage_i$, the time at which an update originating at any node has reached i nodes.

In order to quantify the staleness of information read by a Fast_Read operation performed at node j, a new metric is introduced, denoted by age_j, which is defined as the number of updates that a read has missed. Therefore, the higher the value, the more stale the data is. This metric is computed as:
- $age_j = global\_version - version_j + \gamma * abort\_age_j$, where

$abort\_age_j$ is the number of Opt_Write operations that j has executed but that will abort. Since an optimistic node executes the update before it commits, then if a read occurs meanwhile, its age (i.e.number of missing updates) would be less if the update will eventually commit and will be higher if it will eventually abort. It is assumed that the penalty of reading from a node that has executed x Opt_Writes which will abort, will add x units to its age multiplied by a weighting factor γ. γ is taken here to be 1. Therefore, in the simulation, when a node i executes an Opt_Write that will eventually commit, it increments $version_i$. When a node i executes an Opt_Write that will eventually abort, it increments $abort\_age_i$ which will increase its age. When it receives the corresponding abort, and has undone the update, it decrements $abort\_age_i$, which will restore its age and cancel the effect of having executed an update which will abort.

Also, a metric which is of interest is the *local age* which measures the number of updates missing between a node and other nodes in its neighbourhood rather than the global age. This metric is computed by taking the average of the differences between the version of the node performing the read and the versions of its neighbours, parent and children.

In the experiments, the means of the desired measurements are obtained by using the methods of batch means [Ferrari 78]. *Num_batch* batches have been done for each run. Each run is left to run until at least *batch_length* jobs are generated at each node in the network. To eliminate the warmup effect, after generating *warmup_per* jobs at every node, all the statistics are reset. The results reported here are within the 90% confidence interval for the quantities measured. The measurements were within 10% of their true mean in almost all cases.

6.2.7 Verification of the Simulator

An analytical study has been performed to evaluate the performance of the suggested model. The goal of the study is to verify the results of the simulator. The system has been modelled as an *Open Queueing Network Model with Multiple Job Classes*. An approximate solution based on the *Mean Value Analysis* approach [Lazowska 84] has been adopted and its results has been compared to those obtained from the simulator.

It has been observed that the analytical results are generally close to the ones obtained through simulation. The utilisation of the resources are identical for both models. Although there is a difference between the response and reach times obtained from the analytical model and those obtained through the simulation, the differences are small. Further, the differences are justified by the approximations used in the analytical model. Also, the same conclusions drawn from the simulation results can be deduced from the analytical ones. Therefore, it is believed that the results presented by the simulation are correct. Details of the analytical solution, the results and analysis are presented in [Adly 95d].

6.3 Experiments with Synchronous Updates

In this section, several experiments are presented where updates are either Slow_Write or Opt_Write and reads are a mix of synchronous and asynchronous operations. Sections 6.3.1 through 6.3.4 present the results obtained under several parameter variations, namely, varying the read mix, the mix between Opt_Write and Slow_Write, the load intensity and the communication overhead. Those experiments are performed over the physical network Net 1, shown in Figure 6.1. Section 6.3.5 discusses the performance over different hierarchical network topologies when the read mix is varied. Finally, Section 6.3.6 reports on the results obtained when considering a non-hierarchical topology.

6.3.1 Varying Read Mixes, when all Writes are Slow_Write

Three experiments are carried out to explore the effect of varying a mix of Slow_Read and Fast_Read when all updates are Slow_Write. Experiment A, does not include Fast_Read_1 operations and varies the mix between Fast_Read_0 and Slow_Read. Experiment B, does not include Fast_Read_0 operations and varies the mix between Fast_Read_1 and Slow_Read. Experiment C varies the mix between Fast_Read and Slow_Read operations such that Fast_Read operations are split evenly between Fast_Read_0 and Fast_Read_1. The setting of the parameters is shown below; the remaining parameters are at their default values (Table 6.1).

Exp	$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$
A	1	0	0	0-1	0	0-1
B	1	0	0	0	0-1	0-1
C	1	0	0	0-1*	0-1*	0-1

* The ratio between $fast_{r0}$ to $fast_{r1}$ is kept 1:1

Results are plotted in Figure 6.2 and as shown, as the frequency of Fast_Read increases, the average response time of reads and writes decreases substantially. For instance, for a point where $slow_r$=0.5, there is an improvement in read response over $slow_r$=1 of 56% when all fast reads are Fast_Read_0 (Exp A), 41% when all fast reads are Fast_Read_1 (Exp B) and 48% when fast reads are a combination of both (Exp C).

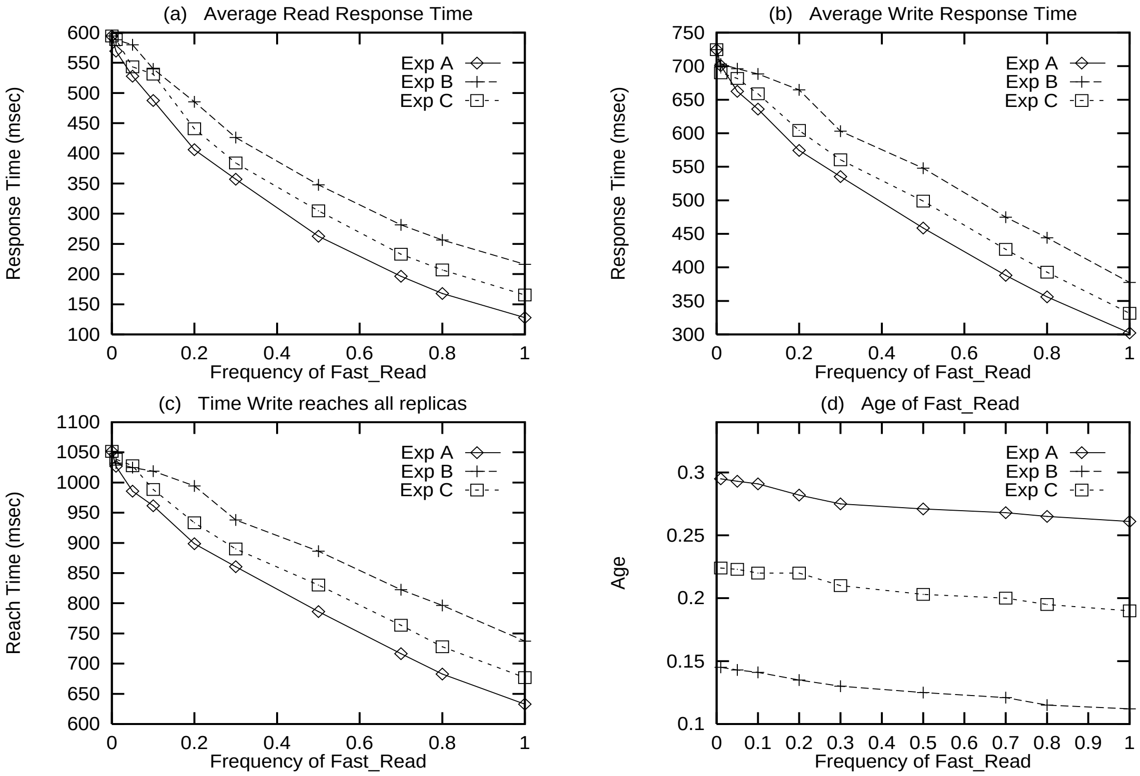

Figure 6.2: Varying read mix, writes are Slow_Write

This improvement is due to the reduction of the load on the top cluster nodes and communication links as Fast_Reads are introduced. Also, the data contention decreases as $slow_r$ decreases, which improves the commit time of Slow_Read and Slow_Write. Consequently, the time at which updates reach all replicas improves significantly as depicted in Figure 6.2(c). The penalty paid by converting Slow_Read to Fast_Read is that fast reads return old information. It is observed that for a point of $slow_r$=0.5; the age of information returned by Fast_Read is around .27 (Exp A), .12 (Exp B), and .2 (Exp C) as shown in Figure 6.2(d). So, it is noticed that it is only a fraction of an update that it is missed (Age=0.2 could be interpreted as one update is missed once in five reads). As the frequency of Fast_Read increases, the age decreases: since the reach time decreases, then replicas are updated more quickly and hence less stale.

From Figure 6.2(d), it is observed that Fast_Read_1 offers better age than Fast_Read_0 with an improvement ranging between 51% to 57%. On the other hand Fast_Read_0 has better performance in terms of response time, nearly 41% lower than Fast_Read_1. Therefore, an even mix of $fast_{r0}$ and $fast_{r1}$ appears to be a good compromise that combines the two benefits.

As shown above, converting Slow_Read to Fast_Read improves all performance metrics with a little loss in age. So, the choice for the mix of synchronous versus asynchronous reads depends on the application: to the extent that the application is ready to accept some loss in the age, as the performance gets better.

6.3.2 Varying Opt_Write versus Slow_Write

Opt_Write is expected to give better response time than Slow_Write, however, it applies the update to the replica before commit takes place. In this experiment, $slow_w$ versus opt_w is varied to study the tradeoffs between Slow_Write and Opt_Write and to evaluate the performance of a combination of load mix between them. Parameters setting is as shown below; the remaining parameters are at their default values.

$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$
0-1	0-1	0	0.25	0.25	0.5

As shown in Figure 6.3(a), the average write response time decreases as opt_w increases. This is because the response returns to the user before the update actually commits. As shown, the response time of Slow_Write, Opt_Write and read are basically the same; since converting Slow_Write to Opt_Write does not alter either the load on processors or the load on the links or the level of the data contention. Therefore, the improvement in write response time is nearly a linear combination of the response of Opt_Write and Slow_Write. As opt_w increases, the coverage time decreases (Figure 6.3(b)), especially for the first few nodes (optimistic nodes), then it follows the same curve as Slow_Write.

However, as shown in Figure 6.3(c), the average age increases as opt_w increases. The reason of the degradation is that updates commit at local replicas and update the global version

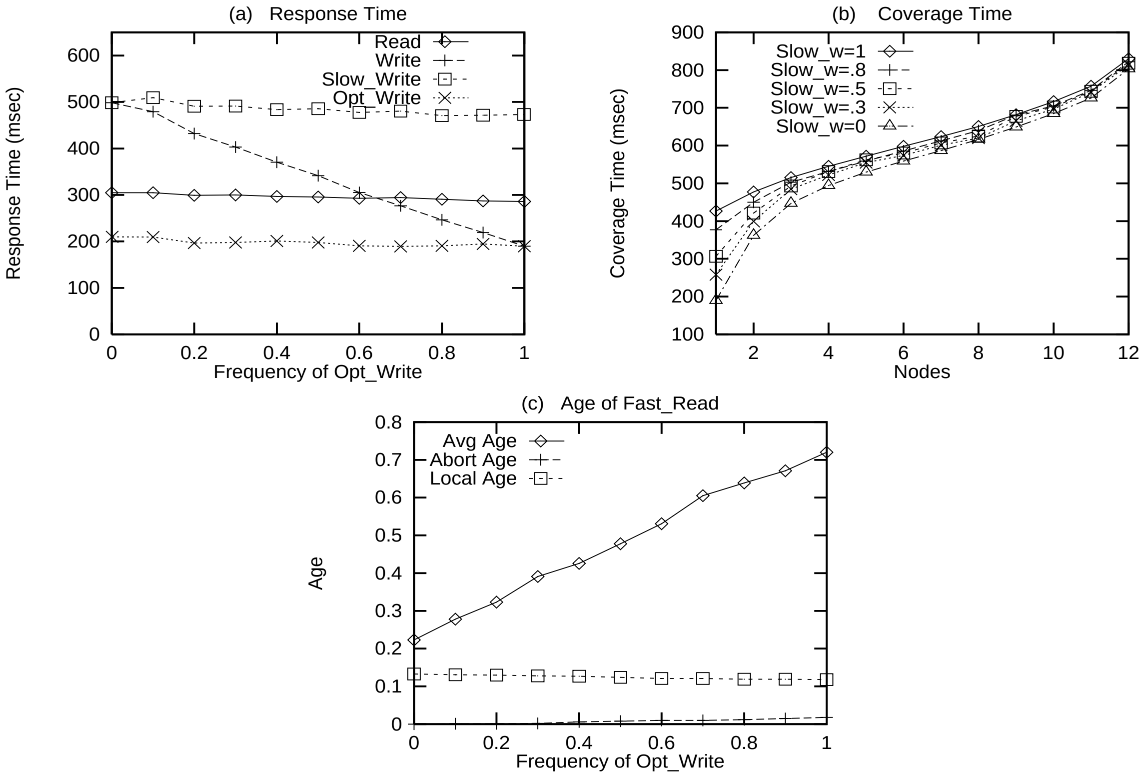

Figure 6.3: Varying Opt_Write versus Slow_Write

rather than updating it while assembling the quorum. So nodes at the other extreme of the tree experience higher age. The local age is slightly better with Opt_Write due to the improvement in the coverage time. The abort age increases as opt_w increases but it is quite low.

Therefore, it seems that an even mix between Opt_Write and Slow_Write can compromise the benefit of reduction in write response time, coverage and local age with the loss in the overall age.

In a different experiment, not plotted here, the probability by which an Opt_Write commits (*pr_commit*) is varied. It has been observed that, as expected, as *pr_commit* decreases the throughput decreases and the abort age increases due to the large number of aborts. The results have revealed that applications that can guarantee that more than 80% of their Opt_Write will commit can combine the benefits of reduction in response time and coverage while not suffering from severe reduction in throughput. Further, it can still benefit from a low local age and does not suffer from an increase in the abort age.

6.3.3 Varying the Load Intensity

In this experiment, the arrival rate is varied to see how sensitive performance is to the load intensity. Two loads are considered with one load (A) containing more synchronous read operations than the other load (B). The parameter setting is as follows:

Exp	$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$	$inter\_arr\_time$
A	0.5	0.5	0	0.25	0.25	0.5	600-1400 msec
B	0.5	0.5	0	0.4	0.4	0.2	600-1400 msec

For load A, as shown in Figure 6.4(a), the response time increases following an exponential curve. Steady state was not able to be reached for *inter_arr_time* less than 700 msec; limiting the overall throughput to 0.017 request/msec. Although utilisation of resources increases, they are still very lightly loaded as shown in Figure 6.4(b). Therefore, the bottleneck of the system lies in data contention (contentions for locks). The reach time also follows an exponential curve (Figure 6.4(c)); which is due the fact that the updates are propagated only after commitment, and are affected by contention for locks. Consequently, the age of information returned by fast reads as well as the overall age seen by all reads increase as shown in Figure 6.4(d).

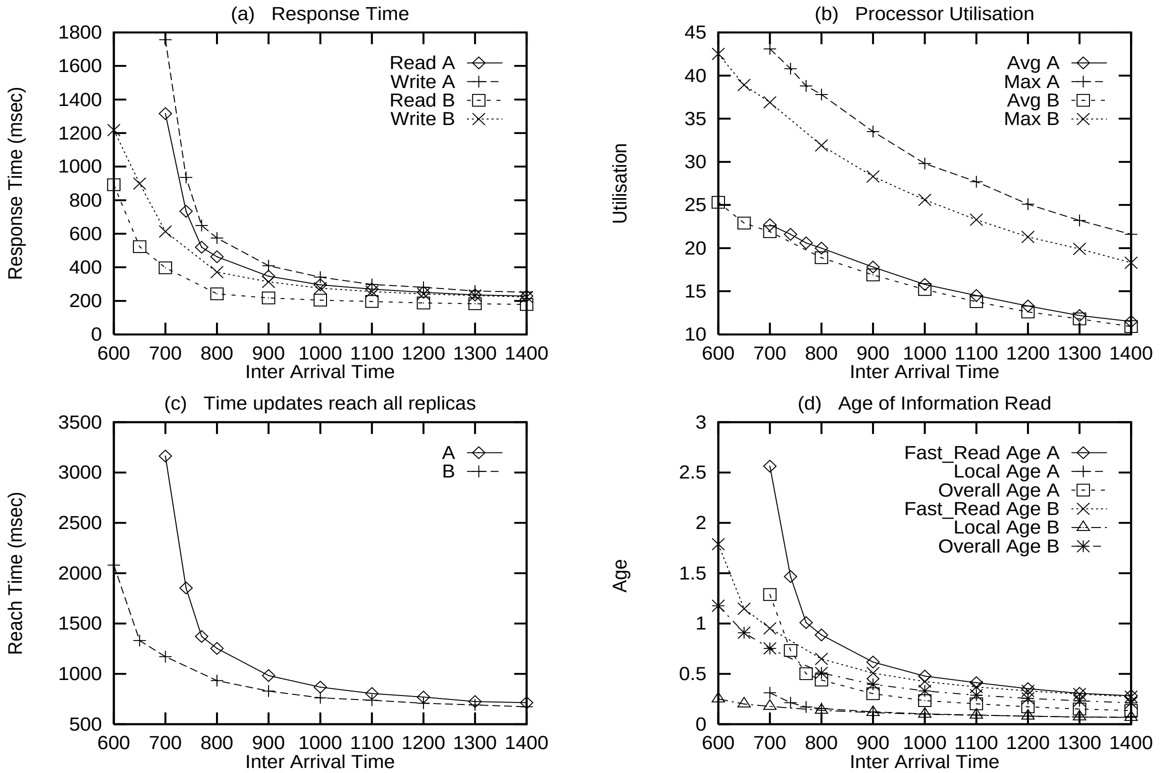

Figure 6.4: Varying the arrival rate, writes are synchronous

Load B shows the effect of converting some Slow_Read to Fast_Read. Reducing $slow_r$ from 0.5 to 0.2 reduces the data contention and consequently, the response times of reads and writes reduce as shown in Figure 6.4(a). The reduction in read (write) response is around 31% (19.5%) for *inter_arr_time* 1000, it increases to 45% (23.4%) for *inter_arr_time* 900 and continues increasing as the load increases. Further, it is able to cope with a higher arrival rate increasing the throughput by 16.8%. The reach time is better in B than A and the improvement is more obvious for high load. The age returned by Fast_Read is nearly the same for low loads at A and B, but it is much less for B as the load increases as shown in Figure 6.4(d). It was interesting to see that although load B has only 20% of its read Slow_Read while A has 50%, the reduction in the age returned by Fast_Read has affected the overall age such that the overall age of B is only slightly higher than A for low loads and is much better than A for high loads.

From here one can deduce that increasing a load involving synchronous operations slows down the system operation and affects the performance tremendously. Response time deteriorates as well as age while resources are underutilised and throughput is limited due to data contention. Therefore, this calls for applications that need better performance and throughput to relax consistency to reduce data contention; if they can afford it.

In another experiment, not reported here, the mix between reads and writes ($freq_r$) was varied to explore the limitations that synchronous writes impose on the performance. It was noticed that performance degrades by increasing the frequency of writes, which was again due to data contention as writes do not share locks while reads do. A steady state could not be reached for a frequency of writes more than 27%, when locks are saturated; limiting utilization of the resources to 23% on the average.

6.3.4 Varying the Communication Overhead

The following experiments explore the sensitivity of performance to the delay on the network. The ratio of *lan_del:wan_del* is varied from 1:10 to 1:5000. First, *lan_del* is fixed to 0.02 msec and *wan_del* is varied from 0.2 to 100 msec. Then, *wan_del* is fixed to 10 msec and *lan_del* is varied from 0.002 to 1.0 msec.

The performance has shown to be is insensitive to varying *lan_del*. This is expected since it has low values so it does not have a great impact on performance in comparison to *wan_del*. Therefore, the results are not shown and this parameter can be factored out. Results are shown for varying *wan_del* for two load mixes; where load A contains more synchronous reads than load B. Parameters setting is as follows:

Exp	$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$	*wan_del*
A	0.5	0.5	0	0.25	0.25	0.5	0.2-100 msec
B	0.5	0.5	0	0.4	0.4	0.2	0.2-100 msec

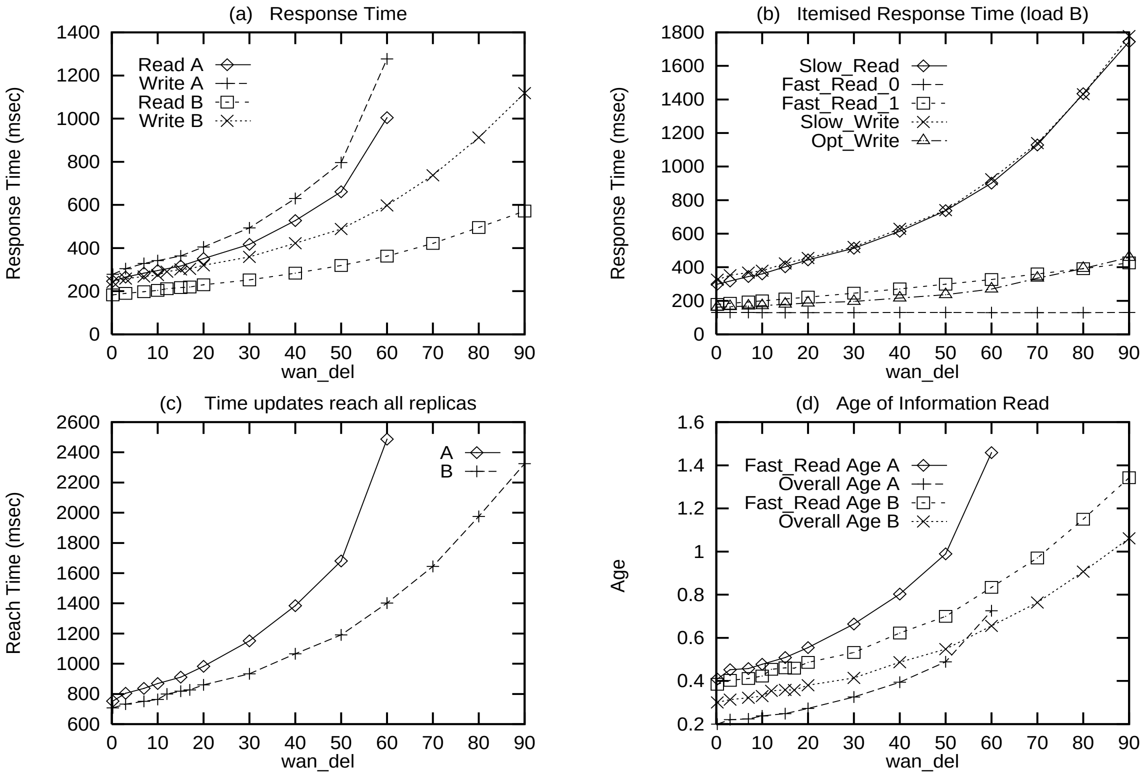

Figure 6.5: Varying *wan_del*, writes are synchronous

As shown in Figure 6.5(a), the response times of reads and writes are very sensitive to *wan_del*. They are quite low for values of *wan_del* less than 20 msec then, they increase sharply. It is obvious that load A is more sensitive to *wan_del* than load B and saturation is reached more quickly. Figure 6.5(b) shows that the responses of Fast_Read_0, Fast_Read_1 and Opt_Write increase only slightly, while those of Slow_Write and Slow_Read increase sharply and dominate the results. The resources were found to be underutilised and data contention was the bottleneck of the system. The reason is that: while *wan_del* increases, the time it takes an operation to assemble a quorum and commit increases and it holds locks for a longer time. Consequently, the time other operations wait for locks increases. As shown in Figure 6.5(d), the age returned by Fast_Read with A is higher than B since updates take longer to commit and consequently take longer for reaching other replicas. The overall age of A is slightly lower than B for low values of *wan_del* then the gap decreases and eventually it becomes higher than B for high values of *wan_del*.

From here it can be concluded that performance is very sensitive to *wan_del*. Synchronous operations suffer and dominate the degradation in performance with the increase of the delay. This suggests that if applications can accept observing some stale information then they should increase their frequency of Fast_Read, especially when the network delay becomes higher, as this offers better performance as well as lower age.

In a different experiment varying the communication processing overhead (*cpu_msg*), results have revealed that performance is quite sensitive to high values of *cpu_msg* due to high CPU contention and data contention, especially when the load mix has a high frequency of synchronous operations. The performance has shown nearly the same sensitivity to *cpu_msg* as *wan_del*, hence, the details are not described.

6.3.5 Varying the Hierarchical Network Topologies

In the following experiments, the performance is evaluated for various hierarchical physical (and hence logical) networks. Since the number of possible topologies can be infinite, we have limited our choices to a 12-nodes network, three of them are fully connected through long distance links to form a backbone and the remaining nodes are grouped into 3-clusters each forming a LAN. The connections between the clusters are varied to reflect certain aspects that might affect performance. The different networks considered are shown in Figure 6.6.

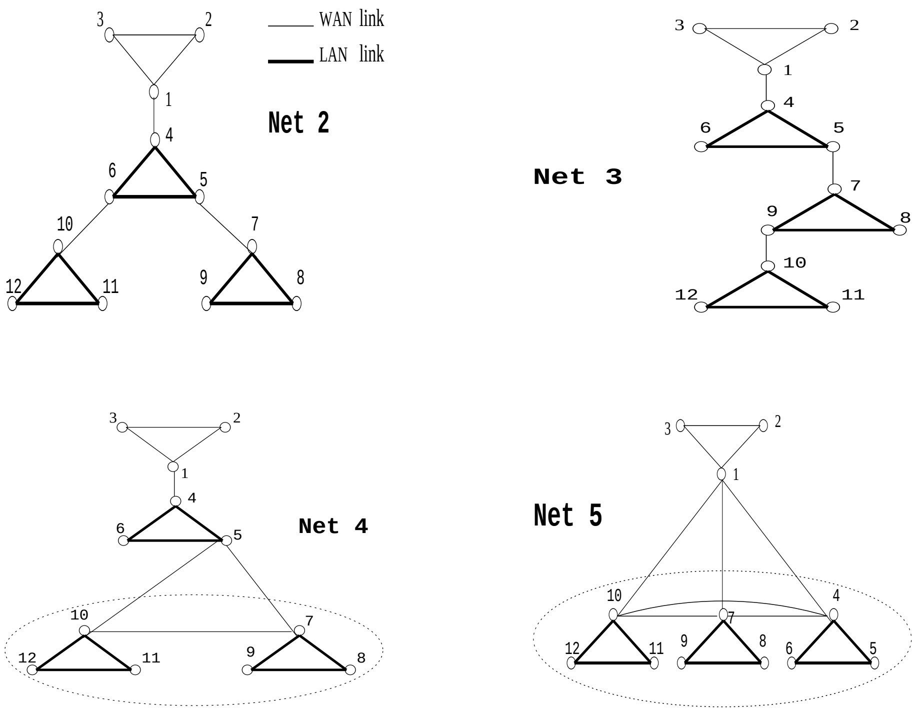

Figure 6.6: Different hierarchical network topologies for 12 replicas

As shown, Net 2, Net 3 and Net 4 explore the effect of increasing the number of levels of the hierarchy. Net 2 spans three levels by connecting one of the LAN clusters to the backbone, and the two other clusters are linked to the backbone through that cluster. Net 3 is organised like a chain, spanning four levels. Net 4 shows the effect of connecting the two clusters in the bottom level -forming one logical cluster- and both clusters are connected through the same node (node 5) to the higher level. Net 5 explores the effect of connecting the three LAN-clusters at the bottom levels, forming one logical cluster, which is linked to the backbone via node 1.

For this set of experiments, the read mix is varied and the results are compared with those of Net 1. The parameters setting is as follows:

$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$
0.5	0.5	0	0-1*	0-1*	0-1

* The ratio between $fast_{r0}$ to $fast_{r1}$ is kept 1:1

It was observed that for the four networks, node 1 is overloaded with extra communication and 2PC overhead; since it acts as a CO for most of the nodes. The load on this node is the worst in Net 5 since it acts as the parent of most nodes as well, hence, it performs all their Fast_Read_1 requests, which incurs extra processing and communication overhead. Increasing the number of levels (Net 2, Net 3 and Net 4) increases the time Slow_Write and Slow_Read spend in communication in order to send requests and receive results from COs, which increases the response time, being the highest in Net 3 since it spans four levels. Further, on these networks, the traffic on the link between nodes 1 and 4 was observed to be higher than in Net 1. Due to the above factors, the response time of Slow_Write and Slow_Read on all networks is higher than on Net 1 and the gap is larger for higher frequency of Slow_Read.

A sample of the results of Net 5 and Net 2 are plotted in Figures 6.7 and 6.8 respectively. Results of Net 3 and Net 4 are nearly identical to Net 2 but with a slight degradation. For Net 3 this is due to the higher communication time spent to cross the four levels. For Net 4 it is due to the extra load placed on node 5 which performs more communication and processing overhead.

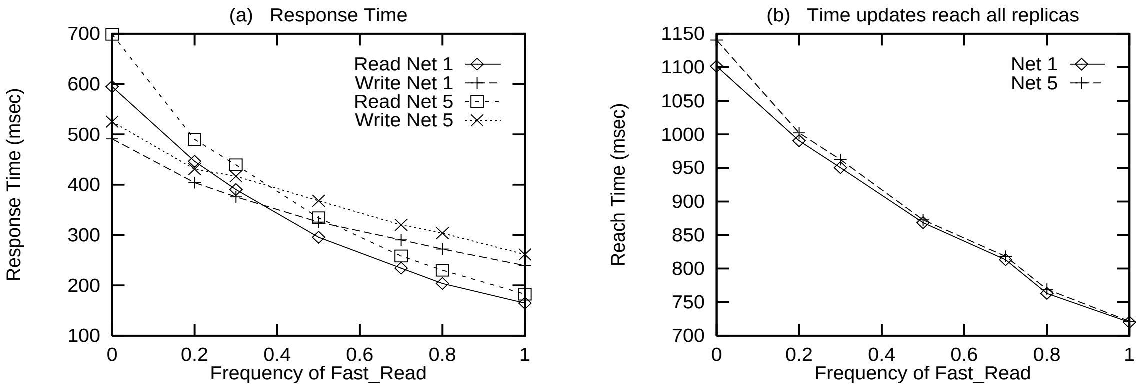

Figure 6.7: Varying read mix on Net 5, writes are synchronous

For Net 2, Net 3 and Net 4, since updates take longer to commit and since propagated updates have to travel down three or four levels instead of two, the reach time is observed to

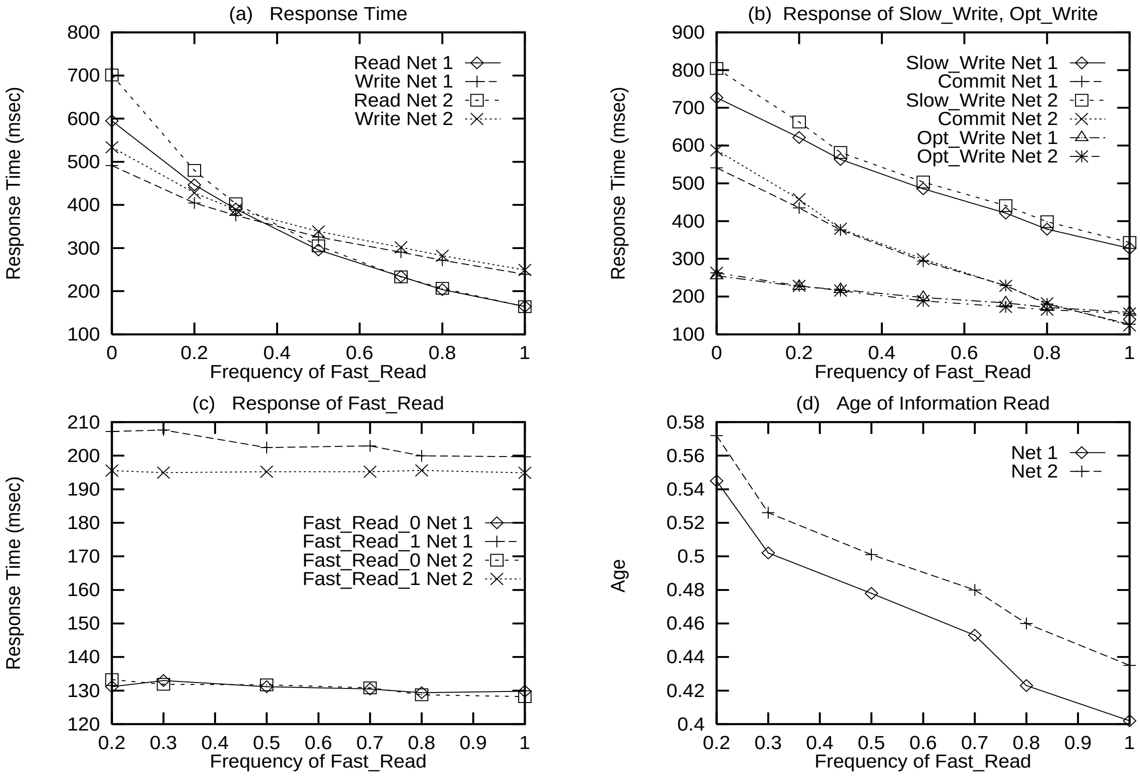

Figure 6.8: Varying read mix on Net 2, writes are synchronous

be constantly higher than on Net 1. On Net 5, an update on a leaf node needs to cross fewer communication lines than in Net 1 to reach all other replicas. Hence, the communication time to propagate an update on Net 5 is less than Net 1. However, the increase in the commit time is higher than the decrease in the communication time and causes the reach time to be higher than in Net 1, as shown in Figure 6.7(b). Consequently, the age returned by Fast_Read is worse than Net 1 for the four networks.

On Net 2, Net 3 and Net 4, the response time of Fast_Read_1 experiences lower value than Net 1, especially for high frequency of Slow_Read. This is because on those networks, nearly half of the nodes grant their Fast_Read_1 requests from parents which are not in the top cluster, hence, less loaded, while in Net 1 all nodes grant their requests from the top cluster. In Net 5, the response of Fast_Read_1 is constantly higher than in Net 1 due to the excessive load on node 1.

From the above it can be concluded that Slow_Write and Slow_Read perform better on Net 1 than on the four networks considered; mainly because the load is evenly distributed among the nodes in the top cluster. Increasing the number of levels in the hierarchy exempts the top cluster from executing some of the Fast_Read_1 requests, which experience lower response time on those networks. However, the response time of Slow_Write and Slow_Read increases and the age returned by Fast_Read is higher due to the increase in the reach time, hence, it is a tradeoff.

6.3.6 Comparing with a Non-hierarchical Network

This experiment shows the performance over a network where all nodes are in one logical cluster, more or less fully connected (Net 6), as shown in Figure 6.9. Applying synchronous operations on such a network represents the performance of standard protocols used in the literature, more precisely, using majority quorum consensus over the whole network. Since there are no levels in the hierarchy, then Opt_Write is converted to Slow_Write and Fast_Read_1 is converted to Fast_Read_0. In order to compare results with those of Net 1, the experiment varies a read mix consisting of Slow_Read and Fast_Read_0 when all updates are Slow_Write. That is the parameters setting is as follows:

$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$
1	0	0	0-1	0	0-1

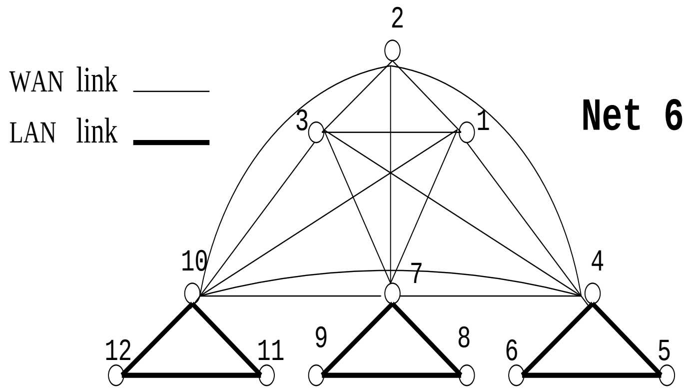

Figure 6.9: A non-hierarchical network, Net 6

The results are plotted in Figure 6.10 and it has been observed that:

• Steady state was not able to be reached on Net 6 for $slow_r > 0.5$. Assembling a quorum from seven nodes (versus two in Net 1), increases data contention. Further, since each node can be a CO, the rate at which deadlock occurs is higher. Finally, resources are more loaded since all nodes are involved with 2PC and communication overhead. These factors increase the time by which operations commit and eventually no steady state could be reached.

• For loads where a steady state could be reached, most of the performance metrics showed a degradation. The average response time of writes is higher than in Net 1, with a degradation between 46% and 52%, as shown in Figure 6.10(a), due to high commit time. Although in Net 6 Slow_Write and Slow_Read avoid the communication time needed to send requests and receive results from CO, this does not compensate for the huge increase in the commit time.

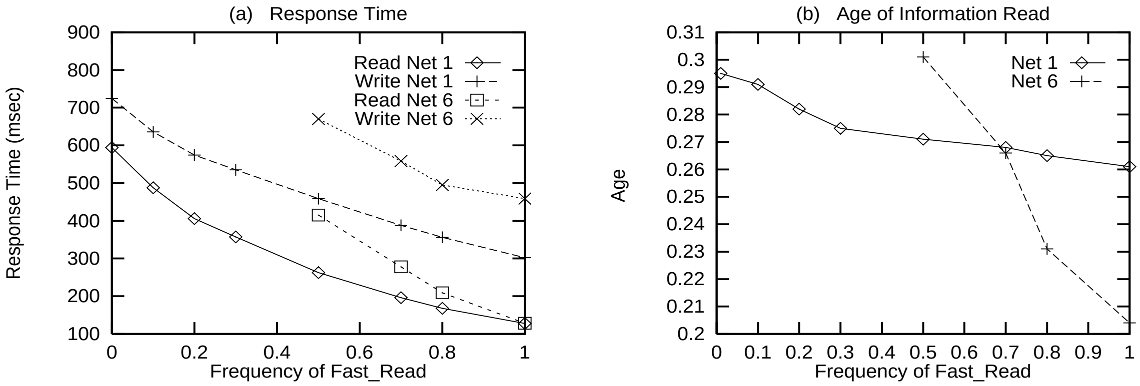

Figure 6.10: Varying read mix on Net 6, writes are synchronous

The average response time of reads on Net 6 is higher than Net 1 for high $slow_r$, then it decreases to approach Net 1 when all reads are Fast_Read_0.

The time at which updates reach replicas for Net 6 is higher than Net 1. Although Net 6 needs to cross fewer links than Net 1 to propagate updates to other replicas not participating in the quorum, the increase in the commit time is higher than the reduction in the communication time. The age returned by Fast_Read in Net 6 is lower than that of Net 1 for high frequency of Fast_Read, as shown in Figure 6.10(b). Since a majority of the nodes participate in the quorum, they are up-to-date, and since Fast_Read is performed directly from them, this explains the lower age. However, as $slow_r$ increases, the commit time of updates increases which slows down the speed of propagation. Consequently, the age in Net 6 increases and bypasses that of Net 1 when more than 30% of the reads are Slow_Read as shown.

In conclusion, it is observed that performance of synchronous operations on Net 6 is much worse than Net 1. Throughput is limited tremendously, especially with high loads. Within Net 1, HARP is able to achieve better throughput for the same types of operations since it assembles a smaller quorum. The response times of Slow_Write, Slow_Read and Fast_Read_0 are worse than Net 1, as well as the time at which updates reach all replicas. The gain in the age over Net 1 is only for high frequency of Fast_Read, and the improvement is of a maximum of 21%, which appears to be much less than the losses in all other metrics.

6.4 Experiments with Asynchronous Updates

In this section, several experiments are presented where updates are Fast_Write. Sections 6.4.1 through 6.4.3 consider varying the read mix, the load intensity and the communication overhead to observe how sensitive asynchronous operations are to those parameters. Also, the results are compared with those when updates are synchronous. These experiments are performed over the physical network Net 1. Section 6.4.4 presents the performance when other hierarchical network topologies are considered and the read mix is varied. Finally, Section 6.4.5 reports on the results obtained when considering a non-hierarchical topology.

6.4.1 Comparing Slow_Write versus Fast_Write, Varying a Mix of Fast_Reads

In this experiment, a mix of Fast_Read is varied for two loads: load A where all updates are Fast_Write and load B where all updates are Slow_Write. The parameters setting is as follows:

Exp	$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$	$freq_r$
A	0	0	1	0-1	0-1	0	0.7
B	1	0	0	0-1	0-1	0	0.7

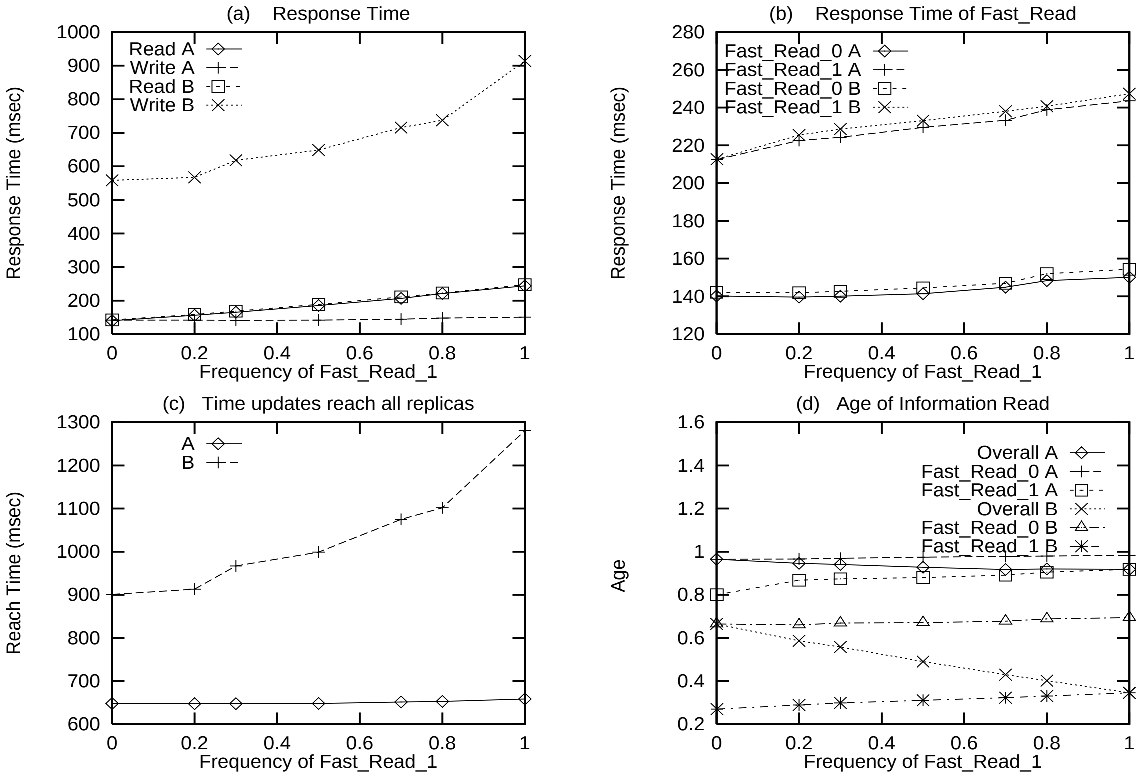

Figure 6.11: Comparing Slow_Write versus Fast_Write, varying a mix of Fast_Read

There is a huge difference between the write response of Fast_Write and Slow_Write, ranging between 3.9 and and 6 times better, as shown in Figure 6.11(a), since Fast_Write commits at the local replica. Also, the increase in the response of Slow_Write with the increase of $fast_{r1}$ is much larger than that of Fast_Write. This is because with Fast_Read_1 nodes in the top cluster are more loaded which increases the commit time and consequently the response time. The response time of fast reads with Slow_Write is observed to be slightly higher than with Fast_Write, as shown in Figure 6.11(b).

Figure 6.11(c) shows the reach time which is much higher for Slow_Write than Fast_Write and it increases sharply as $fast_{r1}$ increases (it is 27% higher for $fast_{r1}$=0 and 95% higher for $fast_{r1}$=1), which is due to the high commit time. Similarly, the coverage time was observed to be much better with Fast_Write than Slow_Write. However, the age returned by Fast_Read with Slow_Write is better than that with Fast_Write, as shown in Figure 6.11(d). This is expected as replicas in the top cluster with Slow_Write are updated first, hence the age (especially that of Fast_Read_1) is better. However, since the reach time is better with Fast_Write than Slow_Write, so replicas with Fast_Write catch up quickly. It is observed that the improvement in the age with Slow_Write ranges between 31% to 62%.

In conclusion, with Fast_Write every performance metric is better than with Slow_Write except the age. However, the loss in age is considered to be low and the benefits appear to be much higher than the losses.

As before, it is observed that Fast_Read_1 offers better age in exchange of some loss in the response time. For instance, for an even mix of Fast_Read_0 and Fast_Read_1, Fast_Read_1 offers an improvement in the age of 10% for load A and 54% for load B, with a degradation in the response time of 60%.

6.4.2 Varying the Load Intensity

This experiment explores the sensitivity of asynchronous operations to the load intensity and compares it to that of synchronous operations. Therefore, it compares a load (A) consisting of Fast_Write and Fast_Read with a load (B) where updates are a mix of Slow_Write and Opt_Write and reads are a mix of Slow_Read and Fast_Read while the arrival rate is varied. Parameters setting is as follows:

Exp	$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$	*inter_arr_time*
A	0	0	1	0.5	0.5	0	250-1300 msec
B	0.5	0.5	0	0.25	0.25	0.5	250-1300 msec

Results are plotted in Figure 6.12. As shown, the performance of load A degrades as the arrival rate increases, which is mainly due to resource contention. However, in comparison with load B, it is observed that with asynchronous operations we can achieve a much higher throughput than with synchronous (2.8 times higher). Further, the performance is much better with asynchronous operations; for instance for a point with *inter_arr_time* = 700 msec, the response time of writes is 12.7 times lower, the response time of reads is 7 times lower and the reach time is 5 times lower. Even the age returned by Fast_Read is better: 0.682 versus 2.563 due to the quick coverage and low reach time associated with Fast_Write.

Similar results have been obtained by varying the mix between reads and writes and com-

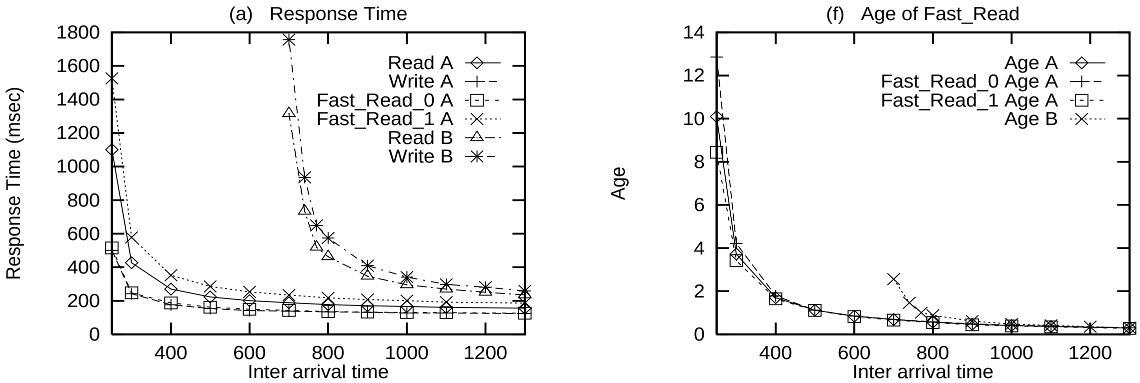

Figure 6.12: Varying the arrival rate, writes are asynchronous

paring synchronous and asynchronous operations.

6.4.3 Varying the Communication Overhead

Figure 6.13 reports results of an experiment set to compare the behaviour of asynchronous and synchronous operations under the same load intensity while varying the communication processing overhead (*cpu_msg*). The parameters setting is as follows:

Exp	$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$	*cpu_msg*
A	0	0	1	0.5	0.5	0	0-15 msec
B	0.5	0.5	0	0.25	0.25	0.5	0-15 msec

As shown in Figure 6.13(a), the average read and write response times of load A are far better than those of B. The difference is small for small values of *cpu_msg* then it increases as *cpu_msg* increases. The reason is due to that the communication overhead that load B places on the processors is much more than in A as shown in Figure 6.13(b). Consequently, the time at which updates reach replicas in A is much better than B (Figure 6.13(c)). Although it increases with the increase of *cpu_msg*, the increase is very small in comparison to that of B. It is interesting to see that although load B contains mostly synchronous updates, the age returned by Fast_Read in A is better than B, especially for high values of *cpu_msg*, as depicted in Figure 6.13(d).

Similar results have been obtained by varying the network delay. From here one can say that asynchronous operations are less sensitive to communication overhead than are synchronous operations and they perform better, especially under systems with high communication overhead.

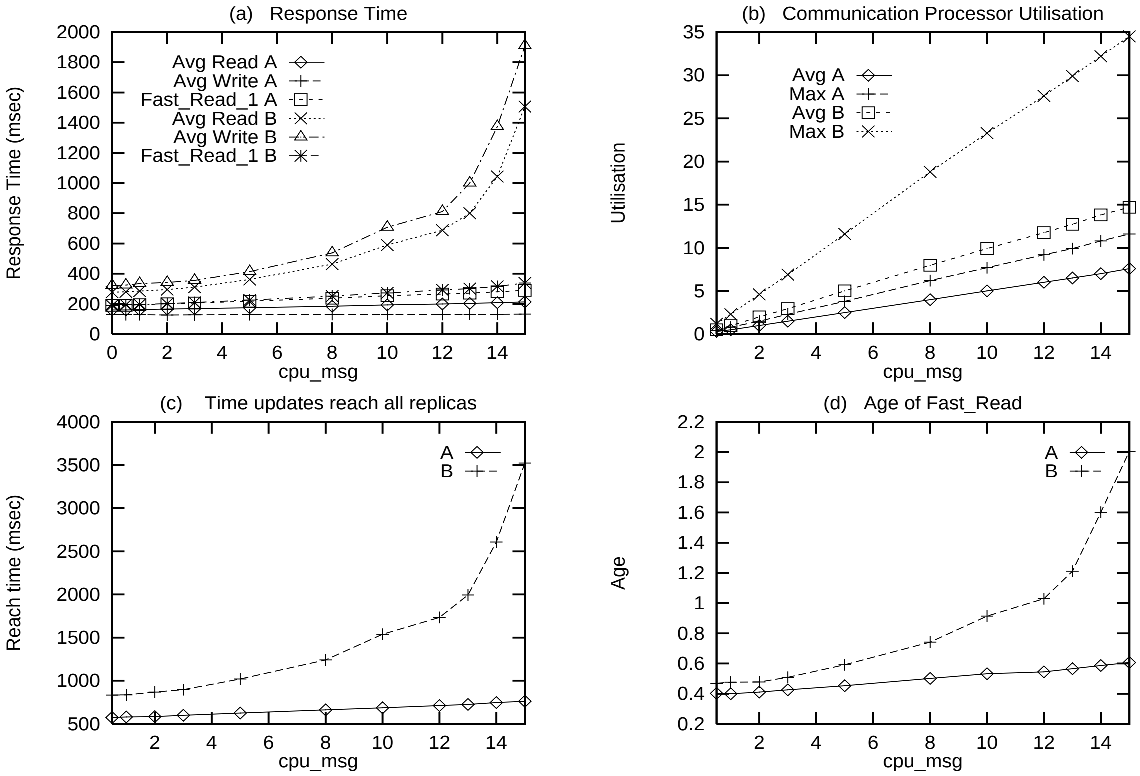

Figure 6.13: Varying *cpu_msg*, comparing asynchronous and synchronous operations

6.4.4 Varying the Hierarchical Network Topologies

This section explores how different hierarchical network topologies can affect the performance of asynchronous operations. The same set of networks shown in Figure 6.6 is considered. For this set of experiments, the read mix between Fast_Read_0 and Fast_Read_1 is varied when updates are Fast_Write and the results compared to those of Net 1. Since asynchronous operations can cope with higher load than synchronous operation, the experiments are set to generate a higher load than the default. The following setting is considered:

$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$	*inter_arr_time*	$freq_r$	*cpu_msg*	*wan_del*
0	0	1	0-1	0-1	0	600 msec	0.7	5 msec	30 msec

The results have shown that, for low frequency of Fast_Read_1, the performance of Net 2 and Net 3 is similar to Net 1. For high frequency of Fast_Read_1, Net 2 and Net 3 offer better response time because they relieve nodes in the top cluster from satisfying all Fast_Read_1 requests, hence the load is more balanced. However, the age of information read is higher than Net 1 because Fast_Read_1 requests are not satisfied from the most up-to-date nodes. The increase in the age is higher in Net 3 because the reach time experiences higher value on this network since an update needs to cross more hops to reach all replicas. Results of Net 3 are plotted in Figure 6.14; those of Net 2 are omitted since they were found quite

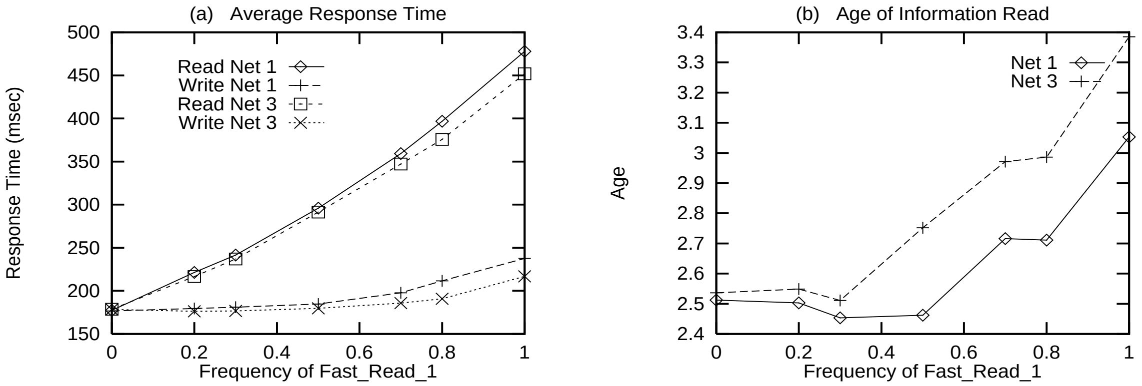

Figure 6.14: Varying read mix on Net 3, writes are asynchronous

similar to Net 3, just slightly better.

For Net 4 and Net 5, it is observed that the response time is higher than in Net 1, especially for high frequency of Fast_Read_1, being worst in Net 5. Results of Net 4 are plotted in Figure 6.15. This is mainly due to the excessive load Net 4 and Net 5 place on node 5 and node 1 respectively. This extra load consists of the processing overhead of granting Fast_Read_1 requests as well as excessive communication and propagation overhead since those nodes are the link between their child clusters and the rest of the network. The reach time and the coverage time of the first few nodes in Net 4 are better than in Net 1 because the nodes in the bottom cluster are directly connected, so updates propagate faster to their neighbours. However, this improvement applies only for low frequency of Fast_Read_1, then the reach time increases sharply due to the contention on node 5. Consequently, the age is slightly lower for low $fast_{r1}$, then it increases by more than 45% at $fast_{r1}$=1. Although in Net 5 updates need to travel fewer hops to reach all replicas, the reach time was found to be higher than Net 1 as well as the age because of the contention on the resources.

In conclusion, this set of experiments has revealed that increasing the number of levels in the hierarchy reduces the load on nodes in the top cluster giving better response times, but information read is more stale. So, it is a tradeoff. Further, it has shown that increasing the number of communication lines connecting lower level clusters and grouping them into one logical cluster would increase the speed of propagation of updates and reduce the staleness only if the load is evenly distributed among all nodes in the network.

6.4.5 Comparing with a Non-hierarchical Network

This experiment shows the performance of asynchronous operations over the fully connected network Net 6 (Figure 6.9) when the arrival rate is varied and results are compared to Net 1. Since Net 6 involves 15 WAN links while Net 1 involves only six, assuming that lines of Net 6 have the same bandwidth as Net 1 seems unfair, if economical constraints are taken into

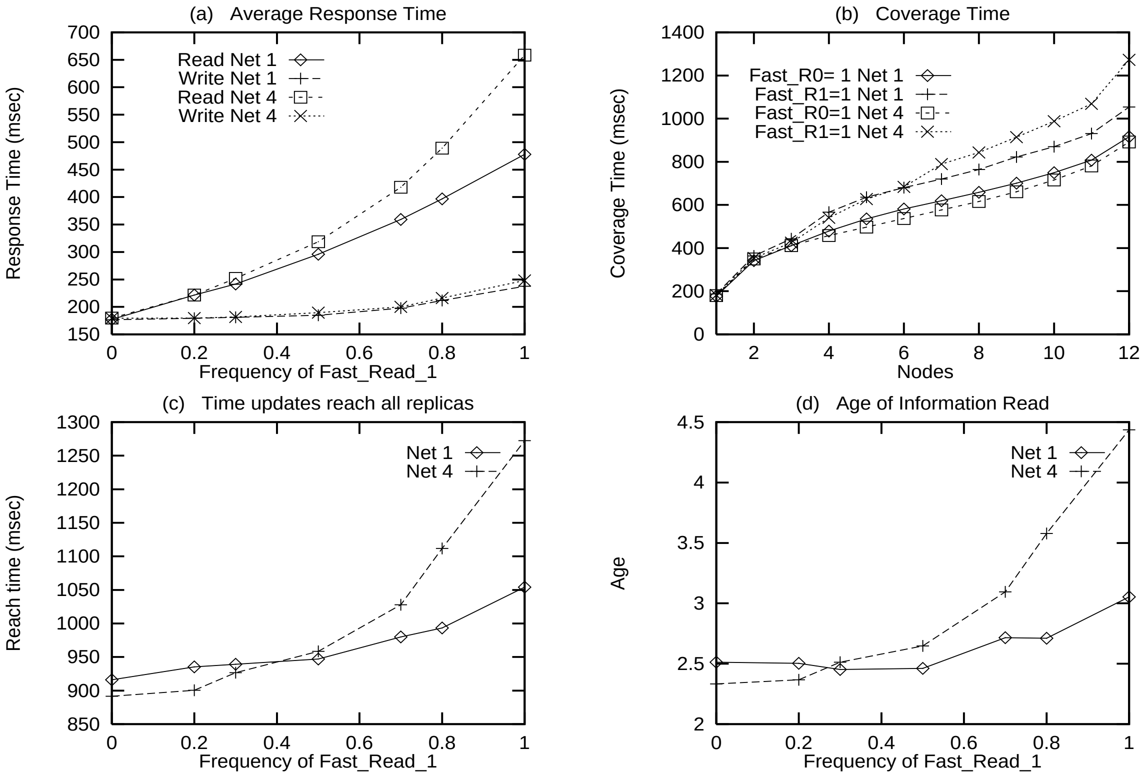

Figure 6.15: Varying read mix on Net 4, writes are asynchronous

account in the comparison. Therefore, two versions of Net 6 are considered. The first one (Net 6_A), assumes the bandwidth per line is equal to those of Net 1. The second one (Net 6_B), simulates budget constraints by assuming that the total bandwidth is constant and the bandwidth is reassigned per line accordingly. Therefore, *wan_del* is set to 75 msec for Net 6_B versus 30 msec for Net 1 and Net 6_A. The parameters are set as follows:

Exp	$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$	$freq_r$	*cpu_msg*	*wan_del*	*inter_arr_time*
Net 1	0	0	1	1	0	0	0.5	5 msec	30 msec	500-1200 msec
Net 6_A	0	0	1	1	0	0	0.5	5 msec	30 msec	500-1200 msec
Net 6_B	0	0	1	1	0	0	0.5	5 msec	75 msec	500-1200 msec

As shown in Figure 6.16(a), the response times for both Net 6_A and Net 6_B are higher than Net 1 with a difference about 30-40 msec for high loads. This is because more processing communication overhead is placed on the nodes in Net 6 due to the extra communication lines. The processing load on the resources of those networks was observed to be constantly higher than Net 1.

For the same reasons, the reach time is higher in Net 6_A than in Net 1. It is the highest in Net 6_B, since the transmission delays are higher which creates more contention at the communication lines. Consequently, the age is the best in Net 1, followed by Net 6_A and

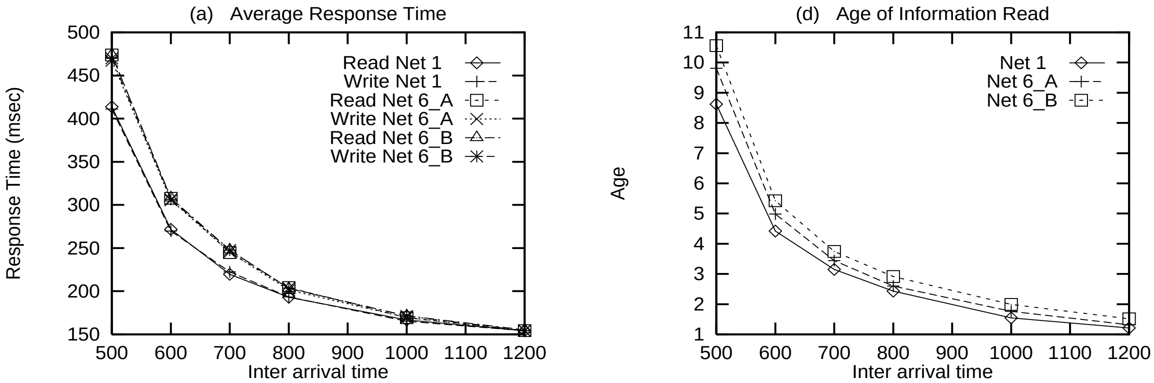

Figure 6.16: Varying the arrival rate on Network 6, writes are asynchronous

is the worst in Net 6_B as shown in Figure 6.16 (b).

Therefore, it can be concluded that the performance of the fully connected network with asynchronous operations is constantly worse than Net 1 especially under high loads.

6.5 Experiments with a Larger Number of Nodes

These experiments explore the performance over a network consisting of 39 nodes to see how a larger degree of replication can affect performance. Three different physical (and hence logical) structures are considered, as shown in Figure 6.17. The experiments consider a load consisting of synchronous updates and varying a mix of Slow_Read and Fast_Read. The following setting is used:

$slow_w$	opt_w	$fast_w$	$fast_{r0}$	$fast_{r1}$	$slow_r$	$inter\_arr\_time$
0.5	0.5	0	0-1*	0-1*	0-1	3300 msec

* The ratio between $fast_{r0}$ to $fast_{r1}$ is kept 1:1

As shown in Figure 6.18, comparing the results of Net 7 and Net 8, it is observed that performance of Net 8 is constantly worse than Net 7. It could not cope with high loads especially with a high frequency of Slow_Write and Slow_Read (steady state was not able to be reached for $slow_r > 0.5$). This is because Net 8 places more load on the resources due to larger quorum size which increases the commit time and the rate of deadlock. Also, Net 8 places more communication overhead on the nodes in the top cluster due to the extra communication lines which increases the resource contention.

Comparing Net 7 and Net 9, as shown in Figure 6.19, Net 7 shows better performance than Net 9 since it distributes the load more evenly among the nodes and does not overload nodes in the top cluster with extra communication and propagation overhead and processing most of the Fast_Read_1 requests.

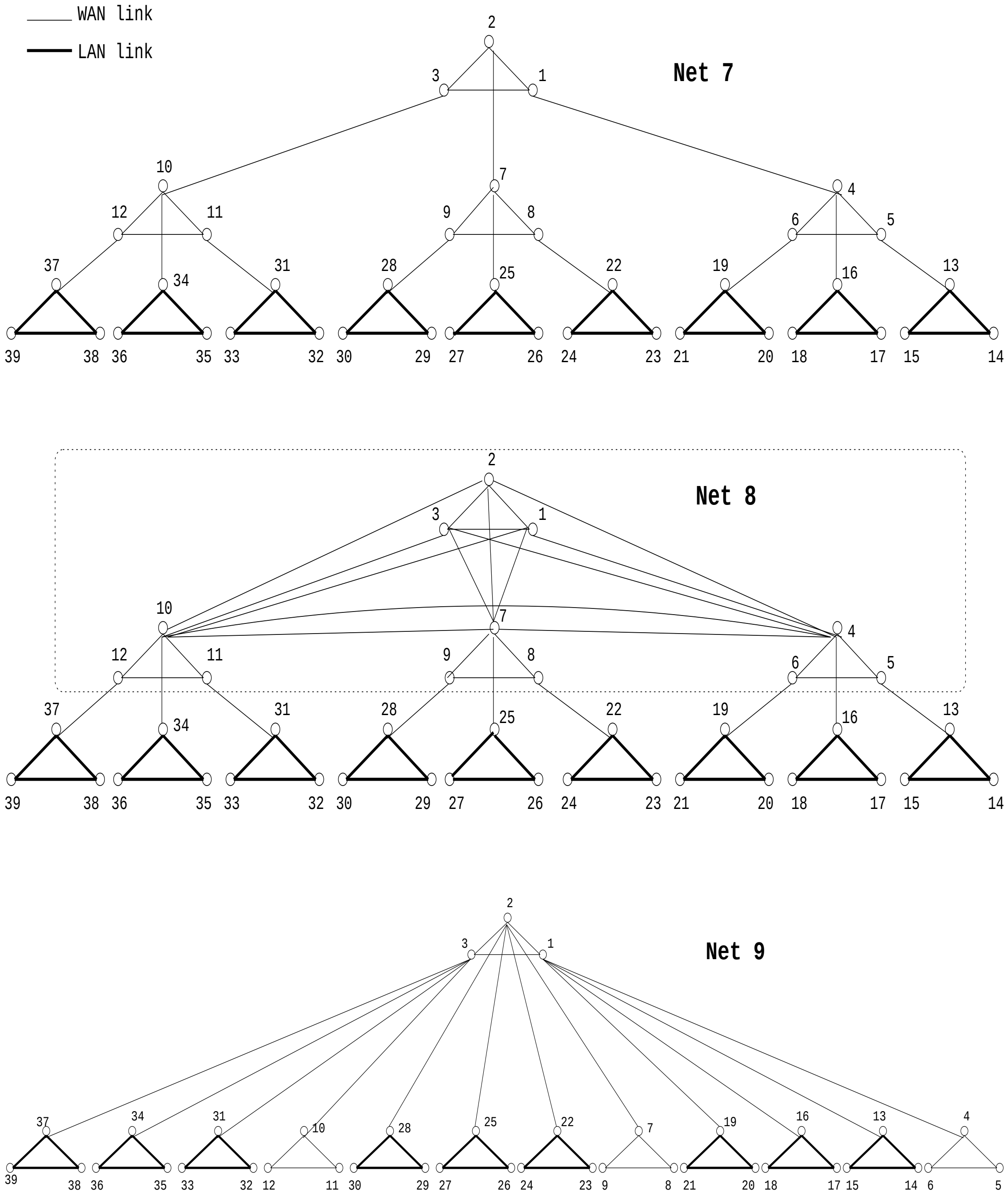

Figure 6.17: Different network topologies for 39 replicas

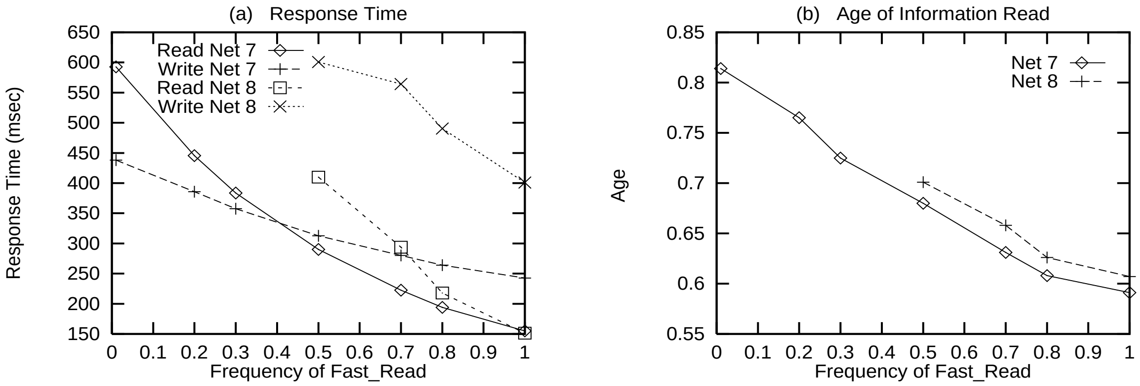

Figure 6.18: Varying read mix, comparing Net 7 and Net 8, writes are synchronous

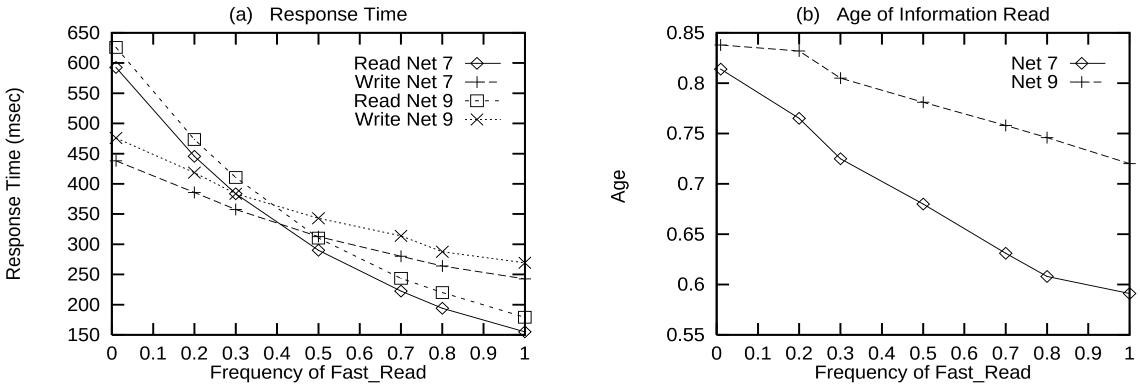

Figure 6.19: Varying read mix, comparing Net 7 and Net 9, writes are synchronous

Similarly, Net 7 and Net 8 have shown worse performance than Net 1 when updates are asynchronous and the read mix is varied, as shown in Figure 6.20 and 6.21.

Therefore, it can be concluded that for a large number of replicas, the hierarchy should be designed such that the number of nodes in the top cluster is not too large in order to reduce the overhead of quorum assembly. Also, for better performance, the nodes in the top cluster should be freed from extra load resulting from asynchronous operations. This can be achieved by increasing the number of levels in the hierarchy.

In general, it is observed that with 39 nodes, the limitations placed by synchronous operations are higher than with 12 nodes and the maximum achieved throughput dropped to less than one-third. Further, the age returned by Fast_Read is generally higher with larger number of replicas since updates take longer to reach all nodes.

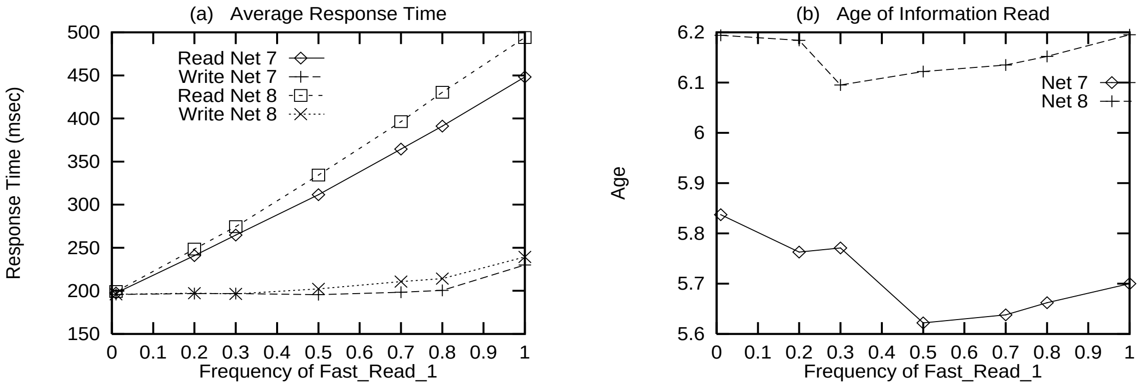

Figure 6.20: Varying read mix, comparing Net 7 and Net 8, writes are asynchronous

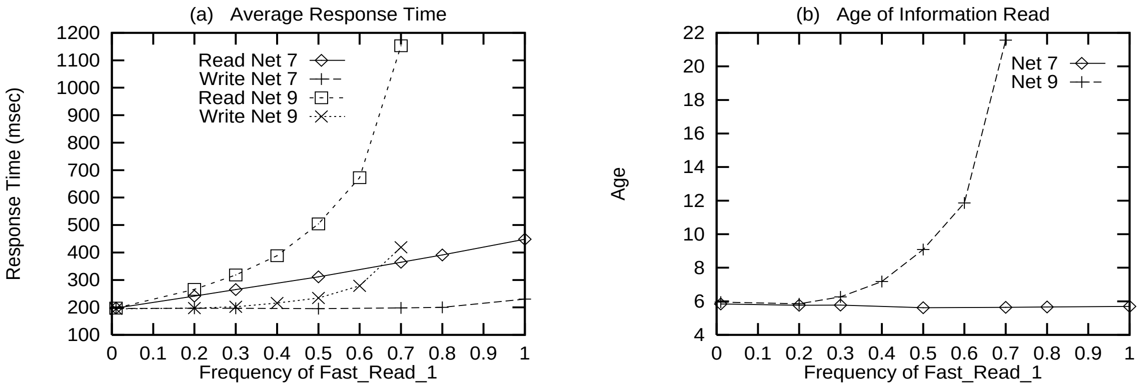

Figure 6.21: Varying read mix, comparing Net 7 and Net 9, writes are asynchronous

6.6 Summary

A detailed simulation study was performed to evaluate the benefits and losses resulting from using synchronous versus asynchronous operations under different system configurations and load mixes. It has been shown that in internetworks, synchronous operations degrade the performance and limit the throughput tremendously. This is mainly due to acquiring and maintaining locks across long distance links which increases the data contention as well as the processing and communication overhead. The imposed limitations are severe with high loads, slow links and high update rates, especially when the number of nodes is large. On the other hand, asynchronous operations have shown much better performance and can cope with heavily loaded systems with a little loss in reading some out-of-date information.

The obtained results revealed that Opt_Write offers better response time, coverage time and local age than Slow_Write while it suffers an increase in the overall and abort age. For applications that have high update rates, Fast_Write is more suitable than Slow_Write and Opt_Write as long as the application can reconcile conflicting updates, since it offers much better performance and throughput and can cope better with heavily loaded systems and networks.

The results obtained about read operations have suggested that if the applications can accept observing some stale information then they should increase their frequency of Fast_Read especially under heavy load conditions, as this offers better performance as well as lower age. Further, Fast_Read operations performed at higher level of the hierarchy have shown to return less stale information with some increase in the response time.

The performance of the protocol has been evaluated on several network topologies that reflect a hierarchical structure. It has been shown that a symmetric hierarchical structure, where the load is balanced among the nodes yields the best results and that distributing the load evenly among the nodes in the top cluster is crucial for synchronous operations. Further, it is suggested to keep the number of nodes in the top cluster small, especially when the number of replicas is large. This is to reduce the overhead of quorum assembly while using synchronous operations, and to reduce the communication and propagation overhead on those nodes while using asynchronous operations. Also, it has been shown that the performance is very sensitive to the number of levels in the hierarchy. If it is too large, the time updates reach all replicas and the staleness of the data increase. If it is too small, nodes in the upper levels of the hierarchy suffer from a high overhead and may become a bottleneck when the number of replicas is large.

For completeness, the performance was evaluated over a network that is more or less fully connected and all nodes are in one logical cluster. Applying synchronous operations to such a network represents the performance of traditional protocols such as the majority quorum consensus protocol. It has been shown that in internetworks, assembling quorums from all nodes in the system degrades the performance immensely and might not be feasible, even for moderate loads, due to the high overhead generated and data contention.

Chapter 7

Comparison with TSAE Protocol

This chapter presents a simulation study conducted to compare the performance of HARP to another weak consistency replication protocol, the *Time Stamped Anti Entropy* (TSAE) [Golding 92c] (see Section 2.2.4). TSAE has been chosen for comparison since it is one of the most recent and widely used weak consistency replication algorithms for wide area services. The obtained results show that in internetworks, HARP results in considerable performance improvement over TSAE despite the extra overhead generated from executing HARP for every message rather than periodically. Further, it is shown that using localised communication improves performance tremendously in internetworks. In Section 7.1, a brief description of the TSAE protocol is given. Section 7.2 describes the simulation model and Section 7.3 presents the obtained results and analysis.

7.1 TSAE Protocol Description

Inspired by Xerox Clearinghouse's anti-entropy protocol [Oppen 83], TSAE provides a reliable, eventual delivery of messages. In TSAE, each replica maintains three data structures: a *message log*, a *timestamp vector* and an *acknowledgement matrix*. The *message log* contains messages received by a node. Messages are entered into the log on receipt, and removed when all other replicas have received them. The messages are stamped with the identity of the replica that initiated the message and a timestamp. The *summary timestamp vector* (STV) records what updates a replica has received and is similar to VV in HARP. The vector holds one timestamp for every replica. Replica i records a timestamp k for replica j when i has received all update messages originating at j up to message k. The *acknowledgement timestamp matrix* (ATM) records what messages have been acknowledged by other replicas. It is similar to $AckM$ of HARP and is used to purge the log.

From time to time, a node i selects a partner node j and starts an *anti-entropy session*. A session begins with the two processes allocating a session timestamp, then exchanging their STV and ATM. Each process determines if it has messages the other has perhaps not yet received, by observing that some of its STV entries are greater than the corresponding ones of its partner. These messages are retrieved from the log and sent to the other process using a reliable stream protocol. The session ends with an exchange of acknowledgement messages. If any step of the exchange fails, either process can abort the session. At the end of a successful session, both replicas have the same continuous sequence of messages sent by each replica. Nodes i and j set their STV and ATM to the element-wise maximum of their own STV and ATM and the ones received from the other process.

In [Golding 92c], several policies for partner selection have been suggested. *Random* selection is the simplest one, where a node selects a partner randomly among all nodes with equal probability. *Distance-biased* partner selection weights the chance of randomly selecting a partner based on its distance. *Oldest-biased* partner selection selects a partner with a probability proportional to the age of its entry in the summary vector. *Oldest-first* selects the node with the oldest value in the summary vector. The performance implications of these different policies have been examined in [Golding 92c]. The results have shown that *random* and *distance-biased* policies give essentially identical performance. *Oldest-biased* provide slightly better performance while *Oldest-first* was worse than the others regarding message propagation time. *Random* policy has been adopted for the *Refdbms* system as it is simple and performed quite well. However, it should be noticed that these results were built on a model that considers a fully connected network, which is not a realistic assumption for a wide area network. Further, network delays were not considered in the model, which are believed to have a great impact on performance.

7.2 The Simulation Model

The same model considered for simulating HARP was adopted for simulating TSAE with some variations to cope with the differences. The only operations considered by TSAE are Fast_Write and Fast_Read which are executed at the origin node. The **algorithm overhead**; that is, the overhead of executing a TSAE session is modelled by three components:

init_tsae_ovhd which is the time the originator of a TSAE session takes to start up a session. This overhead accounts for maintaining and responding to timers, selecting a partner and preparing the message to send to the selected partner.

phs1_tsae_ovhd which is the time taken by either the originator or the partner to perform phase one of TSAE. It accounts for comparing summary vectors, deciding which messages the other is missing and extracting them from the log.

phs2_tsae_ovhd which is the time taken by either the originator or the partner to perform phase two of TSAE. This overhead accounts for updating state information and handling acknowledgements.

A TSAE session is triggered at each node periodically, every *tsae_per* seconds which is exponentially distributed. A node (*originator*) starts a session by executing the processing overhead *init_tsae_ovhd*, selects a partner and sends it an init_tsae message including its STV. When the *partner* receives init_tsae, it sends the *originator* a status message including its STV. Then, it executes the processing overhead of phase one (*phs1_tsae_ovhd*), compares its STV with the *originator*'s STV element by element, extracts the messages that the *originator* is missing and sends them to him. When the *originator* receives the *partner*'s status message, it performs the same steps: executes the processing overhead of phase one (*phs1_tsae_ovhd*), compares the STV's, extracts missing messages and sends them to the *partner*. When the *originator* or *partner* receives the missing messages, it executes the processing overhead of phase two (*phs2_tsae_ovhd*) and for every received message it executes the update, increments its version and updates its STV. Modelling the log is ignored and the system is modelled during normal operation; that is, in the no failure case.

Since a node can participate in more than one session at a time (either as an originator or as a partner), and since status messages are received out-of-date due to network delays, then duplicate messages can be sent from one node to another. When a node receives a duplicate message, it discards it, however, sending duplicate messages consumes resources while sent over the network.

Execution of local requests is given higher priority at the processing and I/O devices than updates received through TSAE sessions. This is to avoid local requests facing long queueing delays when a session results in scheduling a large batch of updates.

The **network model** is the same as in HARP. The physical network considered is Net 1 as shown in Figure 6.1. Since TSAE sends messages in batches, it is essential to model the communication overhead taking into consideration the message length. For simplicity, it is assumed that all update messages are of the same size and are treated as one unit (typical value considered is 1 Kbyte).

Running the network protocols and the operating system consists of some functions that their overhead depends on the size of the messages; such as fragmentation of messages into packets and their assembly, checksum verification, moving data into memory and so on. While other functions are independent of the size of the messages; such as processing message header, searching for local state information of the connection, flow control and multiplexing, routeing and host addressing, handling interrupts, managing timers and schedulers and so on. Therefore, two new parameters are added: *const_cpu_msg* which is the portion of *cpu_msg* that does not depend on the message size and *msg_per_pckt* which is the number of messages that fit into one packet. The processing overhead associated with sending m updates in one batch becomes: $no\_pckts * const\_cpu\_msg + (cpu\_msg - const\_cpu\_msg) * m$ where $no\_pckts = \lceil \frac{m}{msg\_per\_pckt} \rceil$. The ratio *cpu_msg*:*const_cpu_msg* is taken in accordance to [Clark 89]. As for the network delays, the propagation delay (*lan_propg_del* for LAN and *wan_propg_del* for WAN) is independent of the message size; while the transmission delay (*lan_del* - *lan_propg_del* for LAN and *wan_del* - *wan_propg_del* for WAN) is linearly proportional to the message size.

Assuming a message size of 1 Kbyte, the number of messages that fit into one packet is taken to be 1.5, considering the typical packet size in the Ethernet. The system parameters, their description and default values used are listed in Table 7.1.

parameter	Description	Default value
n	Number of nodes in the network	12
inter_arr_time	Inter arrival time	600 msec
$freq_r$	Frequency of read requests	0.7
cpu_req	Processing requirements of a request (exponential)	50 msec
io_req	I/O requirements of a request (exponential)	60 msec
init_tsae_ovhd	Time originator takes to start up a TSAE session	1 msec
phs1_tsae_ovhd	Overhead of phase one of TSAE session	2 msec
phs2_tsae_ovhd	Overhead of phase two of TSAE session	1 msec
tsae_per	The period by which TSAE sessions are triggered (exponential)	5 sec
cpu_msg	CPU overhead to send/receive a message	2.0 msec
const_cpu_msg	Portion of *cpu_msg* that does not depend on message size	0.7 msec
lan_del	Network delay over LAN (uniform)	0.02 msec
lan_propg_del	Portion of *lan_del* spent in propagation	0.01 msec
wan_del	Network delay over WAN (exponential)	20.0 msec
wan_propg_del	Portion of *wan_del* spent in propagation	5 msec
msg_per_pckt	Number of messages that fit into one packet	1.5
$power\_cpu_i$	Speed of CPU at node i	1
$power\_io_i$	Speed of I/O at node i	1
$power\_transc_i$	Speed of transceiver i	1
batch_length	Length of a single simulation batch	2000 job
num_batch	Number of batches for which simulation runs	3
warmup_per	Number of jobs after which statistics are reset	200 job

Table 7.1: System parameters of TSAE

The **output metrics** considered are the same as in HARP. Two additional metrics are considered: *session time* which is the time it takes a node to complete a TSAE session. This metric spans the time from which the node initiates the session until all updates are received from the partner and processed. Also, the amount of duplicate messages sent, *duplication*, is measured which is computed as the ratio between the total number of duplicate messages sent to the total number of non-duplicate messages sent.

7.3 Experiments and Results

In the following experiments, a number of parameters is varied to see how they affect the performance of the TSAE protocol and the results are compared with their equivalent in

HARP. $fast_w$ and $fast_{r0}$ are set to 1 in HARP. Two variants of selecting a partner for TSAE are considered:

1. TSAE_R: where a node chooses a partner *randomly* from all nodes in the network.
2. TSAE_L: where a node chooses a partner from the *local* neighbourhood. The logical hierarchy of HARP is used to implement TSAE_L. Simply, a node chooses a partner randomly from its own correspondents in the logical hierarchy, i.e. a neighbour, the parent or a child. Hence, it uses the principle of localised communication.

7.3.1 Varying the Activation Period

This experiment explores the effect of varying the period at which the TSAE protocol is activated, *tsae_per*. Parameters are set to their default values and *tsae_per* is varied from 0.5 to 8 seconds.

As shown in Figure 7.1(a), the load on the transceivers is the least in HARP, followed by TSAE_L then TSAE_R. The same applies to the load on the processors. Resource utilisation of TSAE_L increases slowly as *tsae_per* decreases, while that of TSAE_R increases sharply and resources are saturated for *tsae_per* < 2 seconds. The increase is mainly due to communication overhead: as *tsae_per* decreases, more concurrent sessions take place and more duplicates are sent (as originals are on the way but STV is not updated yet) which increases the load on the resources. Further, both TSAE involve sending status information in every session -which is not present at HARP- hence consuming more resources. As seen in Figure 7.1(b), TSAE_R involves more duplicates than TSAE_L because TSAE_R contacts more distant partners, hence state information is more out-of-date. Also, it consumes more resources from intermediate nodes and links while TSAE_L uses only local communication.

The time by which a node completes a session in TSAE_L is less than TSAE_R, as shown in Figure 7.1(c), since resources are less loaded. Generally, the time decreases as *tsae_per* decreases since there are less updates to be sent and executed at a session. For TSAE_R, this applies only for *tsae_per* > 3, then it increases sharply due to overhead on the resources. While in TSAE_L, it increases slowly only for *tsae_per* < 1.

From Figure 7.1(d), we see that the response time of reads and writes is nearly the same for all of them for high values of *tsae_per* with HARP being the best. The response of both TSAE decreases as *tsae_per* decreases, since there are less updates to be executed per session, hence, local requests are faced with less contention. The response of TSAE_R increases sharply at *tsae_per* < 3, while TSAE_L starts increasing slightly at *tsae_per* < 1.

The reach time of TSAE_L is higher than TSAE_R for large values of *tsae_per*, as shown in Figure 7.1(e). The reason is that in TSAE_L, updates needs more sessions to reach nodes in the other extreme of the tree while TSAE_R contacts distant replicas directly. The reach

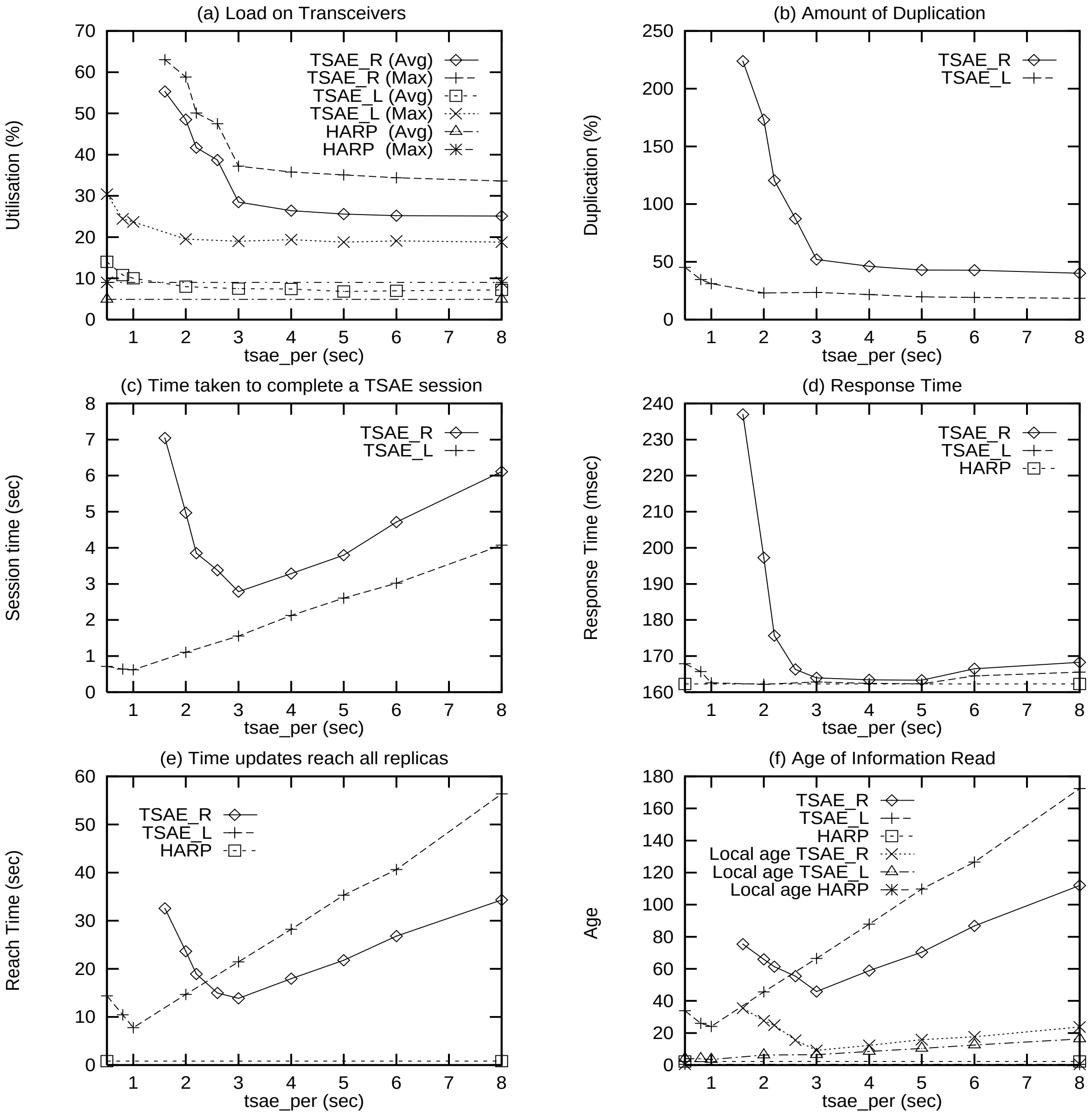

Figure 7.1: Varying the activation period, *tsae_per*

time of both TSAE_L and TSAE_R decreases as *tsae_per* decreases, but TSAE_L's decrease is steeper and becomes better than TSAE_R for *tsae_per* < 3. It should be noticed that, although TSAE_L incurs higher overhead for running the protocols for *tsae_per* < 3, it generates less communication overhead than TSAE_R and the overall system overhead of TSAE_L is lower. It is observed that the reach time of both TSAE is much higher than HARP. Partly, because of the overhead resulting from sending duplicates. Also, TSAE relies on periodic rather than immediate propagation. The age follows the same curve as the reach time as illustrated in Figure 7.1(f). Further, since TSAE_L uses local communication, the coverage time of the first few nodes was found to be better than TSAE_R and consequently the local age is better as shown in Figure 7.1(f).

In conclusion, TSAE is very sensitive to the period at which the protocol is activated. If it is set too high, it takes longer for updates to reach other replicas, which increases the reach time and age. If it is set too low, then this involves large overheads and the performance degrades. TSAE_L performs better than TSAE_R in terms of coverage, local age, resource consumption, duplication and overhead since it involves less communication. TSAE_R offers better age and reach time only for large values of *tsae_per*. For low values of *tsae_per*, it is worse than TSAE_L in all metrics. Finally, it is observed that HARP outperforms both TSAE_R and TSAE_L. In the remaining experiments, *tsae_per* is set to 3 seconds for TSAE_R, and to 1 second for TSAE_L since this yields the best performance.

7.3.2 Varying the Frequency of Reads versus Writes

This experiment examines the effect of varying the frequency of reads versus writes ($freq_r$). $freq_r$ is varied from 0.5 to 0.95 and the remaining parameters are set to their default values.

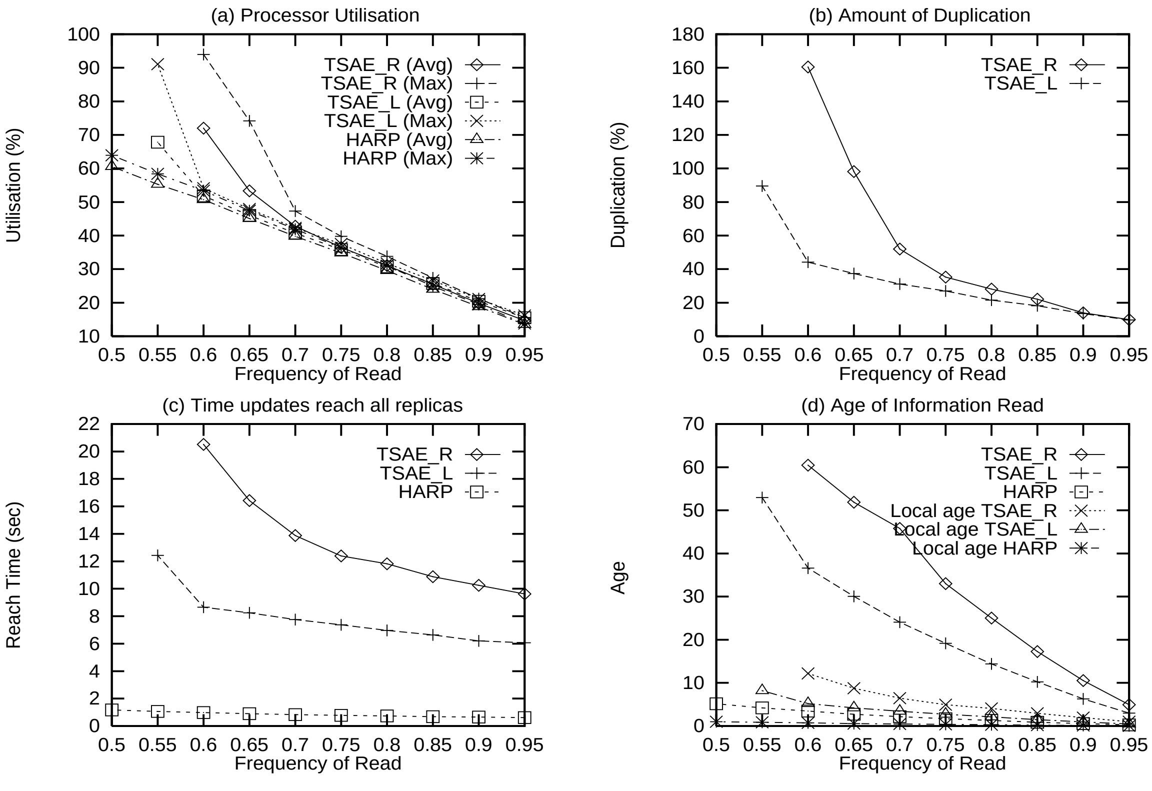

Figure 7.2: Varying the frequency of reads versus writes

As shown in Figure 7.2(a), the resource utilisation increases for all protocols as more writes are present since they need to be propagated and executed. The increase is small for HARP but it is much higher for both TSAE due to the overhead placed on the resources for sending duplicates and status messages. The amount of duplication is small for low frequency of writes, then it increases sharply for both TSAE as illustrated in Figure 7.2(b). The duplication overhead causes instability of the system for $freq_r < 0.7$ and 0.6 for TSAE_R and TSAE_L respectively.

The reach time is the best for HARP, followed by TSAE_L, and both are better than TSAE_R as shown in Figure 7.2(c). The same applies to the age and the local age of the information read as illustrated in Figure 7.2(d). The time to complete a session in TSAE_L is found to be better than TSAE_R since the former uses local communication and produces less overhead. The response time is the same for the three protocols for high values of $freq_r$, since the load on the resources is nearly the same. However, TSAE experiences higher response for low values of $freq_r$.

Therefore, it is observed that in general, as $freq_r$ decreases performance degrades. However, HARP copes with higher frequency of writes better than TSAE as it does not generate duplicate messages and does not rely on periodic propagation. TSAE_L performs better than TSAE_R since it produces less overhead and duplicates due to localised communication.

7.3.3 Varying the Arrival Rate

This experiment examines the sensitivity of performance to the load intensity. $freq_r$ is set to 0.8, *inter_arr_time* is varied from 200 to 1000 msec and the remaining parameters are at their default values.

As shown in Figure 7.3, the performance degrades as the arrival rate increases. The system is saturated at arrival rates 5, 4 and 2.8 request/sec for HARP, TSAE_L and TSAE_R respectively. Resource utilisation is nearly the same for the three protocols at low loads then it increases, being the highest with TSAE_R due to the communication overhead, mainly because of duplicate messages. Since TSAE_L produces less duplicate than TSAE_R, it performs better for all metrics as shown, while HARP has achieved better performance than both of them.

7.3.4 Varying the Number of Messages per Packet

This experiment discusses the effect of varying the number of messages that can fit into one packet (*msg_per_pckt*) on the performance. *inter_arr_time* is set to 400 msec, $freq_r$ to 0.8, *msg_per_pckt* is varied from 1.5 to 15 and the remaining parameters are at their default values.

Since TSAE sends messages in batches, then increasing *msg_per_pckt* results in improving

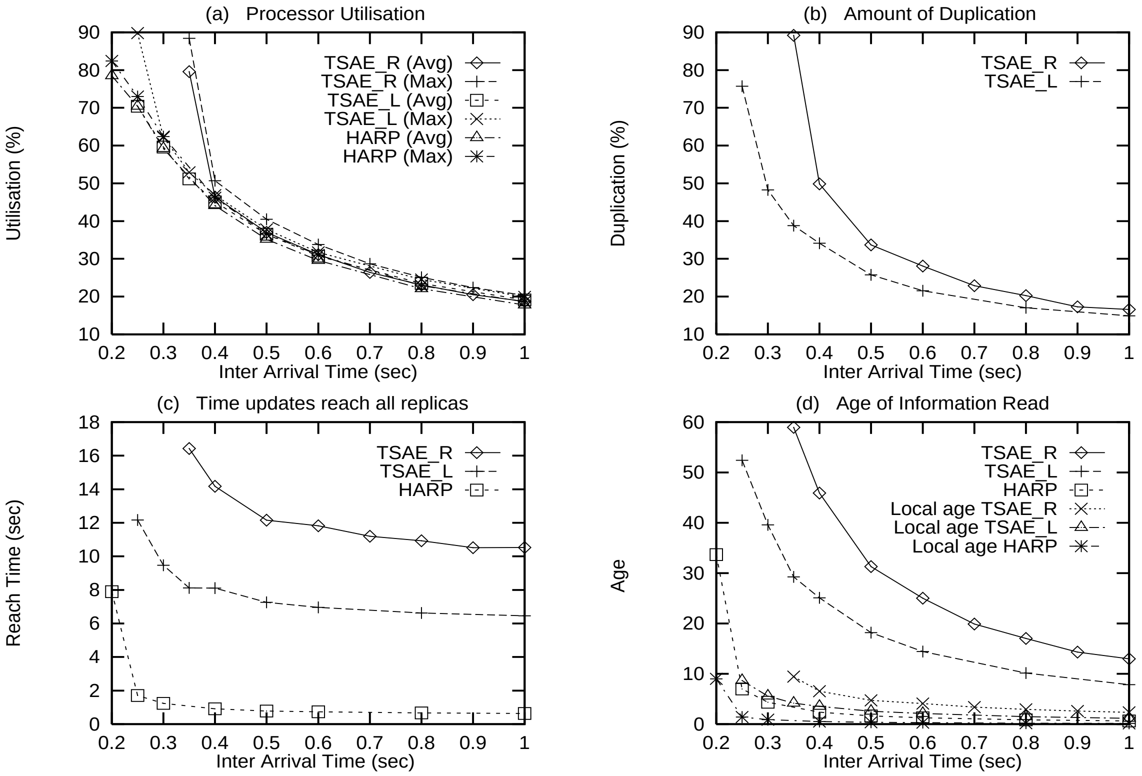

Figure 7.3: Varying the arrival rate

the processor utilisation, as shown in Figure 7.4(a), due to the reduction in communication overhead. TSAE_L processor utilisation decreases and becomes lower than that of HARP for $msg\_per\_pckt > 5$. As for TSAE_R, it decreases, but is still higher than both HARP and TSAE_L due to high duplication. As shown in Figure 7.4(b), the amount of duplication decreases as *msg_per_pckt* increases, however, it is still at a high value. Due to the reduction in the processing load, the response time decreases as shown in Figure 7.4(c) for both TSAE and that of TSAE_L becomes eventually better than HARP for large values of *msg_per_pckt*. There is a slight decrease in the reach time of both TSAE due to lower communication overhead. However, they are still higher than that of HARP, since HARP does not rely on periodic propagation and does not generate duplicate messages. Consequently, the age and the local age returned by read operations experience a slight reduction, as shown in Figure 7.4(d). But they are still higher than HARP which returns an age of 2.35 and a local age of 0.5. So, there is a tradeoff between the load on the processors and the response time, and between the reach time, the age and the load on the communication links.

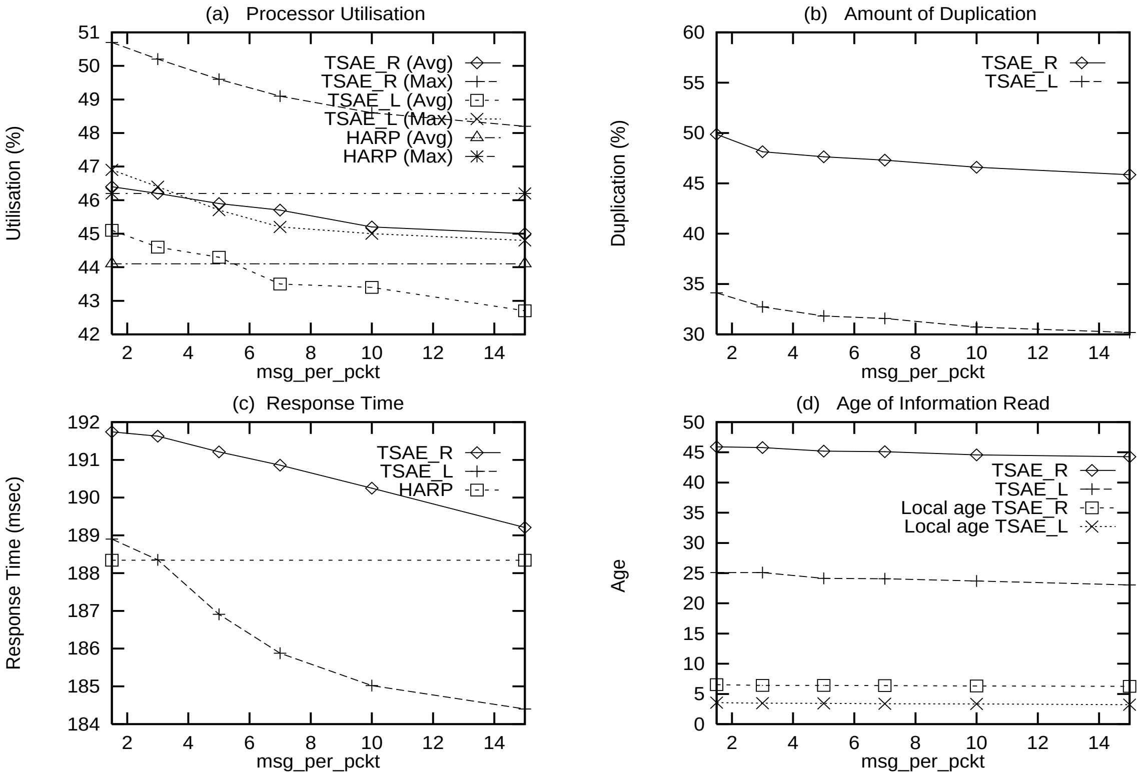

Figure 7.4: Varying the number of messages per packet

7.3.5 Varying the Communication Processing Overhead

This experiment shows the effect of varying the communication processing overhead (*cpu_msg*) on performance. *inter_arr_time* is set to 400 msec, $freq_r$ to 0.8, *msg_per_pckt* to 10, *cpu_msg* is varied from 0 to 10 msec and the remaining parameters are at their default values.

As shown in Figure 7.5(a), in TSAE_L the load on the processors is lower than that of HARP for low values of *cpu_msg*. For high values of *cpu_msg*, since the message delay increases, the state information is more out-of-date. Consequently, the amount of duplication increases, as shown in Figure 7.5(b), which increases the communication overhead and eventually, the load on the processors bypasses that of HARP. TSAE_R's processing load is lower than both TSAE_L and HARP for the ideal case of *cpu_msg*=0, when the only overhead is that of running the algorithm. However, for *cpu_msg* > 0, the load of TSAE_R increases sharply and the system is saturated at *cpu_msg* > 7 msec.

Although for low values of *cpu_msg* TSAE_L has lower processing overhead than HARP, it consumes more bandwidth due to duplicate messages. Also, it relies on periodic updates. Therefore, the reach time is higher than HARP, as shown in Figure 7.5(c), and it increases as *cpu_msg* increases. The reach time of TSAE_R is much worse than both TSAE_L and HARP

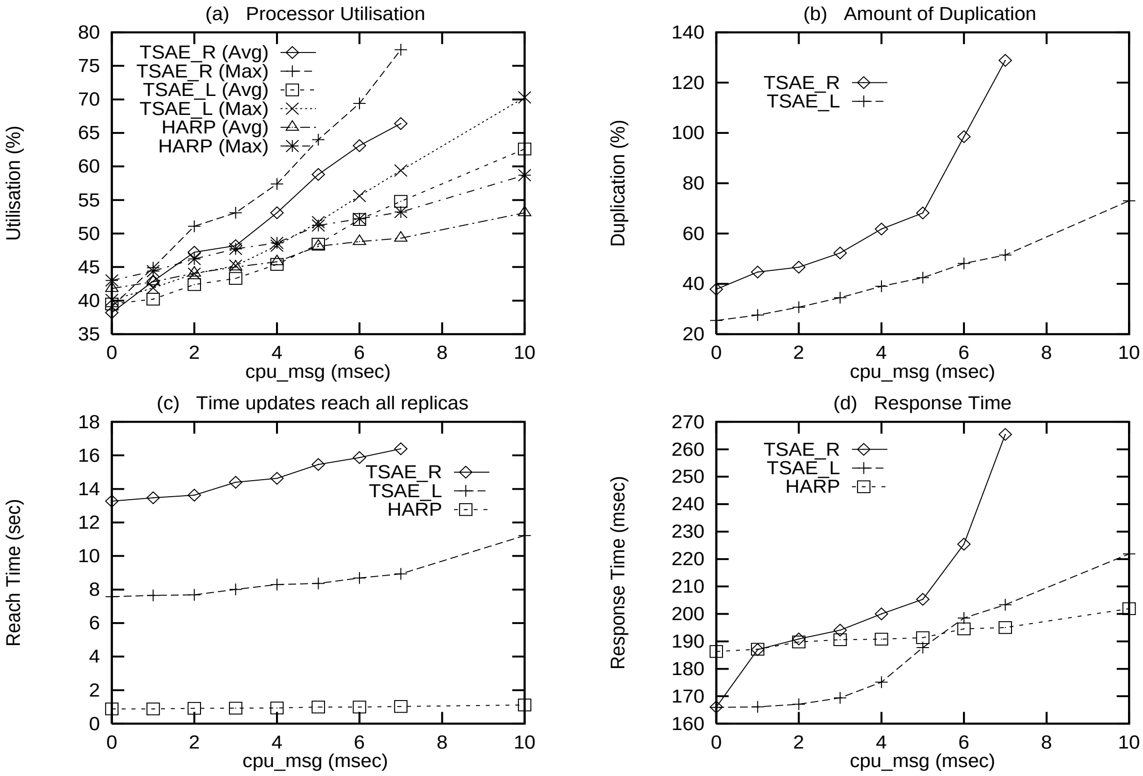

Figure 7.5: Varying the communication processing overhead

as it imposes higher communication overhead. The age returned by reads follows the same curve as the reach time.

The response time of TSAE_L (Figure 7.5(d)) is better than HARP for low values of *cpu_msg*, then it increases and bypasses it for high values due to the increase in the load of the resources. TSAE_R is lower than HARP only when *cpu_msg*=0, and then it increases sharply as shown.

From here it can be deduced that TSAE_L has lower processing communication overhead and better response time than HARP for low values of *cpu_msg*, while HARP consumes less bandwidth and offers better reach time and age. For high values of *cpu_msg*, HARP outperforms TSAE_L in all metrics. As for TSAE_R, it has shown worse performance than both TSAE_L and HARP in general.

7.3.6 Varying the Network Delay

This experiment explores the effect of varying the communication delay (*wan_del*) on performance. *inter_arr_time* is set to 400 msec, $freq_r$ to 0.8, *msg_per_pckt* to 10, *wan_del* is varied from 0 to 40 msec and the remaining parameters are at their default values.

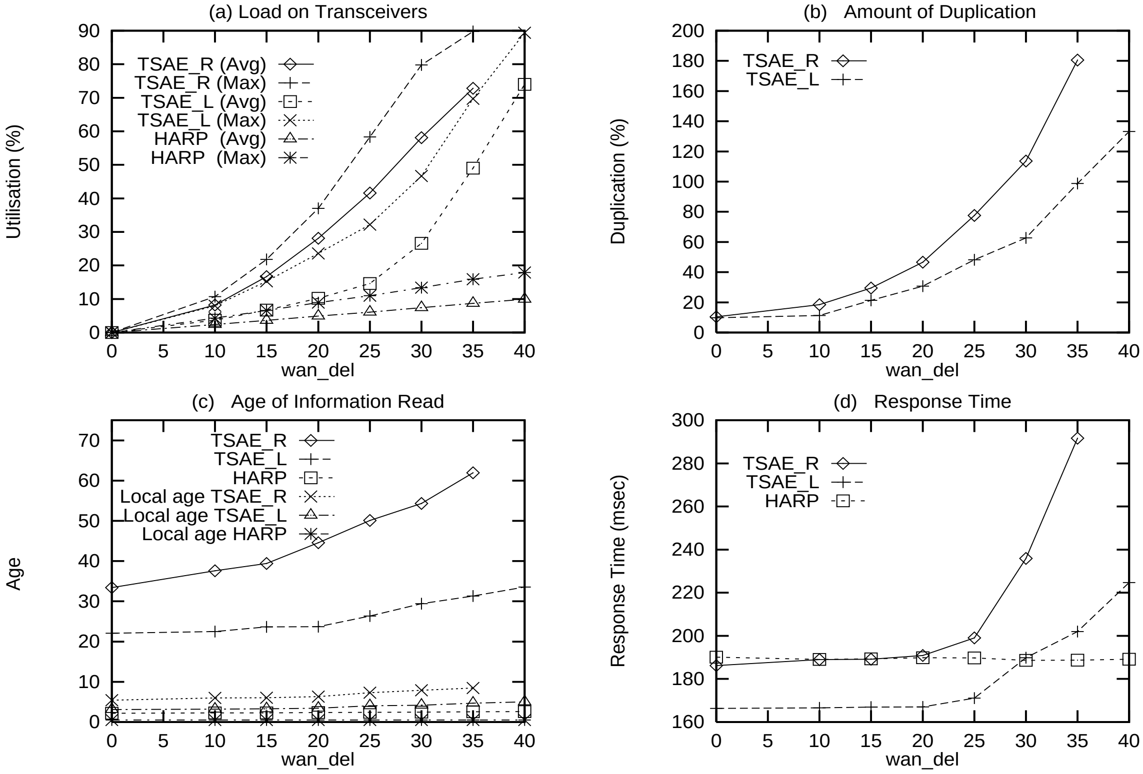

Figure 7.6: Varying the network delay

The traffic on the communication links increases as *wan_del* increases, as shown in Figure 7.6(a). Bandwidth consumption is the least in HARP followed by TSAE_L and is the highest in TSAE_R, especially for high values of *wan_del*. There are two reasons for the increase in the traffic. First, TSAE involves exchanging status information messages in every session which places a load on communication links. Second, TSAE involves exchanging duplicate messages which increases as *wan_del* increases, as shown in Figure 7.6(b), causing more traffic to be generated and finally saturation of the links. Consequently, the reach time in HARP is better than both TSAE with TSAE_R being the worst. The same applies to the age of information read as shown in Figure 7.6(c).

Processor utilisation of TSAE_L is lower than HARP for low values of *wan_del*, and then it increases sharply; mainly due to the communication overhead for processing duplicate messages. Consequently, the response time of TSAE_L is better than HARP for low values of *wan_del*, then it increases and bypasses it, as shown in Figure 7.6(d). The increase in the response time of TSAE_R is much larger for high values of network delay.

From this experiment, it can be concluded that the three protocols are sensitive to the increase in the network delay, with HARP being the least sensitive. Both TSAE have shown severe degradation in performance with high values of *wan_del* and the degradation in TSAE_R is much higher due to contacting distant replicas.

7.4 Discussion

A variation of TSAE, suggested by its author, is to combine it with an unreliable multicast to propagate updates rapidly. When a node originates an update, it multicasts it to other nodes. Some of them might not receive the multicast, but they will receive the message later when they conduct an anti-entropy session with another node that has received the message. Although this suggestion seems to speed up dissemination of messages, it incurs a large amount of duplication. This is because the anti-entropy session considers only summary vectors when deciding whether to transmit a message to a partner, regardless of whether the partner has already received the message by multicast. This means that if the multicast is reliable, every message will be sent at least twice. Since the conducted experiments have shown that a large amount of duplication is generated even without adding the multicast, it is believed that adding the multicast will cause a high contention for the resources, and therefore this option was not considered in the experiments.

Although the log is not modelled in the simulation, the reach time gives an indication of its size; since a message is deleted from the log when it is received by all replicas. As shown, HARP has achieved a much better reach time than TSAE. Therefore, entries in the log are purged more quickly and the log size is expected to be smaller.

It was observed that TSAE can incur lower overhead than HARP since it relies on periodic updates. A simple variation can be added to HARP such that it sends messages in batches rather than individually. This requires a slight modification to the protocol such that it propagates messages either:

- when a certain number of messages is generated or received, then it propagates them in one aggregate message using the HARP propagation protocol. Or,
- periodically, each node gathers the updates generated and received into one message and propagates it using the HARP propagation protocol.

In both cases, the recipient de-assembles the message and processes each message as if it was received individually as before. This variation gains the advantage of sending messages in batches, hence, it has lower overhead in running the propagation protocol and lower communication overhead. So, it is asymptotically equivalent to TSAE in terms of protocol and communication processing overhead. Further, it retains the advantages of not sending duplicate messages and not exchanging status messages in propagation. Therefore, this variant of HARP is expected to outperform TSAE in all metrics. However, as was shown for TSAE, the periodic updates result in degradation in the reach time and the age of the information read. Therefore, in choosing between HARP and its variation with periodic updates, there is a tradeoff between reach time and age versus communication and processing overhead.

7.5 Summary

This chapter has presented a comparison between HARP and two variants of the TSAE protocol. The results have revealed that, TSAE_L has three disadvantages over HARP. First, it exchanges status messages in every session which places an extra load on the resources. Second, it involves exchanging duplicate messages which increases the traffic and degrades performance. Finally, it relies on periodic propagation which slows down the speed by which updates reach all replicas. On the other hand, TSAE_L has two advantages over HARP. First, it has lower overhead of running the protocol. Second, it sends messages in batches so communication overhead is lower.

From the conducted experiments, it has been observed that HARP constantly offers better reach time, coverage time, age and bandwidth consumption. TSAE_L offers better response time and processing communication overhead only if the number of duplicate messages generated is small. It was shown that this applies only when the number of messages that can fit in a single packet is large, the processing communication overhead per message is low and the network delay is small. However, since these parameters are not under the control of the system manager (not tunable parameters), it is believed that HARP performs better than TSAE_L under most system configurations.

TSAE_R has shown worse performance than TSAE_L because TSAE_R performs anti-entropy sessions with distant replicas. Therefore, status information is more out-of-date due to network delays. Consequently, more duplicate messages are exchanged. Further, while contacting distant replicas, TSAE_R places more load on intermediate nodes and links which increases the overhead of the system. Therefore, it is concluded that using localised communication has a large impact on performance in internetworks.

Chapter 8

An Alternative Hierarchical Propagation Protocol (HPP)

8.1 Introduction

In Chapter 4, storage space optimisations within HARP have been discussed, where the size of the data structures has been reduced significantly by exploiting locality of propagation. However, HARP still requires the VV structure to be propagated periodically and the VV and LL structures to be exchanged during the reconfiguration protocols and while handling failures and partitions. Since the size of these structures is $O(n)$, this may impose a high communication overhead for very large scale systems. An alternative hierarchical propagation protocol (HPP) is proposed [Adly 95c] which overcomes this drawback.

The main difference between HPP and HARP is that HPP avoids the transmission of global information. Similar to the idea of Compact Vectors, HPP uses aggregate state vectors recording the messages a node has received from its correspondents only rather than from all nodes. In HPP, each node maintains state vectors describing not only its own state, but also the state of its correspondents. These vectors are of a size proportional to the number of correspondents and they provide enough information for a node to determine missing messages. Reorganisation protocols and methods for handling failures are suggested which rely on these local state vectors and avoid exchanging global information. Therefore, HPP is more scalable. However, it is less reliable than HARP as it can tolerate only special patterns of failure. Further, its support for ordering delivery is weaker than HARP. Therefore, it is suitable for applications that require no order of delivery or applications with simple ordering requirements such as a bulletin board service.

This chapter presents the proposed protocol HPP. As in HARP, nodes in HPP are organised in a logical hierarchy. For ease of presentation, the protocol is described for a simplified hierarchy, where a node communicates only with its parent and children, but not with neighbours. This could be regarded as a hierarchy where the size of each cluster is one, although the protocol can be extended to support a hierarchical structure with clusters of size greater than one.

The organisation of the chapter is as follows. Section 8.2 describes the operation of the propagation algorithm during normal conditions and the data structures used. Then, Section 8.3 explains how network reorganisation and restructuring take place. Next, Section 8.4 describes operation in failure mode. Section 8.5 reviews the strengths and limitations of the protocol and compares it to HARP.

8.2 Propagation During Normal Operation

Figure 8.1 shows an example of the hierarchical structure considered. The notation $P(i)$ is used to denote the parent node of node i, and $Ch_j(i)$ to denote the j^{th} child of node i. Without loss of generality, it is assumed that each node has c children.

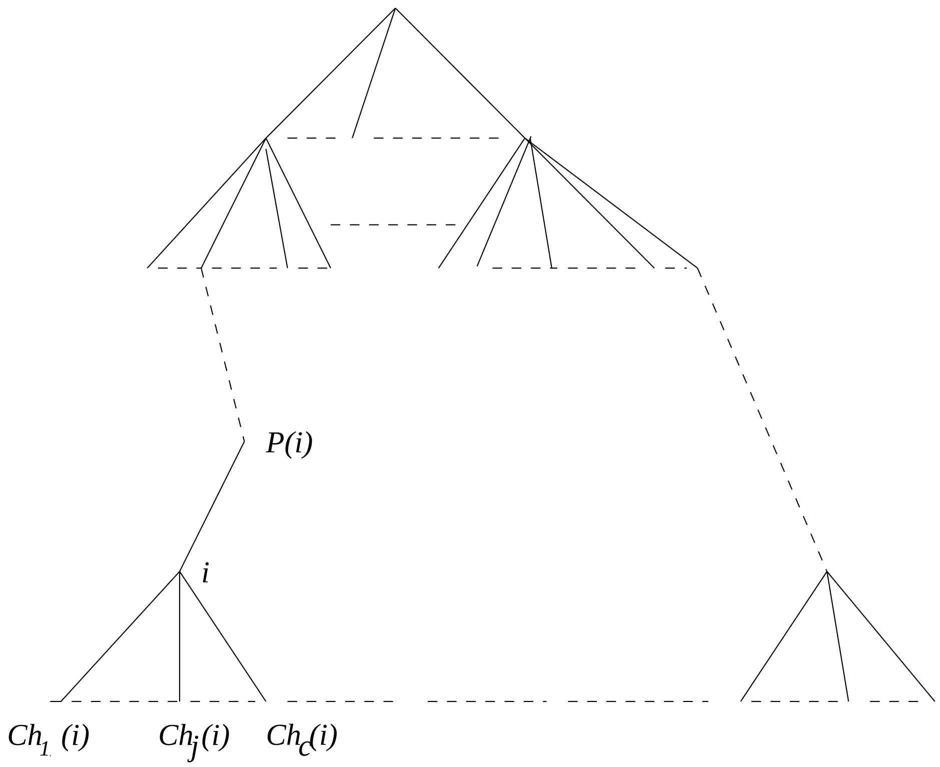

Figure 8.1: A multilevel hierarchy of nodes with HPP

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 for applications like bulletin board. 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_ack1* and *last_msg* messages described in Section 8.3.

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, sends the message to all correspondents *except* the one the message comes 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.

As in HARP, every message m keeps the identity of the originator *m_org* (i.e. the node that creates the message) and a corresponding sequence number *m_seq*. The combination ⟨*m_org, m_seq*⟩ 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 along with each message (1 bit per correspondent) to keep track of acknowledgements received from the correspondents.

Each node i keeps the following state vectors:

- V_i, a vector to keep track of the messages node i has generated or received from its own correspondents.

  $V_i = (U, D_1, D_2, ..., D_c, L, W, W')$, 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 $Ch_j(i)$
  $L =$ the number of the last *local* message generated by node i itself
  $W =$ sequence number of the downward stream i.e. a counter to keep track of the number of messages sent by i to its child nodes.
  $W' =$ sequence number of the upward stream, i.e. another counter for messages sent by i to its parent.

As before, the notation $V_i.x$ is used 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.

  $V_i^P = (U, D_1, D_2, ..., D_c, L)$

  The definition of each entry is the same as V_i but with respect to the parent.

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

  $V_i^{Ch} = (L_1, L_2, L_3, ..., L_c)$, where

$V_i^{Ch}.L_j$= the number of the last message generated locally by the child $Ch_j(i)$ and received by i.

- Each node maintains a *Version Vector* VV which keeps track of the last message number received from every other node in the network; the same as in HARP. This vector is local to each node; that is, it is not exchanged between nodes and is just used to detect duplicates.

Therefore, state vector V_i contains information about messages received by node i itself, and V_i^P and V_i^{Ch} 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 and enable i to determine messages that have been missed.

The state vectors V, V^P, V^{Ch} and VV are updated upon the generation or receipt of any message. The details of updating the state vectors are shown in Figure 8.2. 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, it is assumed that node i is the r^{th} child of its parent $P(i)$.

When node i sends messages to correspondent j, j processes these messages in the same order they were sent from i. Messages received out of order are inserted in a queue for later processing. In general, there are $c+1$ such 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. This order is referred to as H_FIFO to distinguish it from the standard FIFO order, and these queues are denoted by H_FIFO queues [1]. Since all nodes observe H_FIFO order while propagating messages, and there is only one unique path in the logical hierarchy for a message between any pair of nodes, then causal ordering is maintained during normal operation. Theorem 9 given in Appendix C formally proves that causal order is achieved during normal operation.

Each node keeps a partial view of the hierarchy including the identities of its own correspondents (i.e., its parent node and all child nodes) and their correspondents. Therefore, each node maintains a $[c+2][c+1]$ matrix $View$, where, row `0` holds the ids of its own correspondents, row `1` gives the ids of its parent's correspondents, and row `2` through `c+1` gives the ids of the correspondents of its children (i.e., child nodes 1 through c, respectively). In each row, the first entry denotes the id of the parent and entries `1` through `c` are the ids

[1] To support H_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; if $VR[j]+1$ is equal to the tagged value, then the message is received in H_FIFO order.

When node i generates a message m:

```
{
increment V_i.L, V_i.W, V_i.W' and VV_i[i];
set m_prev = 0;
send m to Ch_j(i),    ∀j;       /* children */
send m to P(i) and increment V_i^P.D_r;    /* parent */
}
```

When a node i receives a message m with $uid = \langle m\_org, m\_seq \rangle$ from its parent $P(i)$:

```
{
If m is in H_FIFO order Then
  If m_seq ≤ VV_i[m_org] Then
    discard m;        /* it is a duplicate */
    Else {
      increment V_i.U, V_i.W and VV_i[m_org] ;
      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 Ch_j(i),    ∀j;    /* children */
      send acknowledgement to P(i);
      }
  Else
    insert m in H_FIFO queue corresponding to P(i);
}
```

When node i receives a message m with $uid = \langle m\_org, m\_seq \rangle$ from its j^{th} child $Ch_j(i)$:

```
{
If m is in H_FIFO order Then
  If m_seq ≤ VV_i[m_org] Then
    discard m;        /* it is a duplicate */
    Else {
      increment V_i.D_j, V_i.W, V_i.W' and VV_i[m_org];
      If m_org = Ch_j(i) Then increment V_i^Ch.L_j ;
      set m_prev = j ;
      send m to Ch_k(i), ∀k ≠ j ;
      send m to P(i) and increment V_i^P.D_r ;
      send acknowledgement to Ch_j(i) ;
      }
  Else
    insert m in H_FIFO queue corresponding to Ch_j(i);
}
```

Figure 8.2: Algorithm for updating the state vectors in HPP, upon originating or receiving a message

of the child nodes `1` through `c`, respectively.

Messages are kept in a *log*. A message is inserted into the log when received (in H_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 acknowledgement was received.

The duration of T has to be specific to each site at the discretion of the site manager. It is assumed to be large enough to ensure that correspondents of the node's correspondents have received the message before it is deleted. It is anticipated that a value of anywhere between 1 and 7 days would be reasonable. A summary of the data structures maintained at each node i, their descriptions and their sizes is presented in Table 8.1.

Variable name	Description	Size
$View_i[c+2][c+1]$	Partial view of the tree	$(c+2)(c+1)$
V_i	Vector describing i's state	$c+4$
V_i^P	Vector describing i's record of its parent's state	$c+2$
V_i^{Ch}	i's record of messages originated by its child nodes	c
$VV_i[n]$	Version Vector	n
log_i	Log keeping received messages	variable
c+1 H_FIFO queues	Queues maintained for H_FIFO ordering messages	variable
$VS_i[c+1]$	Vector keeping sequences of sent messages	$c+1$
$VR_i[c+1]$	Vector keeping sequences of received messages	$c+1$

Table 8.1: Summary of data structures in HPP, their descriptions and sizes

8.3 Reorganisation

HPP adopts a different approach than HARP to reorganise the hierarchy such that it avoids exchanging global state information while still guaranteeing that no messages are lost while the switch is taking place. Without comparing global states, a node cannot detach itself from the hierarchy and rejoin as it happens in HARP. The basic idea behind the reconfiguration scheme in HPP is that a node wishing to change its position in the hierarchy has to join the new nodes it wishes to connect to first, then it detaches itself from the old nodes it was affiliated with. That is, for a certain period of time, it is linked to both parties in order to ensure no loss of messages while avoiding exchanging global state. The reorganisation operation that is of interest is the one that allows a node to change its parent node, which would be the same as a `move` operation, and it will be described in detail.

As before, the moving node must either be a leaf node or it should move all its descendants along with it to the new position. The operation is initiated by the moving node which informs the new parent of its intention to join. The new parent starts buffering any new messages for the new child. On receiving a confirmation from the new parent, it leaves the old parent and joins the new parent, buffering any new messages for the new parent. During the join, the new parent and the moving node exchange the messages they have buffered for each other before starting propagating messages to one another. The protocol ensures that no messages are lost even if a failure occurs along the path from the old parent to the new parent while the reorganisation is in progress. To ensure correctness, a node attempts to move only if its parent is not down.

Consider a node i with an old parent *old_p* that wishes to join a new parent *new_p*. The details of the protocol are given in Figure 8.4, and a brief explanation of the protocol follows. Without loss of generality, the algorithms in Figure 8.4 assume that i is the r^{th} child of the old parent and will become the $c + 1^{th}$ child of the new parent. The scenario of message exchange is illustrated in Figure 8.3.

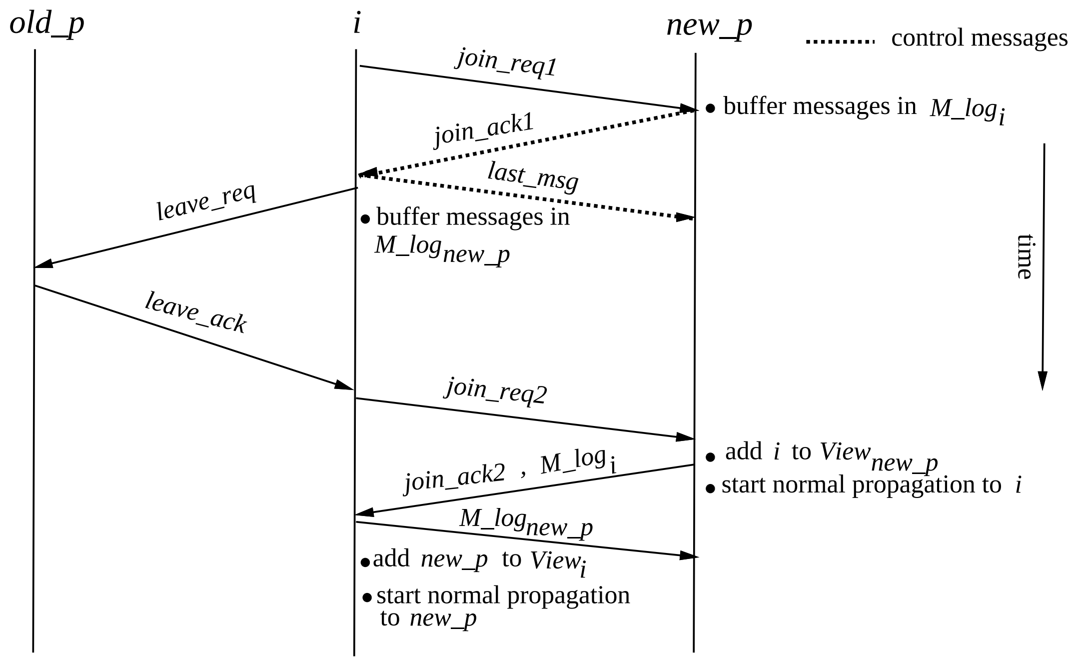

Figure 8.3: Scenario of message exchange while a node i is changing its parent from *old_p* to *new_p* within HPP

Node i sends a *join_req1* message to *new_p* to inform it of its intention to join. On receiving *join_req1*, *new_p* sends a *join_ack1* message to i and starts buffering any further messages generated or received for i in a log called M_log_i. When i receives *join_ack1*, it sends a *last_msg* message to *new_p* to mark the last message sent from i to *new_p* through *old_p* (in the present hierarchy). The *join_ack1* and *last_msg* messages are *special control messages* which travel through the hierarchy exactly like a normal data message. Then, i sends to *old_p* a *leave_req* message and stops sending further messages to *old_p*. However, it starts buffering for *new_p* (in a log called $M\_log_{new\_p}$) all new messages generated or received from its child nodes. When *old_p* receives the *leave_req* message, it stops sending further messages to i, discards i from its $View$, informs its own correspondents of the change and sends a *leave_ack* message to i. On receiving *leave_ack*, i removes *old_p* from its $View$ and sends *join_req2* to *new_p*. *new_p* will then add i to its $View$, inform all its correspondents about the new child, and send an acknowledgement *join_ack2* to i including all messages it has buffered for it in M_log_i previously. On receiving *join_ack2*, i processes these messages, adds *new_p* to its view and finally sends to *new_p* all messages it has buffered for it. *new_p* waits for *last_msg*, and, after receiving it, starts processing messages received from i. The move is considered complete when node i receives *join_ack2* from *new_p*.

It can be shown that no messages will be lost as a result of the reorganisation by arguing that *old_p* and *new_p* will receive all messages i receives from below, and that i will receive all messages that *new_p* has received before the move or will receive after the move [2]. Assume i sends *leave_req* at time t_1. Then, *new_p* will receive directly from i every message i has received from below at time $t \geq t_1$. All messages received by i at $t < t_1$ from below would be propagated by it to *old_p*, and then *old_p* will further propagate them to *new_p* by the normal propagation mechanism. Therefore, *new_p* receives *all* messages received by i from below. Similarly, *old_p* receives directly from i all messages it has received from below at $t < t_1$. All messages received by i from below at $t \geq t_1$ will be sent to *new_p* and *new_p* will further propagate them to *old_p* by the normal propagation mechanism. Therefore, *old_p* receives *all* messages received by i from below.

Assume *new_p* receives i's *join_req1* at time t_2. Messages received by *new_p* at t $\geq t_2$ will be sent to i. *new_p* sends *join_ack1* at $t = t_2$. Since *join_ack1* propagates through the network like other normal messages and since the H_FIFO order is preserved in transmitting messages between correspondents, *any message that is received by new_p at* $t < t_2$ *will be received by old_p and i before join_ack1.* Thus, i is guaranteed to receive all messages that *new_p* received at $t < t_2$ from *old_p* and no messages are missed.

To preserve causal order while the switch is taking place, node *new_p* does not process messages received directly from i until the *last_msg* has been received. This ensures that *new_p* processes all messages sent to *new_p* through *old_p* first, and then it processes any new messages received directly from i. Further, since *join_ack1* propagates through the network like a normal message (before the change of correspondents), i receives *join_ack1* *after* all regular messages propagated by *new_p* before the move. Since i starts receiving new messages directly from *new_p* after *new_p* receives *join_req2* - which is sent by i only after *join_ack1* is received - then the ordering is preserved. Based on the above arguments, Theorem 10, given in Appendix C, shows that causal order is not violated while the change is taking place.

For a new node to join the hierarchy, it follows the same protocol for joining *new_p* from the step of sending *join_req2*. Node *new_p* sends the new node a copy of the database with

[2]The case of reorganisation in the presence of failures is discussed in Section 8.4.5.

Node i

  send *join_req1* to *new_p*;
  on receiving *join_ack1* from *new_p*,
    send a special control message *last_msg* ;
    send to *old_p* *leave_req* and stop sending it messages;
    insert new or child node messages in $M\_Log_{new\_p}$;
  on receiving *leave_ack* from *old_p*,
    set old_View = $View_i[1]$; old_V^P = V_i^P;
    set $View_i[0,0] = new\_p$ and send *join_req2*($View_i[0]$) to *new_p*;
  on receiving *join_ack2*(V_{new_p}, $View_{new\_p}[0]$, M_Log_i) from *new_p*,
    process messages in M_Log_i;
    set $View_i[1]$ = $View_{new\_p}[0]$; V_i^P = V_{new_p}; $V_i.U$ = $V_{new\_p}.W$;
    send $M\_Log_{new\_p}$ to *new_p*;
    send *commit* to *old_p* and discard old_View and old_V^P;

Node *new_p*

  on receiving *join_req1*,
    send a special control message *join_ack1* ;
    insert new generated or received messages in M_Log_i;
  on receiving *join_req2*($View_i[0]$) from i
    add i to its list of child nodes: set $View_{new\_p}[0, c+1] = i$;
    set $View_{new\_p}[c+2]$ = $View_i[0]$; $V_{new\_p}.D_{c+1} = 0$; $V_{new\_p}^{Ch}.L_{c+1} = 0$;
    send add_child(i) to its correspondents;
    send i an acknowledgement *join_ack2*(V_{new_p}, $View_{new\_p}[0]$, M_Log_i);
  on receiving $M\_Log_{new\_p}$ from i,
    wait for *last_msg* then process messages in $M\_Log_{new\_p}$;

Correspondent k of *new_p* on receiving add_child(i)

  update $View_k$ to include i as a child of *new_p*;
  If k is a child of *new_p* then add an entry $V_k^P.D_{c+1}$;

Node *old_p*

  upon receiving *leave_req* from i,
    keep old status for i: set
      old_V_i = $V_{old\_p}.D_r$; $old\_V_i^{Ch}$ = $V_{old\_p}^{Ch}.L_r$;
      $old\_View_i[0]$ = $View_{old\_p}[0]$; $old\_View_i[1]$ = $View_{old\_p}[r+1]$;
    remove i from $View_{old\_p}[0]$, V_{old_p} and $V_{old\_p}^{Ch}$;
    remove row r+1 from $View_{old\_p}$ and rearrange the rows ;
    send del_child(i) to its correspondents;
    send *leave_ack* message to i ;
  on receiving *commit* message from i, discard old status kept for i;

Correspondent k of *old_p* on receiving del_child(i)

  remove i from $View_k$;
  If k is a child of *old_p* then remove the entry $V_k^P.D_r$;

Figure 8.4: Reorganisation algorithm in HPP

join_ack2.

For a node to leave permanently the hierarchy, it has to be a leaf node. Any previous descendants must already have moved. The node sends a leave request to the parent after ensuring that the parent has received and acknowledged all previously sent messages. The parent then removes the node from its view, informs its other children of the change and sends an acknowledgement to the leaving node, which can then leave.

8.4 Failures

This section describes how the protocol circumvents failures, i.e. messages can be propagated past a failed node in both directions. The 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. As in HARP, the child nodes of the failed node elect a coordinator which would take-over all the functions of the failed node. However, the procedure for handling failures is different from the one adopted in HARP, basically because HPP avoids exchanging global states and relies on the information in the aggregate state vectors kept by the correspondents of the failed node.

Assume the failed node is f. Once all its child nodes agree that f has failed, they elect a coordinator CO. The various steps required to be performed after a failure have been grouped into three phases which are summarised below:

- *Phase 1: Transition*
  CO contacts all correspondents of f and a *transition algorithm* is run which is responsible for propagating any message f failed to send successfully to all its correspondents before failing.
- *Phase 2: Diffusion*
  CO takes over the responsibilities of node f and acts as a temporary parent for each child of f, $Ch_j(f)$, and as a temporary child for $P(f)$ to ensure the continuity of propagation flow while f is down.
- *Phase 3: Recovery*
  When f comes up, f runs a *recovery algorithm* to bring itself up-to-date and takes back its function from CO.

In the following, details of the three phases are described.

8.4.1 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 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 8.5. 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, the transition algorithm must ensure that such a message does reach *all* correspondents of f. This problem is divided 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. Each of these four cases is discussed separately below, and in each case it is explained how the appropriate step from the 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 8.5).

- **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

(1) Ask node $P(f)$ for $V_{P(f)}.W$, $V^{Ch}_{P(f)}.L_j$ and $V_{P(f)}.D_j$;
(2) Ask nodes $Ch_i(f)$ for $V^P_{Ch_i(f)}$, $i = 1 \ldots c$;
(3) Let Max_L_f = $\mathsf{Max}\{V^{Ch}_{P(f)}.L_j, \mathsf{Max}_i\{V^P_{Ch_i(f)}.L\}\}$;
If ($Max\_L_f = V^{Ch}_{P(f)}.L_j$) **Then** $i^* = 0$;
Else $i^* = i$ s.t. $Max\_L_f = V^P_{Ch_i(f)}.L$;
(4) Construct the vectors $V^{P_{MAX}}$ and $V^{P_{MIN}}$ s.t.,
$V^{P_{MAX}}[i] = V^P_{Ch_i(f)}.D_i$, $\forall i$;
$V^{P_{MIN}}[i] = \mathsf{Min}_k\{V^P_{Ch_k(f)}.D_i\}$, $\forall i$;
(5) Ask $P(f)$ to send $Ch_i(f)$, $\forall i$, the last ($V_{P(f)}.W - V^P_{Ch_i(f)}.U$) messages sent downwards except messages with $m\_prev = j$;
(6) **If** ($i^* = 0$) **Then**
Ask $P(f)$ to send $Ch_i(f)$, $\forall i$, the last ($Max\_L_f - V^P_{Ch_i(f)}.L$) messages received with m_org=f;
Else {
Ask node i^* to send $P(f)$ the last ($Max\_L_f - V^{Ch}_{P(f)}.L_j$) messages received with m_org= f;
$\forall k \neq i^*$,
Ask node i^* to send $Ch_k(f)$ the last ($Max\_L_f - V^P_{Ch_k(f)}.L$) messages received with m_org=f;
}
(7) Ask nodes $Ch_i(f)$, $\forall i$, to send to their *neighbours* their last ($V^{P_{MAX}}[i] - V^{P_{MIN}}[i]$) messages sent upwards;
(8) Let $y = \Sigma_k V^P_{Ch_k(f)}.D_k - (V_{P(f)}.D_j - Max\_L_f)$;
If $y > 0$ **Then**
Ask $Ch_i(f)$, $\forall i$, to send the last y messages sent upwards to $P(f)$;

Figure 8.5: Transition algorithm

$V_{P(f)}.W$ with $V^P_{Ch_i(f)}.U$: if $V_{P(f)}.W > V^P_{Ch_i(f)}.U$, then $Ch_i(f)$ might have missed some of those messages, and CO asks $P(f)$ to send the missing messages to $Ch_i(f)$ (*see Step 5*).

- **Case 2:** f has generated a message, sent it to $P(f)$, and then died (denoted by $f \to P(f)$, 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 $V^{Ch}_{P(f)}.L_j$ with $V^P_{Ch_i(f)}.L$: if $V^{Ch}_{P(f)}.L_j > V^P_{Ch_i(f)}.L$, then $Ch_i(f)$ has missed some messages and CO asks $P(f)$ to send $Ch_i(f)$ the missing messages (*see Step 6*).

- **Case 3:** f has generated a message, sent it to one of its child nodes (say, $Ch_i(f)$), and then died (denoted $f \to Ch_i(f)$, f dies)
  In order to ensure that such messages are propagated upwards, CO compares $V^P_{Ch_i(f)}.L$

with $V^{Ch}_{P(f)}.L_j$: if $V^{P}_{Ch_i(f)}.L > V^{Ch}_{P(f)}.L_j$, then $P(f)$ has missed one or more messages and CO must ask $Ch_i(f)$ to send the missing messages to $P(f)$. In order to ensure that the messages are propagated downwards, CO compares $V^{P}_{Ch_i(f)}.L$ with $V^{P}_{Ch_k(f)}.L$: if $V^{P}_{Ch_i(f)}.L > V^{P}_{Ch_k(f)}.L$, then $Ch_k(f)$ has missed some messages and CO asks $Ch_i(f)$ to send the missing messages to $Ch_k(f)$ (*see Step 6*).

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

  To ensure that such messages are propagated downwards, CO compares $V^{P}_{Ch_i(f)}.D_i$ with $V^{P}_{Ch_k(f)}.D_i$: if $V^{P}_{Ch_i(f)}.D_i > V^{P}_{Ch_k(f)}.D_i$, then $Ch_k(f)$ has missed some messages sent by $Ch_i(f)$ and CO asks $Ch_i(f)$ to send those messages to $Ch_k(f)$ (*see Step 7*). To ensure that such messages are propagated upwards, CO checks if: $(V_{P(f)}.D_j - Max\_L_f) < \Sigma_k V^{P}_{Ch_k(f)}.D_k$. 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 summarised in Table 8.2. A "–" in Table 8.2 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	–	$V_{P(f)}.W > V^{P}_{Ch_i(f)}.U$
$f \rightarrow Ch_i(f)$, f dies	$V^{P}_{Ch_i(f)}.L > V^{Ch}_{P(f)}.L_j$	$V^{P}_{Ch_i(f)}.L > V^{P}_{Ch_k(f)}.L$
$f \rightarrow P(f)$, f dies	–	$V^{Ch}_{P(f)}.L_j > V^{P}_{Ch_i(f)}.L$
$Ch_i(f) \rightarrow f$, f dies	$V_{P(f)}.D_j - Max\_L_f < \Sigma_k V^{P}_{Ch_k(f)}.D_k$	$V^{P}_{Ch_i(f)}.D_i > V^{P}_{Ch_k(f)}.D_i$

Table 8.2: A summary of failure scenarios and conditions to detect upwards and downwards propagation in HPP

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 $Ch_i(f)$ into one message including the total number of messages that each of them is supposed to receive from others. $P(f)$ or $Ch_i(f)$ receiving diffused messages treat them exactly as if they were coming from f. The only change is that the child nodes $Ch_i(f)$ $(i = 1 \ldots c)$ need not update $V^{P}_{Ch_i(f)}$, 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 reply reaches before the original message, thus disturbing the causal order. In such a case, sorting will ensure that the ordering requirements are still satisfied. Similarly, if there are further replies to replies, it is possible to process them in the correct order by attaching a tag field to each reply, and incrementing the field whenever there is a reply to a reply. The replies would then be sorted in order of this tag field, and a reply (tag value = 1) would be processed before a reply-to-a-reply (tag value = 2).

8.4.2 Diffusion

While node f is down, CO takes over the responsibilities of f temporarily until f comes up again. That is, $Ch_i(f)$'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 H_FIFO order between CO and correspondents of f. This phase starts once the transition phase is terminated, i.e., when $P(f)$ or $Ch_i(f)$'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 8.2 (see Figure 8.2), i.e. CO must update VV_{CO}, increment $V_{CO}.W$ and $V_{CO}.U$ and send m to its own correspondents, but it does not update V_{CO}^P. 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 8.2), but, in addition, sends it to all correspondents of f.

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 8.2); 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 8.2), but does not update $V_{Ch_i(f)}^P$.

8.4.3 Recovery

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 node f comes up, it triggers the recovery algorithm which consists of the following steps:

1. f sends *recovering* message to each of its correspondents. The message to correspondent i includes any messages in f's log which has not been acknowledged by i.

2. Correspondent i receiving such a message should:

   (a) check whether any of the included messages is a duplicate - as it might have already received it during the transition phase-, and, if so discards it; otherwise, it accepts the message and updates V_i.

   (b) check whether it is holding any pending messages in the H_FIFO queue for f, process the non-duplicates then flush the queue.

   (c) extract from its log all messages not acknowledged by f and sends them to f along with an *ack*.

3. f receiving the *ack* from correspondent i:

   (a) accept non-duplicate messages (but does not forward them to its other correspondents as they already have them).

   (b) increment $V_f.U$, 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.

4. After f receives *ack*s from all its correspondents:

   (a) increment $V_f.W$ by the total number of non-duplicate message received from all correspondents.

   (b) send V_f to all its child nodes $Ch_i(f)$.

   (c) ask $P(f)$ for $V_{P(f)}$.

5. normal operation of f is resumed.

Steps 1 through 2(b) ensure 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, it is essential for correspondents of f, who might or might not have received these messages, to flush any pending messages in the H_FIFO queue each of them keeps for f. Step 2(c) ensures that f will receive *all* messages it missed during the failure. Step 3 and 4 allows f to bring itself up-to-date before resuming its functions. Recall that, while f was down, its child nodes were not required to update their $V^P_{Ch_i(f)}$ vector. Therefore, on recovering, f sends its V_f vector to all its child nodes $Ch_i(f)$ so that they can use the incoming vector as their new $V^P_{Ch_i(f)}$. 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. [3] It should be noted that, unlike HARP, when a node comes up, it must take the same position in the hierarchy it was taking when it failed to ensure no loss of messages.

[3] 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.

8.4.4 Partitions

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.

8.4.5 Reorganisation Despite Failure

This section discusses the case in which failures occur while reorganisation is taking place. The reorganisation algorithm can proceed in spite of some kinds of failures, but if any of nodes *old_p*, *new_p* or i fail, then it is aborted.

The reorganisation algorithm assumes that nodes i, *old_p* and *new_p* must not fail while reorganisation is taking place. If one of these nodes fail, then the reorganisation must be aborted. This is why nodes *old_p* and i maintain their old views and state vectors (see Figure 8.4) - so that they can be restored in case of failure. In case *new_p* fails, node i is still able to receive all messages from above through *old_p*, in spite of a temporary interruption and small delay on account of the attempted reorganisation. In case *old_p* fails, then i is able to participate in the *transition*, *diffusion* and *recovery* algorithms and vice versa.

On the other hand, if any of the other nodes, say the ones along the path between *old_p* and *new_p*, were to fail, then the reorganisation algorithm can proceed in spite of these failures. The detailed proof for this claim is given in Theorem 11 of Appendix C.

8.5 Discussion

The major advantage of HPP is that it allows aggregation of the important aspects of the state into a few state vectors containing minimal information. By encapsulating the state information, HPP requires each node to maintain vectors of size proportional to the number of its correspondents. These aggregate state vectors provide enough information for the determination of missing messages, and therefore, their exchange is sufficient to circumvent failures. HARP on the other hand requires exchanging vectors of size n periodically or when failures or reorganisation occur, which may result in high communication overhead for very large scale systems. Therefore, HPP allows the system to scale better than HARP.

The main limitation of HPP 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 and the tree to be partitioned. Of course, messages will still not 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. 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. For instance, if each node keeps vectors describing the state of its correspondents and their correspondents, the presented algorithms can be extended to handle two levels of failures and so on. In this case, there is a tradeoff between the probability and associated cost of such successive failures and the cost and complexity of maintaining the additional information.

In HARP, nodes keep $AckM$ to record when messages have been received by all nodes - or a subset of nodes- before discarding them from the log. HPP does not keep such a matrix, hence, saving storage. However, its policy for discarding a message from the log relies on the value T (a timeout duration after which a node discards a message) which is based on estimation. Consequently, a message that has been missed by a node due to a failure is sent to it during the transition algorithm only if the correspondents of the failed node are still keeping the message in their log, which is highly dependent on how large the estimated value of T is.

During reorganisation of the hierarchy, HPP avoids exchanging global information, which reduces communication overhead. However, unlike HARP, a node cannot detach itself from the hierarchy (leave) and then reintegrate (join) independently as it cannot compare its state to the node it is joining. Rather, it has to join the node it wishes to join first, then it leaves the node it wishes to leave to guarantee no loss of messages. This approach works fine for a node moving from one location to another. However, if a node is required to be removed from service temporarily, say for repair, it has to leave the tree permanently then join as a new node and it will receive a complete copy of the database. This process could produce a high overhead if the size of the database is large. However, it depends on the frequency at which such a reorganisation occurs.

HPP's support for ordering of delivery is weaker than HARP. More precisely, HPP supports FIFO and causal ordering during normal operation. However, in the presence of failures the ordering cannot be guaranteed. For applications where the causal dependency is known in advance (such as replies or follow-ups are known to be dependent on the original posting), then their causal order could be preserved. However, it cannot be generalised. Therefore, HPP is suitable for applications such as bulletin board services or applications that require no order to resolve inconsistencies.

8.6 Summary

This chapter has presented a Hierarchical Propagation Protocol (HPP) where each node encapsulates information about itself, its parent and children into state vectors. These aggregate state vectors provide enough information for a node to determine messages that have been missed. A reorganisation algorithm and a procedure for handling failures have been described which rely on those vectors and avoid exchanging global state information.

Therefore, HPP takes advantage of the hierarchical structure in terms of distributing the load evenly among the nodes and minimising redundancy. Further, it exploits the structure to enhance the scalability of the system by compressing the state vectors required to be exchanged to circumvent failures, which reduces the communication overhead. However, it is less reliable as it can tolerate only special patterns of failure.

In comparison with HARP, HARP is a more general protocol than HPP, more reliable and can handle any pattern of failures. Further, it provides several orders of delivery (FIFO, causal and total order) from which an application may choose depending on its requirements. On the other hand, it requires the exchange of global state information.

Chapter 9

Conclusions

This dissertation presented a protocol for management of replicated data in large scale systems. This chapter summarises the main conclusions, and suggests possible further work.

9.1 Summary

Although extensive research has been done in managing replicated data, existing solutions cannot scale to a large number of replicas. Mainly, this is because they manage replicas as one flat group of nodes. Requiring a node to communicate with all other nodes in the system might be adequate for local area networks with few replicas, but is inefficient and entirely impractical for wide area, massively replicated systems. Also, it has been observed that current applications require different degrees of consistency, varying between strong and weak consistency. Further, with weak consistency, some applications require the ability to access information that is more up-to-date than what is available locally. However, existing replication models lack the ability to provide the application with the flexibility of choosing the desired level of consistency.

Motivated by these observations, the new replication protocol presented in this dissertation (HARP) is designed to be scalable to thousands of replicas linked across internetworks. Further, it integrates both strong consistency and weak consistency into the same framework and gives the application the ability to select between them depending on its requirements. The protocol is designed on the basis that replicas are organised into a logical multilevel hierarchy, where nodes are grouped into clusters, and clusters are organised into a tree. The hierarchical structure allows for exploiting localised communication, which is taken as the key to achieve scalability.

A new service interface is proposed (Chapter 3) that provides both asynchronous and synchronous operations. Asynchronous operations commit at any replica, whereas synchronous operations commit after a quorum is assembled from nodes of the top level of the hierarchy. Strong consistency is achieved through synchronous operations, which are less expensive than traditional replication protocols since synchronisation is limited to a small number of nodes. In order to ensure that updates are observed by all replicas, HARP provides an efficient propagation scheme where each node needs to communicate with a few other nodes only, namely its parent, neighbours and children in the hierarchy. It has been shown that the scheme reduces the communication overhead, limits the amount of state each node needs to keep and reduces the size of the log, which enhances scalability. This is achieved while ensuring reliable delivery and avoiding redundancy. Further, the hierarchical pattern of propagation allows the service to provide different levels of staleness by reading from different levels of the hierarchy. Therefore, the proposed interface gives the application developer the flexibility to tailor the service to achieve the required degree of consistency by choosing the appropriate operations to manage the data, while hiding the details of the interface specifications from the users.

Reconfiguration schemes are necessary to allow a node to join or leave the set of replicas and the ability to recover from failures. HARP provides restructuring operations that allow the logical hierarchy to be built, expanded and dynamically reconfigured (Chapter 4). In contrast to contemporary reconfiguration schemes, the proposed operations rely on a weak consistency semantic where a node needs to negotiate with a few other nodes only to perform a change rather than synchronising will all nodes. This approach reduces the overhead associated with reconfiguration, produces low message traffic and does not suspend normal operation while the protocols are running. It has been shown that the operations are correct by proving that no messages are lost as a result of a change and that all replicas converge to a single consistent view of the hierarchy (Appendix A). Based on the reconfiguration operations, methods for handling failures and partitions are presented which ensure propagation flow between the operational and connected sites despite the failures and guarantee that eventual reliable delivery is achieved after repair.

Many applications rely on delivery order mechanisms to resolve conflicts resulting from asynchronous updates. HARP provides a variety of delivery orderings - latest-wins, FIFO, causal and total order - and an application may choose from them. A new algorithm supporting causal order within replicated systems is presented (Chapter 5). The novelty of the algorithm is that it cuts down the size of the timestamp, required to verify causality, to the number of nodes in a cluster as opposed to previous solutions that used a timestamp of size n. Hence, it enhances scalability. This is achieved by relying on the locality of propagation. It has been shown that causal delivery is guaranteed during normal operations as well as under reconfiguration (Appendix B).

A detailed simulation study was carried out to evaluate the performance of HARP (Chapter 6). The benefits and losses resulting from using synchronous versus asynchronous operations have been quantified under different system configurations and load mixes. It has been shown that in internetworks synchronous operations degrade the performance tremendously.

The throughput is severely limited with high loads, high update rates and communication delays, especially when the number of replicas is large. On the other hand, asynchronous operations have shown much better performance and can cope with heavily loaded systems with a little loss in reading some out-of-date information. The results obtained confirm our claim that, for applications that need high performance, asynchronous operations are more suitable as long as they can tolerate some inconsistency. As expected, asynchronous reads performed at higher levels of the hierarchy have been shown to return less stale information with some increase in the response time. Further, it has been shown that maintaining strong consistency by assembling quorums from all nodes in the system, as it is the case in current solutions, is not feasible in internetworks.

The performance has been evaluated on several network topologies and with different numbers of replicas. The results have revealed that a symmetric hierarchical structure yields the best performance and that distributing the load evenly among the nodes in the top cluster is crucial for synchronous operations. Further, it is suggested to keep the number of nodes in the top cluster small to reduce the overhead on those nodes. Also, it has been shown that performance is very sensitive to the number of levels in the hierarchy. These results give some indications on the effect of the design of the logical hierarchy on the performance and give broad guidelines on the factors that should be considered while designing it.

A simulation study was carried out to compare the performance of HARP to the TSAE protocol under different load mixes and communication overhead (Chapter 7). The results obtained revealed that in internetworks, HARP results in considerable performance improvement over TSAE, especially under heavily loaded systems. This is mainly because TSAE involves exchanging a fair amount of duplicate messages which increases the traffic. The speed of propagation was observed to be much lower than HARP due to the periodic updates. Finally, it was shown that relying on a hierarchical structure and using localised communication improves performance greatly.

It has been shown that HARP reduces communication traffic as well as storage space by exploiting locality of propagation. However, it requires the exchange of vectors of size n periodically and during reconfiguration. For some systems, $O(n)$ state is unacceptably large. An alternative hierarchical propagation protocol to HARP is presented in Chapter 8, HPP, which reduces this overhead by encapsulating the state information such that each node keeps state vectors of size proportional to the number of its correspondents in the hierarchy. These aggregate vectors are exchanged when reorganisation or failures occur. Therefore, HPP allows the system to scale better than HARP. However, it is less reliable as it can tolerate only special patterns of failure due to the minimal information kept. Further, its support for ordering delivery is weaker than HARP. Therefore, it is suitable for applications that require no order of delivery or applications with simple ordering requirements.

To conclude, this work has proposed a protocol for managing replicated data based on a logical hierarchy. It has been shown that the hierarchical structure allows the system to scale well. Further, it gives the ability to provide different degrees of consistency. As many design aspects were considered as time and resources allowed. It is believed that the protocol is suitable for many distributed applications such as bibliographic databases, network news systems, Archie and many other information services. With the growth of internetworks, these services are accessed by a vast number of users. It is anticipated that they will be highly replicated in order to meet their performance requirements, in which case current replication solutions are inadequate since they cannot scale.

9.2 Future Work

In this work, methods by which the hierarchy can be reconfigured have been presented. But the problem of designing the hierarchy and determining which nodes are grouped into which clusters has not been addressed. This problem involves several issues that need to be fully exploited. Since network behaviour changes over time, different mechanisms should be examined to effectively monitor and report on the performance of the underlying network. Based on this information, algorithms need to be developed to determine the appropriate clustering to use within the hierarchy as well as deciding which changes are to be made to the hierarchy to adapt to the dynamic characteristics of the network. As pointed out in Section 3.7 and from the performance results of Chapter 6, the design of the hierarchy is an optimisation problem that should consider several factors, such as topology, network delays, reliability, number of levels in the hierarchy, number of nodes per cluster and so on. These algorithms need to be validated and different objective functions should be examined.

A related problem to the above is how a new replica would know which cluster it should join and how to inform nodes about the restructuring operations they should perform to reconfigure the hierarchy. This service could be regarded as an enhanced location service, which maintains a dynamic view of the logical hierarchy as well as information about the nodes, such as their network addresses. When a new replica wants to join the hierarchy, the replica queries the service which informs it which cluster to join. Similarly, when changes in the network are detected and a new hierarchy is recomputed, the service informs the nodes of the changes that need to be done. This service as well as its data should be replicated and the weak consistency approach should be used to maintain it. The degree of replication of this service is one design issue that has to be investigated. One option is to place a replica at each node, and another option is to have a replica of the service responsible for a certain portion of the hierarchy. Also, reconciliation methods need to be devised to reconcile conflicting decisions taken by different servers.

The performance study evaluating HARP can be extended to include the evaluation of the restructuring protocols. The overhead of these protocols should be quantified under different system configurations to evaluate how well they perform. Further, the simulation model has considered only unordered Fast_Write. The various other delivery orderings supported need to be incorporated to evaluate their performance. Also, the model needs to be extended to account for site and link failures. Several metrics have to be evaluated such as the time it takes the protocol to recover from these failures, how the speed of propagation is affected as well as the associated overhead.

Due to the detailed simulation model adopted, we were unable to measure the performance for more than 39 nodes. It is desirable to study the performance of the protocol for thousands of replicas to evaluate its behaviour in very large scale systems. This requires the model to be greatly simplified for those experiments to become feasible. Further, the performance evaluations reported are all based on simulation and artificial loads. The analysis would benefit from a re-evaluation using real workload and real network costs. For instance, traces of usage of information services, such as Archie, could be used to drive the simulators.

Studying the *availability* of synchronous operations within HARP, which is the probability that a quorum can be formed despite failures, and comparing it to previous synchronous protocols is a topic which deserves further investigation. It is expected that the availability can be kept high with HARP since the restructuring operations enable more functional nodes to be added to the top cluster when failures occur, keeping the number of nodes in this cluster nearly constant. Further research is needed to study this issue.

Another area that needs to be explored is to associate with HARP methods for controlling the encountered inconsistency. It is believed that the General Site Escrow technique (GSE) (see Section 2.2.9) could be extended using HARP for getting better performance. Since nodes at higher levels of the tree have a better global view of the transactions occurring in the system, then it might be advantageous to borrow escrow from those nodes rather than from a randomly chosen node. This would result in a less conservative escrow estimate than GSE, hence, granting more requests and increasing concurrency. Consequently, the issue of allocating more escrow to nodes in higher levels needs to be studied to see whether it gives better performance. Also, it would be interesting to see whether the hierarchical structure can be used to reduce the worst case quorum size (n) of the GSE technique.

The reconfiguration schemes have provided four basic operations which have been used to compose more complex operations. Two of the composite operations have been chosen and optimised to reduce some of the overhead. Similarly, algorithms can be developed to optimise other composite operations, such as move, merge, split, etc, if they are expected to be used frequently.

HPP has been shown to survive one level of failure but not successive failures. Further analysis should be performed to evaluate the probability of occurrence of such patterns of failure and how they can affect the performance. Also, HPP can be extended to bypass these failures by keeping additional information in the state vectors. This extension needs to be exploited in detail and the additional cost has to be contrasted with the likelihood that these situations occur.

Appendix A

Correctness of Restructuring Operations

This appendix establishes the correctness of the restructuring operations presented in Chapter 4. It begins by presenting the correctness of the basic operations. 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. Next, it presents the correctness of the change-parent and take-over operations.

A.1 The Basic Operations

Lemma 1 *While node i is joining a cluster C_x, $join\_list_i$ is updated to include all current members of C_x and the parent of C_x.*

Proof: The proof is based on the way the cluster is constructed. Assume $join\_list_i = \{k\}$; that is, i is executing join(C_x, k) and k is the *connect node.* Further, assume that p is the *parent* of C_x; that is, $C_x \in View_p.p.Child\_Cids$.

Case 1 : C_x is empty. Two subcases can occur:

Case 1.1 $k = null$

Then, $join\_list_i = \phi$, as i is the only member in the hierarchy.

Case 1.2 $k = p$

The parent p receiving $join\_req$ from i, sets $View_p.i.Cid = C_x$ and replies with its view of C_x $Cl\_view_p = Get\_View(p, C_x) = \{p\}$ as there are no *current* members and the lemma holds.

After i receives ack_join from p, it updates $View_i$ s.t. $C_x \in View_i.p.Child\_Cids$

and then, i becomes *current*. That is; i observes p as a parent in its *View* and p observes i as a child. Therefore,

When a cluster has one current member i and a parent p, then
- $C_x \in View_i.p.Child\_Cids$, *and*
- $View_p.i.Cid = C_x$. (1)

Case 2 : C_x contains one member j

Case 2.1 $k = p$

if j is a *current* member, then from (1), $j \in Cl\_view_p$. Since i updates $join\_list_i$ from Cl_view_p to include additional members then $join\_list_i = \{p, j\}$ and the lemma holds.

Case 2.2 $k = j$

If j is a *current* or a *leaving* member, then from (1), $p \in Cl\_view_j$ and $join\_list_i = \{p, j\}$. If j is a *joining* member, then from (Cond 2), j should be joining through p i.e. $p \in Cl\_view_j$ and the lemma holds.

Further, if j is a *current* member (i.e. $join\_list_i = \{p, j\}$) then p and j receiving a *join_req* from i will set $View_p.i.Cid$ and $View_j.i.Cid$ to C_x. Node i receiving *ack_join* from p and j includes them in $View_i$ as parent and neighbour respectively and then it becomes *current*. Therefore,

When C_x has two current members i, j and a parent *p, then*
- $View_j.i.Cid = C_x$ *and* $C_x \in View_j.p.Child\_Cids$,
- $View_i.j.Cid = C_x$ *and* $C_x \in View_i.p.Child\_Cids$, *and*
- $View_p.i.Cid = View_p.j.Cid = C_x$ (2)

Case 3 : C_x contains two members r and j

Case 3.1 $k = p$

If j (or r) is a *current* member then, from (1), j (or r) $\in Cl\_view_p$. If j and r are *current* then from (2) j and $r \in Cl\_view_p$ and the lemma holds.

Case 3.2 $k = r$

Case 3.2.1 r is a *current* or *leaving* member

If j is not *current*, then from (1) $p \in Cl\_view_r$. If j is *current*, then from (2), p and $j \in Cl\_view_r$

Case 3.2.2 r is *joining*

From (Cond 2), either $p \in Cl\_view_r$ or $j \in Cl\_view_r$, where j is a *current* or a *leaving* member. Hence, i will send p (or j) a *join_req* and from (1), $j \in Cl\_view_p$ if j is *current* (or $p \in Cl\_view_j$) and the lemma holds.

Case 3.3 $k = j$

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

Further, if r and j are *current*, since $join\_list_i = \{p, r, j\}$ then p, r and j will set $View_p.i.Cid, View_r.i.Cid$ and $View_j.i.Cid$ respectively to C_x after receiving *join_req* from i. Since i becomes *current* only when it receives *ack_join* from p, r and j then by then, it will have $View_i$ updated s.t. $View_i.r.Cid = View_i.j.Cid = C_x$ and $C_x \in View_i.p.Child\_Cids$. (3)

From (1), (2) and (3) we can generalise that from the way C_x is constructed, if C_x contains m *current* members and a *parent* p then
For each current member j,

$View_j.r.Cid = C_x$, $\forall r$ *current members and* $C_x \in View_j.p.Child\_Cids$.

For the parent p,

$View_p.r.Cid = C_x$, $\forall$ r *current members of* C_x. (4)

Therefore, in general, while i is joining C_x through the connect node k:

If k is either *current* or *leaving* or is the *parent* of C_x then from (4) Cl_view_k contains all *current* members and the *parent* of C_x.

If k is a *joining* member, then from (Cond 2), i updates $join\_list_i$ from Cl_view_k such that it will contain at least one node r where r is a *leaving* or *current* member of C_x or the *parent* of C_x. Since i sends a $join\_req$ to r and updates $join\_list_i$ from Cl_view_r, then from (4), the lemma holds. □

Corollary 1 *If* i *is a current or a leaving member of* C_x *or is the parent of* C_x*, then the function Get_View(*i, C_x*) returns all current members and the parent of* C_x*.*

Proof: Node i becomes *current* only when it receives *ack_join* from j, $\forall$ $j \in join\_list_i$. By Lemma 1, when i is *current* $View_i.j.Cid = C_x$ for every *current* member j and if the *parent* is p then $C_x \in View_i.p.Child\_Cids$. Therefore, the corollary holds if i is *current*. Since i while *leaving*, changes only $View_i.i$, then its view of C_x is the same as when it was *current* but excluding i and the corollary holds. □

Lemma 2 *If two nodes* i *and* j *are leaving cluster* C_x *simultaneously then* $leave\_list_i$ *and* $leave\_list_j$ *are formed or updated such that*

2.1 $i \notin leave\_list_j$

2.2 $j \notin leave\_list_i$

Proof: Assume node i (j)

- propagates an *update_view* declaring it is leaving at time $t = t_i$ (t_j),
- receives *update_view* from j (i) at $t = t_i^1$ (t_j^1), and
- receives *ack_leave* from j (i) at $t = t_i^2$ (t_j^2)

The leave operation can occur in two ways:

Case 1 : in serial; that is, $t_j^1 < t_j$ or $t_i^1 < t_i$

Case 1.1 $t_j^1 < t_j$

In this case, j receives i's *update_view* before it sends its own, as shown in Figure A.1(a). At $t = t_i$, from Corollary 1, $j \in leave\_list_i$ and i sends j an *update_view*. At $t = t_j^1$, j sets $View_j.i.Cid = null$ and sends i *ack_leave*. Then, at $t = t_j$, $leave\_list_j$ is formed s.t. $i \notin leave\_list_j$ and 2.2 holds. At $t = t_i^2$, i removes j from $leave\_list_i$ and 2.1 holds.

Case 1.2 $t_i^1 < t_i$

The same proof as Case 1.1 applies, replacing every i by j and vice versa.

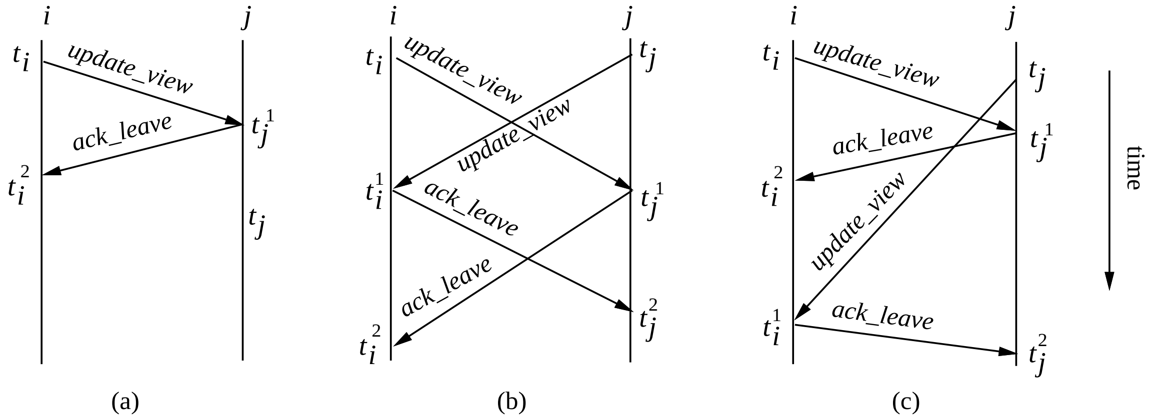

Figure A.1: Two nodes leaving the same cluster simultaneously

Case 2: in parallel; that is, $t_i < t_i^1$ and $t_j < t_j^1$

Then $j \in leave\_list_i$ and $i \in leave\_list_j$. Three subcases could occur:

Case 2.1 $t_i^1 < t_i^2$ and $t_j^1 < t_j^2$ (Figure A.1(b))

At $t = t_i^1$, i removes j from $leave\_list_i$ and sends j *ack_leave*. Similarly, j removes i from $leave\_list_j$ at $t = t_j^1$.

Case 2.2 $t_i^2 < t_i^1$ and $t_j^1 < t_j^2$ (Figure A.1(c))

At $t = t_i^2$, i sets $flag$ of j in $leave\_list_i$ to 1. Node j discards i from $leave\_list_j$ at $t = t_j^1$. Node i discards j from $leave\_list_i$ at $t > t_i^2$ either when $flag = 1, \forall r \in leave\_list_i$ or at $t = t_i^1$, whichever happens earlier.

Case 2.3 $t_i^1 < t_i^2$ and $t_j^2 < t_j^1$

This case is similar to Case 2.2 replacing every i by j and vice versa.

Therefore, in any cases i removes j from $leave\_list_i$ and vice versa and the lemma holds. □

Lemma 3 *If i is joining C_x and j is leaving C_x simultaneously then $join\_list_i$ and $leave\_list_j$ are updated such that i will receive any message that j had before leaving C_x and i does not have.*

Proof: Two cases can occur:

Case 1: while i is joining, $j \in join\_list_i$. Assume:
- j propagates $update\_view$ at $t = t_j$,
- j receives $join\_req$ from i at $t = t_j^1$, and
- j receives ack_leave from k, $\forall k \in leave\_list_j$ at $t = t_j^2$ ($t_j^2 > t_j$)

Case 1.1 $t_j^1 < t_j^2$

In this case, $join\_req$ reaches j while it is *leaving*, as shown in Figure A.2 (a). At $t = t_j^1$, j adds i to $leave\_list_j$ and sends i ack_join. As i sends ack_leave to j only when it has received the missing messages needed from j; and since j discards $leave\_list_j$ and leaves only when it has received ack_leave from $k, \forall k \in leave\_list_j$; then the lemma holds.

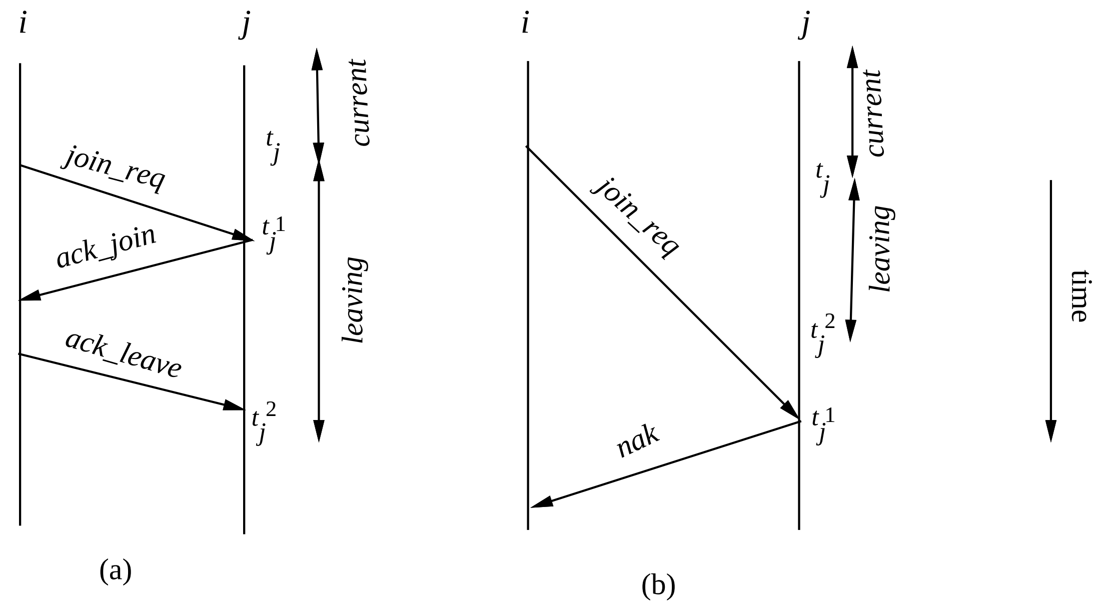

Figure A.2: Node i is joining a cluster and node j is leaving the same cluster simultaneously

Case 1.2 $t_j^1 > t_j^2$

In this case, $join\_req$ reaches j after it discarded $leave\_list_j$, as shown in Figure A.2(b). At $t = t_j^1$, j sends a nak to i, then i updates $join\_list_i$ by replacing j by any other member k of C_x and asks k for any missing messages. Since j stops sending messages at $t = t_j$ and leaves at $t = t_j^2$, then at $t = t_j^2$, k has received every message j had at $t < t_j$. Since i asks k for messages at $t > t_j^2$, then i receives every message j had at $t < t_j$ and the lemma holds.

Case 2: while i is joining, $j \notin join\_list_i$

In this case, i is joining after j has already left. From (Cond 1), there exists at least one *current* member of C_x. From Lemma 1, $join\_list_i$ contains all *current* members of C_x and i sends them a *join_req*. Assume any *current* member k sends *ack_join* to i at $t = t_k$. Then, i updates $join\_list_i$ from k's view, and as $j \notin join\_list_i$ then $View_k.j.Cid \neq C_x$ at $t = t_k$. This implies that j left C_x at $t < t_k$ and that at $t = t_k$, k has received every message j had at t $< t_j$. Since at $t > t_k$, i extracts from k all messages k had and i did not have, then i will receive every message j had at t $< t_j$ and the lemma holds. □

Theorem 1 *If node i is leaving C_x at time t_i, then*

1.1 Any message m that i has originated or received from its children at $t < t_i$ will be received by the current and joining members of C_x and the parent of C_x.

1.2 Messages generated or received by i at $t \geq t_i$ are not sent to members or the parent of C_x.

1.3 After leaving, i will not receive any messages from members or parent of C_x

1.4 $View_j$, $leave\_list_j$ and $join\_list_j$, $\forall j \in N$ are updated to reflect the change.

Proof: From Corollary 1, $leave\_list_i$ contains all *current* members and the *parent* of C_x. Since before propagating *update_view*, i waits until all messages sent to j at $t \leq t_i$ have been acknowledged $\forall j \in leave\_list_i$ and since j sets $View_j.i.Cid$ to *null* only after receiving *update_view* from i then, 1.1 holds for *current* members and the *parent* of C_x. From Lemma 3, it holds for the *joining* members as well.

At $t = t_i$, i sets $View_i.i.Cid = null$ and $View_i.i.P = null$. Therefore, i will not send messages except to its children, if any, and 1.2 holds.

$\forall j$ *current* members and *parent* of C_x, j updates $View_j.i$ when it receives *update_view* from i such that $View_j.i.Cid = null$. Consequently, j will stop sending messages to i. $\forall j$ *joining* members of C_x, since j will receive either *nak* or *ack_join* including $View_i.i$ from i, then in either case $View_j.i.Cid \neq C_x$ and j will not send any messages to i. $\forall j$ *leaving* members of C_x, they stopped sending messages to any member of C_x. Therefore, 1.3 holds.

Since i's row is the only row affected in the $View$ and as all j *current* members and *parent* of C_x will receive *update_view* from i and will proceed in propagating the message, then every node k in any other cluster will update $View_k.i$.

Since $\forall j \in N$, $leave\_list_j$ is affected by i leaving C_x only if j is leaving C_x at the same time. From Lemma 2, $leave\_list_i$ and $leave\_list_j$ are updated correctly.

Since $\forall j \in N$, $join\_list_j$ is affected by i leaving C_x only if j is joining C_x simultaneously. Lemma 3 showed how $join\_list_j$ is updated such that j will receive any messages i had before leaving and j does not have. Then 1.4 is true and the lemma holds. □

Lemma 4 *If two nodes i and j are joining C_x simultaneously, then at least one of them will include the other in its join_list and will send it a join_req.*

Proof: Assume i performs $\mathsf{join}(C_x, k)$ and j performs $\mathsf{join}(C_x, r)$ i.e. i sends k a *join_req* and j sends r a *join_req*. Nodes k and r are the *connect* nodes which are *current, leaving* or *joining* members of C_x or the *parent* of C_x.

Case 1 $r = k$

Assume k receives *join_req* from i and j at $t = t_k$ and $t = t_k^1$ respectively.

Case 1.1 $t_k < t_k^1$ (Figure A.3)

At $t = t_k$, k includes i in its view by setting $View_k.i.Cid = C_x$. So, at t $= t_k^1$, k sends j its view of C_x including i. Consequently, j adds i to $join\_list_j$ and sends i a *join_req* (if i has not already joined) and the lemma holds.

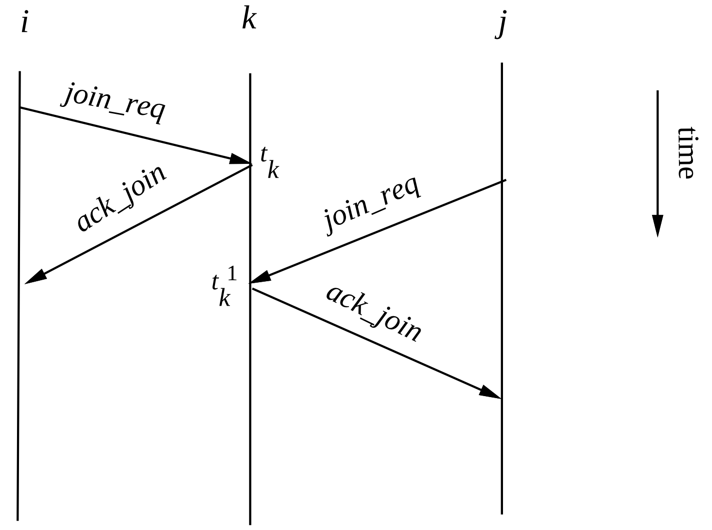

Figure A.3: Nodes i and j are joining the same cluster simultaneously

Case 1.2 $t_k > t_k^1$

Similarly, it can be shown that i adds j to $join\_list_i$ and sends j a *join_req* and the lemma holds.

Case 2 $r \neq k$

From (Cond 1), while i and j are joining there exists at least one node q s.t. q is the *parent* of C_x or a *current* member of C_x. From Lemma 1, $q \in join\_list_i$ and $q \in join\_list_j$. Then, i and j will send q a *join_req* and the same proof of Case 1 applies by replacing k by q. □

Theorem 2 *If a node i joins cluster C_x then*

2.1 i receives every message any member of C_x or the parent of C_x had or will receive.

2.2 every current or joining member of C_x and the parent of C_x receives every message i had or will receive.

2.3 $View_j, leave\_list_j$ and $join\_list_j$ $\forall j \in N$ are updated to reflect the change.

Proof: For any node j, a *current* or a *joining* member or the *parent* of C_x receiving a *join_req* at $t = t_j$, it sends i its VV_j and LL_j and sets $View_j.i.Cid = C_x$. Therefore,

If j is a member of C_x, then any message that j generates or receives from its children at $t \geq t_j$ will be sent to i (1)

If j is the *parent*, then any message that j generates or receives from its neighbours, parent or children of other clusters than C_x at $t \geq t_j$ will be sent to i (2)

When i receives *ack_join* from j at $t = t_j^1$ ($t_j^1 > t_j$), i sets $View_i.j.Cid = C_x$ if j is a neighbour and $View_i.i.P = j$ if j is a parent. Then,

all messages that i generates or receives from its children at $t \geq t_j^1$ will be sent to j (3)

Further, comparing VV_i and LL_i with VV_j and LL_j, i extracts from j every message j has and i does not have. Since VV_j and LL_j reflect j's state at $t = t_j$, then,

i will receive every message j had at $t \leq t_j$ (4)

Since i sends to j every message i has and j does not have, then,

j will receive every message i had at t $\leq t_j^1$. (5)

From (1) and (4) i receives every message a neighbour j has generated or will generate or has received or will receive from a child (6)

From (2) and (4) i receives every message the parent has generated or will generate or has received or will receive from a neighbour, parent or a child of another cluster (7)

From (6) and (7), 2.1 holds for any node j, a *current* or *joining* member or the *parent* of C_x provided that j receives a *join_req*. From (3) and (5), for every node j, a *current* or a *joining* member of C_x or the *parent* of C_x, j will receive every message i already had and every message i generates or will receive from a child. Then 2.2 holds provided that j receives a *join_req*. From Lemma 1, as all *current* members and the *parent* of C_x receive a *join_req* then 2.1 and 2.2 hold for *current* members and the *parent*. From Lemma 4, since at least one of the joining nodes will include the other and sends a *join_req* then 2.1 and 2.2 hold for *joining* members. From Lemma 3, i is guaranteed to receive any message that a *leaving* member had before leaving C_x and i did not have. Then 2.1 holds for *leaving* members.

After i receives *ack_join* from j, $\forall j \in join\_list_i$, i propagates an *update_view* message containing its new row $View_i.i$. From 2.2 every *current* or *joining* member and the *parent* of C_x will receive the message and will proceed in propagating the message. Then, every node j in any other cluster will update $View_j.i$.

$\forall j \in N$, $join\_list_j$ should be updated to include i only if j is joining C_x. From Lemma 4 *join_lists* are updated correctly.

$\forall j \in N$, $leave\_list_j$ is affected by i joining C_x only if j is leaving C_x. Lemma 3 showed that $leave\_list_j$ is updated such that i will receive every message j had before leaving. Then, 2.3 holds and this completes the proof. $\square$

A.2 The change-parent Operation

Theorem 3 *If a cluster C_x change-parent from node old_p to node new_p, then*

3.1 every member of C_x will receive every message new_p had or will receive

3.2 new_p will receive every message a member of C_x had or will receive

3.3 $View_i, leave\_list_i$ and $join\_list_i$ $\forall i \in N$ are updated to reflect the change.

Proof: A node i, a *current* or a *joining* member of C_x, receiving *change_par* from *new_p*, replaces its old parent *old_p* by the new parent *new_p* in $View_i.i$ and starts sending *new_p* messages. Node *new_p*, receiving *ack_par* from i, places i as a child in $View_{new\_p}$ and starts sending i messages after exchanging missing messages with it. Therefore, 3.1 and 3.2 hold for every member of C_x provided that it receives a *change_par* from *new_p*.

Since node *old_p* sends *new_p* a list of all *current* members of C_x with *change_req*, then all *current* members of C_x receive *change_par* from *new_p*. Hence, 3.1 and 3.2 are true for all *current* members of C_x.

If i is joining C_x while C_x is changing parent, then i sends a *join_req* to *old_p*. If *old_p* receives i's *join_req* before sending *change_req* to *new_p*, then the *change_req* to *new_p* will include i as a member of C_x and *new_p* will send i a *change_par*, then 3.1 and 3.2 hold for i. If *old_p* receives i's *join_req* after sending *change_req* to *new_p*, then *old_p* will return to i its view of C_x containing *new_p* as the new parent. Therefore, i will add *new_p* in $join\_list_i$ and will send *new_p* a *join_req*, and from Theorem 2, 3.1 and 3.2 hold for i. Therefore, 3.1 and 3.2 hold for all *joining* members of C_x.

If i is leaving C_x while C_x is changing parent: if i receives *change_par* from *new_p* while leaving, then i will replace *old_p* by *new_p* in $leave\_list_i$ and will not leave until *new_p* extracts needed messages from i. If i receives *change_par* from *new_p* after it has already left C_x, it will send *new_p* a *nak*. Since *new_p* receiving a *nak* will extract any missing messages from any *current* member of C_x then *new_p* will receive any message i had before leaving. Then 3.2 holds for *leaving* members of C_x.

Since every node that its row in $View$ has been updated (members of C_x, *old_p* and *new_p*) propagates an *update_view* message containing its new row, then, every node $i \in N$ will update $View_i$ correctly.

$\forall i \in N$, $join\_list_i$ should be updated only if i is joining C_x. It has been shown above that $join\_list_i$ is updated correctly when i is joining C_x.

$\forall i \in N$, $leave\_list_i$ is affected by C_x changing parent only if i is leaving C_x. Since i will replace old_p by new_p in $leave\_list_i$ after receiving $change\_par$ from new_p, then $leave\_list_i$ is updated correctly. Therefore, 3.3 holds and this completes the proof. □

A.3 The take-over Operation

Theorem 4 *If a node f fails then the* take-over *operation ensures that:*

4.1 every child of f will be linked to another parent node

4.2 if f fails before sending a message m, and at least one node k has m in its Log, where k is a neighbour, a child or the parent of f, then m will be propagated to every node $i \in N$.

Proof Assume a cluster C_x is a child of f and that node i is taking over f.

If j is a *current* member of C_x, then, whether $status_f$ was normal, joining or leaving when it failed, $C_x \in View_i.f.Child\_Cids$ and i will send $change\_par$ to j. Then, 4.1 holds for *current* members of C_x.

If j was joining C_x when f failed then, if e, any member of C_x, receives $join\_req$ from j before receiving $change\_par$ from i, then e will return to i a list of members of C_x including j, consequently i will send j a $change\_par$. If e receives $change\_par$ from i before receiving $join\_req$ from j, then e will return to j its view of C_x including i as the new parent and j will send i a $join\_req$. Therefore, 4.1 holds for joining children.

If j is a member of C_x but was changing parent from a node k to node f when f failed (i.e. $status_f$ = adopting). If f has completed change-parent of C_x before failing (i.e. it has sent an ack_change to the old parent k and has propagated its new row), then $C_x \in View_i.f.Child\_Cids$ and j will receive a $change\_par$ from i. If f failed while performing the operation, then the old parent k will time out not receiving an ack_change from f and will send members of C_x a $change\_par$ request to change their parent back to k. Hence, j will receive a $change\_par$ from k. Then, 4.1 holds if C_x was changing parent.

Therefore, 4.1 holds regardless of the status of f or its children.

Since i joins f's cluster, then by Theorem 2, i will receive every message a neighbour or the parent of f had and i did not have, and vice versa, by exchanging missing messages. Since i sends $change\_par$ to all f's children, then from Theorem 3, i will receive every message a child of f had and i did not have and vice versa. Therefore, if f had a message m and at least a neighbour, a child or the parent of f has m in its *Log*, then i will receive m. Since i receiving a missing message m, sends it to *all* neighbours, children and the parent, then every correspondent of f will receive m. Further, since they are receiving it from a node (i) which is in the same position as f, then they will proceed in propagating m and

m will follow the same pattern of propagation it should have followed if f was alive and consequently will be propagated everywhere and 4.2 holds.

In the case where children were changing their parent to f while f failed, the old parent k will change their parent back to k, hence, receiving any message m that f gave to any of these children before it fails. Since k sends m to *all* neighbours, children and the parent then m will be propagated everywhere and 4.2 holds which completes the proof. $\square$

Appendix B

Correctness of the Causal Order Protocol

In this appendix, it will be shown that following the protocol described in Section 5.5, causal order is preserved. Next, the correctness of the leave, join and change-parent operations with the modifications presented in Section 5.6 is established.

B.1 The Causal Order Algorithm

Theorem 5 *if a node i originates a message m' that is causally related to a previously delivered message m (i.e. $m \to m'$), then every other node in the network will deliver m then m'.*

Proof:

First it will be shown that i's neighbours, parent and children will deliver m then m'. Then, it is generalised for all nodes. In the proof, the timestamp associated with a message m stamped by node i when i sends m to its neighbours and parent is denoted by $CV_i^0(m)$ and the timestamp associated with a message m stamped by node i when i sends m to its children belonging to cluster C_y is denoted by $CV_i^y(m)$. Assume that node j has originated m and that i is the r^{th} member of cluster C_x. There are two cases:

Case 1: $i = j$ (i.e. i originated both messages)

Then, children, neighbours and the parent of i will receive both m and m' from i.

Before sending m and m' to its neighbours and parent, node i stamps m and m' with $CV_i^0(m)$ and $CV_i^0(m')$, respectively. From CV construction, $CV_i^0(m)[r] < CV_i^0(m')[r]$. Therefore, following the check for causality, a neighbour h comparing

$CV_i^0(m)$ and $CV_i^0(m')$ with CV_h^0 $(CV_i^0(m') > CV_i^0(m) > CV_h^0)$, will deliver m' only after it delivers m. Also, node p, the parent of i, comparing $CV_i^0(m)$ and $CV_i^0(m')$ with CV_p^x, will deliver m then m'.

Similarly, before sending m and m' to its children node i stamps m and m' with $CV_i^y(m)$ and $CV_i^y(m')$ respectively, for every child cluster C_y. From CV construction, $CV_i^y(m)[0] < CV_i^y(m')[0]$ $\forall y$. Therefore, following the check for causality, a child g belonging to cluster C_y, comparing $CV_i^y(m)$ and $CV_i^y(m')$ with CV_g^0 $(CV_i^y(m') > CV_i^y(m) > CV_g^0)$, will deliver m' only after it delivers m.

Case 2: $i \neq j$ (i.e. m and m' originated at two different nodes)
Assume i receives m from node k. There are three subcases:

Case 2.1: k is the parent of i
Then, node i receives m from k with stamp $CV_k^x(m)$. Since i increments $CV_i^0[r]$ on originating m' and stamps m' with CV_i^0 before sending it to its neighbours. Therefore, $CV_k^x(m)[r] < CV_i^0(m')[r]$. Since a neighbour h of i receives m from k with $CV_k^x(m)$ and m' from i with $CV_i^0(m')$, then comparing $CV_k^x(m)$ and $CV_i^0(m')$ with CV_h^0 $(CV_i^0(m') > CV_k^x(m) > CV_h^0)$, h will deliver m' only after it delivers m. Therefore, parent and neighbours of i will deliver m then m'.

Since children of i belonging to cluster C_y receive m and m' from i and since i constructs $CV_i^y(m)$ and $CV_i^y(m')$ such that $CV_i^y(m)[0] < CV_i^y(m')[0]$ then, as we argued in Case 1, children belonging to cluster C_y deliver m then m'.

Case 2.2: k is a neighbour of i
Then, node i receives m from k with stamp $CV_k^0(m)$. Since i increments $CV_i^0[r]$ on originating m' and stamps m' with CV_i^0 before sending it to its neighbours and parent. Therefore, $CV_k^0(m)[r] < CV_i^0(m')[r]$. Since the parent and neighbours of i receive m from k with $CV_k^0(m)$ and m' from i with $CV_i^0(m')$, then a neighbour h comparing $CV_k^0(m)$ and $CV_i^0(m')$ with CV_h^0 $(CV_i^0(m') > CV_k^0(m) > CV_h^0)$, will deliver m then m'. Similarly, the parent p, comparing $CV_k^0(m)$ and $CV_i^0(m')$ with CV_p^x $(CV_i^0(m') > CV_k^0(m) > CV_p^x)$, will deliver m' only after delivering m. Therefore, parent and neighbours of i will deliver m then m'.

Since children of i belonging to cluster C_y receive m and m' from i and since i constructs $CV_i^y(m)$ and $CV_i^y(m')$ such that $CV_i^y(m)[0] < CV_i^y(m')[0]$ then, as we argued in Case 1 children belonging to cluster C_y deliver m then m'.

Case 2.3: k is a child of i and k is a member of cluster C_y
Then, node i receives m from k with stamp $CV_k^0(m)$. Since node i increments $CV_i^y[0]$ on originating m' and stamps m' with CV_i^y before sending it to its children belonging to cluster C_y, then $CV_k^0(m)[0] < CV_i^y(m')[0]$. Since a child g of i belonging to cluster C_y (s.t. $g \neq k$) receives m from k with $CV_k^0(m)$ and m' from i with $CV_i^y(m')$, then following the check for causality: comparing CV_g^0 with $CV_k^0(m)$ and $CV_i^y(m')$ $(CV_i^y(m') > CV_k^0(m) > CV_g^0)$, g delivers m' only after delivering m. Therefore, children of i belonging to cluster C_y will deliver m then m'.

Since neighbours and the parent of i receive m and m' from i, and since i constructs $CV_i^0(m)$ and $CV_i^0(m')$ such that $CV_i^0(m)[r] < CV_i^0(m')[r]$, then as we argued in Case 1, neighbours and parent will deliver m then m'.

Children of i belonging to cluster $C_z \neq C_y$, receive m and m' from i. Since i constructs $CV_i^z(m)$ and $CV_i^z(m')$ such that $CV_i^z(m)[0] < CV_i^z(m')[0]$, then as we argued in Case 1, children of i belonging to cluster C_z will deliver m then m'.

Now it has been shown that correspondents of i (i.e. neighbours, parent and children of i) will deliver m then m'. A node j, a neighbour or a child of i, will continue propagating m and m' to its children such that $CV_j^y(m)[0] < CV_j^y(m')[0]$, for any child cluster C_y. Therefore, the children of i's neighbours and the children of i's children will deliver m then m'. Similarly, node p the parent of i, will continue propagating m and m' to its neighbours and parent, stamping the messages such that $CV_p^0(m)[l] < CV_p^0(m')[l]$ (assuming that i's parent is the l^{th} member of its cluster). Therefore, correspondents of the correspondents of i will deliver m then m'. This works recursively and every node in the network will deliver m then m'. □

B.2 The leave Operation

Theorem 6 *If a node i leaves cluster C_x according to the protocol described in Section 4.2.1 and the modifications described in Section 5.6.1, then*

6.1 every member of C_x and the parent of C_x receives every message i had before it leaves and will deliver it in causal order.

6.2 if a member or the parent of C_x receives a message m at the time i was leaving, then it is guaranteed that m is delivered in causal order.

Proof:

Consider a node j, a member or the parent of C_x. From Theorem 1, j is guaranteed to receive every message that i had before it leaves. Since the last message sent by i is the *update_view* message, and since j waits until all messages received from i before receiving *update_view* are delivered and then it discards i's entry from its Compact Vectors, then j will deliver all messages sent from i before *update_view* in the correct order and 6.1 holds.

Assume j is a neighbour of i. To show that 6.2 holds, two cases are considered:

Case 1 : m comes from a child of j belonging to cluster C_y.
Then, $CV(m)$ needs to be compared with CV_j^y which is not affected by the leave operation and m will be delivered correctly.

Case 2 : m comes from a node k, a neighbour or the parent of C_x.
From Theorem 1, j is guaranteed to receive any message m' that is causally related

to m and sent via i. While comparing $CV(m)$ and CV_j^0, there are three subcases:

Case 2.1 : If k has sent m and j has received m before (or after) *both* of them have discarded i's entry from their CV^0, then an entry for i exists (or does not exist) in *both* $CV(m)$ and CV_j^0. Therefore, $CV(m)$ and CV_j^0 are compatible and m will be delivered in order.

Case 2.2 : If k has sent m after discarding i's entry from CV_k^0 but j has received m before discarding i's entry from CV_j^0, then an entry of i exists in CV_j^0 but not in $CV(m)$ and m might be causally dependent on messages sent by i. Since j delays carrying out the causality check for m until i's entry is discarded from CV_j^0, then j will deliver m only after all causally related messages sent by i have been delivered (see Step (1) in Figure 5.5).

Case 2.3 : If k has sent m before discarding i's entry from CV_k^0 but j has received m after discarding i's entry from CV_j^0, then an entry of i exists in $CV(m)$ but not in CV_j^0. Since j has received and delivered every message sent by i before discarding i's entry from CV_j^0, then m cannot be causally related to a message m' sent by i and not delivered at j. Therefore, discarding the entry of i from $CV(m)$ ensures that m will not be indefinitely blocked and will be delivered in order (see Step (2) in Figure 5.5).

Similar arguments hold if j is the parent of C_x or if j is a node joining C_x. □

B.3 The join Operation

Theorem 7 *If a node i joins cluster C_x according to the protocol described in Section 4.2.2 and the modifications described in Section 5.6.1, then*

7.1 i receives every message any member of C_x or the parent of C_x had or will receive and will be delivered in causal order.

7.2 if i has any undelivered message m, then it is guaranteed that m is delivered in causal order.

7.3 every member of C_x and the parent of C_x receives every message i had or will receive and will deliver it in causal order.

7.4 if a member or the parent of C_x receives a message m at the time i was joining, then it is guaranteed that m is delivered in causal order.

Proof:

Assume j is a member of C_x and it receives *join_req* at $t = t_j$. At $t = t_j$, j sends i all messages that j has delivered and i has not, and since i delivers them in the same order they were delivered at j, then i receives all messages j has delivered at $t \leq t_j$ and will deliver them in causal order. Since at $t = t_j$ j includes i in $View_j$ as a neighbour, then for any message that j has received at $t \leq t_j$ but has not yet delivered, or any message that j generates or receives from its children at $t > t_j$ will be sent to i after it is delivered at j, through normal propagation, stamped with CV_j^0. Since i starts receiving messages from j through propagation after i has updated CV_i^0 such that it is compatible with CV_j^0, then i can apply the causal delivery condition on the message when received and it will be delivered in order.

Therefore, i receives every message a neighbour j has generated or will generate or has received or will receive from a child after it is delivered at j and will be delivered at i in causal order. Similarly, it can be shown that i receives every message the parent has generated or will generate or has received or will receive from a neighbour, parent or a child of a cluster other than C_x once it is delivered and i will deliver it in order. Therefore 7.1 holds.

Assume i has an undelivered message m at the time it was joining. If m was received from a child belonging to cluster C_y, then $CV(m)$ needs to be compared with CV_i^y which is not affected by the join operation. If m was received from a previous neighbour or parent, then m will never be delivered since $CV(m)$ is not compatible to CV_i^0 anymore and will be discarded. In this case, the members or the parent of C_x have either received m or will receive m from a path not involving i [1], and from 7.1 i will receive and deliver m in casual order. Then, 7.2 holds.

Node i, on receiving *ack_join* from a node j, a member or the parent of C_x, will send j all messages that i has delivered and j has not, which are delivered at j in the same order they were delivered at i. Any further message i generates or receives from its children or any message that i has already received but has not yet delivered will be sent to j through propagation stamped with CV_i^0 after it is delivered. Since i has updated CV_i^0 to reflect messages that have been received and delivered from members and the parent of C_x before starting propagating messages to j, then j applying the causal delivery condition will deliver m when all causally related messages are received from members or the parent of C_x and causal order is preserved. Then 7.3 holds.

Assume a neighbour j receives a message m at the time i was joining. There are two cases:

Case 1 : m comes from a child of j belonging to cluster C_y.
Then, $CV(m)$ needs to be compared with CV_j^y which is not affected by the join operation and m will be delivered in order.

Case 2 : m comes from a node k, a neighbour or the parent of C_x.
While comparing $CV(m)$ and CV_j^0, there are three subcases:

[1] Since i would have passed m to its descendants if delivered, and since members of C_x or the parent of C_x cannot be the descendants of i, then members and parent of C_x are not relying on i to pass on m to them and will receive it from a different branch of the hierarchy.

Case 2.1 : If m was sent by k and received by j after (or before) *both* of them have received *join_req* from i, then an entry for i exists (or does not exist) in *both* $CV(m)$ and CV_j^0 and m will be delivered in order.

Case 2.2 : If k has sent m before it has received *join_req* from i but j has received m after receiving *join_req* from i, then an entry of i exists in CV_j^0 but not in $CV(m)$. Since m cannot be causally related to a message sent to k by i, then adding an entry for i in $CV(m)$ with value 0 before comparing $CV(m)$ and CV_j^0 guarantees that m will not be indefinitely blocked and will be delivered in order (see Step (3) in Figure 5.5).

Case 2.3 : If k has sent m after it has received *join_req* from i but j has received m before receiving *join_req* from i, then an entry of i exists in $CV(m)$ but not in CV_j^0 and m might be causally dependent on messages sent by i. Since j delays carrying out the causality check for m until *join_req* is received from i and an entry for i is created in CV_j^0, then m will be delivered only after delivering all causally related messages received from members or the parent of C_x, including i, and causal order is preserved (see Step (4) in Figure 5.5).

Similar arguments apply if j is the parent of C_x or if j is another joining node. Therefore 7.4 holds which completes the proof. □

B.4 The change-parent Operation

Theorem 8 *If a cluster C_x change-parent from node old_p to node new_p, then*

8.1 new_p will receive every message a member of C_x had or will receive and will deliver it in causal order.

8.2 every member of C_x will receive every message new_p had or will receive and will deliver it in causal order.

8.3 if new_p or a member of C_x receives a message m while the change is taking place, it is guaranteed that m is delivered in causal order.

Proof:

Assume node i, a member of C_x, sends *ack_par* to *new_p* at $t = t_i$. Then, at $t = t_i$ i sends to *new_p* any messages i has delivered at $t \leq t_i$ and *new_p* has not. Since *new_p* delivers them in the same order they were delivered at i, then causal order is preserved. Further, at $t = t_i$, i replaces *old_p* by *new_p* in $View_i.i$, hence, any messages that i has received at $t \leq t_i$ but has not yet delivered, or messages that i generates or receives from its children at $t > t_i$ will be sent to *new_p* after it is delivered at i, through normal propagation, stamped with CV_i^0. Since *new_p* starts receiving messages from i through propagation after it has updated $CV_{new\_p}^x$ so that it reflects messages that have been received and delivered from members of C_x, then CV_i^0 and $CV_{new\_p}^x$ are compatible. Therefore, *new_p* applying the causal delivery condition on a message received from i, will deliver it only when all causally related messages are received from other members of C_x and causal order is preserved. Then 8.1 holds.

Similarly, *new_p* receiving *ack_par* from i, sends i the delivered messages at *new_p* not yet delivered at i, which are delivered at i in the same order. Further, it places i as a child in $View_{new\_p}$, hence, it will send i, through propagation, any message *new_p* has received but not delivered or any message *new_p* generates or receives from its neighbours, parent or children from clusters other than C_x after it is delivered. Since i has updated CV_i^0 so that it reflects delivered messages that have been received from *new_p*, then i can apply the causal delivery condition on messages received through propagation from *new_p* and they will be delivered in order. Therefore, 8.2 holds.

Assume *new_p* receives a message m while the change is taking place. If m comes from either *new_p*'s neighbours, parent or from children of another cluster C_y, then $CV(m)$ needs to be compared with either $CV_{new\_p}^0$ or $CV_{new\_p}^y$ which are not affected by the change-parent operation and m will be delivered in order. If m comes from a member of C_x, than as shown above, (Part 8.1), m will be delivered in order.

Assume node i, a member of C_x, receives a message m while the change is taking place. There are two cases:

Case 1 : m comes from a child belonging to cluster C_z.
Then $CV(m)$ needs to be compared with CV_i^z which is not affected by the change-parent operation and m will be delivered in order.

Case 2 : m comes from *old_p*.
Since *old_p* waits until all delivered messages are received and acknowledged by all children before issuing the change_parent, then i is guaranteed to receive any message m' that is causally related to m and sent via *old_p* before the change starts. Further, since i waits until m is delivered before resetting $CV_i^0[0]$ to 0, then $CV(m)$ is compatible with CV_i^0 and m will be delivered in the correct order.

Case 3 : m comes from a neighbour j.
While comparing $CV(m)$ and CV_i^0, there are three subcases:

Case 3.1 : If m was sent by j and received by i before (or after) *both* of them have reset their $CV^0[0]$ to 0, then and an entry for *old_p* (or *new_p*) exists in both $CV(m)$ and CV_i^0. So, $CV(m)$ and CV_i^0 are compatible and m will be delivered correctly.

Case 3.2 : If j has sent m after resetting $CV_j^0[0]$ to 0 but i has received m before resetting $CV_i^0[0]$ to 0, then the first entry in $CV(m)$ is for *new_p* while that of CV_i^0 is for *old_p* and m might be causally dependent on messages sent by *new_p*. Since i delays carrying out the causality check for m until *change_par* is received and $CV_i^0[0]$ reset to 0, then m will be delivered only after delivering any messages that are causally related to m that came from members of C_x or *new_p* (see Step (1) in Figure 5.7).

Case 3.3 : If j has sent m before resetting $CV_j^0[0]$ to 0 but i has received m after resetting $CV_i^0[0]$ to 0, then the first entry in $CV(m)$ is for *old_p* while that of CV_i^0 is for *new_p*. Since i has delivered all messages from *old_p*, then m cannot be causally related to a message m' sent by *old_p* and not delivered at i. Therefore, setting $CV(m)[0]$ to 0 before comparing $CV(m)$ and CV_i^0 ensures that m will not be indefinitely blocked and will be delivered in order (see Step (2) in Figure 5.7).

Case 4 : m comes from *new_p*.

From 8.2 m is delivered at i in causal order.

Therefore 8.3 holds and this completes the proof. □

Appendix C

Correctness of HPP

This appendix establishes the correctness the the Hierarchical Propagation Protocol presented in Chapter 8. First it shows that causal order is achieved during normal propagation. Then, it shows that causal order is preserved while reorganisation is taking place. Finally, it shows that if a reorganisation occurs, no messages are lost even in the presence of failures.

Definition 2 *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 2 *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 organised in a tree hierarchy, messages from node i to node j must take the same, unique path. □

Theorem 9 *If every node observes* H_FIFO *order while propagating messages, then causal order is achieved provided the tree hierarchy is not reorganised and no node failures occur.*

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

Case 1: $i = j$

From Corollary 2, 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 H_FIFO order, it follows that all nodes will see m_1 before m_2.

Case 2: $i \neq j$

Consider S, the set of nodes along the unique path (see Corollary 2) from node i to node j. For any node $k \in S$, at $t = t_1$ k has already received m_1, and therefore, it will see m_2 after m_1. Any node $k \notin S$ will receive both m_1 and m_2 from a node y, where $y \in S$. From Corollary 2, the path between k and y is unique. Since every node along a path processes messages in H_FIFO order, k will see m_2 after m_1.

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

Theorem 10 *If no node failures occur, then causal order is preserved while reorganisation takes place.*

Proof: The proof of causality in Theorem 9 was based on there being a unique path between any two nodes. The causal order could conceivably be violated due to the fact that while the change is taking place, messages sent from node i may follow different paths to reach *new_p* or *old_p* and vice versa. The proof is based on showing that even if messages follow different paths:

1) nodes *new_p* and *old_p* will still receive messages sent by i in their correct order, and
2) node i will still receive messages sent by *new_p* or *old_p* in their correct order.

Case 1: For messages received by *new_p* and *old_p* from i:

Assume node i sends a message m_1 to *old_p* for propagation at t_0, and then sends the control message *last_msg* at t_1. Subsequently, it sends a message m_2 to *new_p* at t_2 ($t_0 < t_1 < t_2$). *new_p* receives m_1 and m_2 through different paths: m_1 is received through *old_p*, while m_2 is received directly from i. Similarly, *old_p* receives m_1 directly from i while m_2 is received through *new_p*. We shall show that both *new_p* and *old_p* process m_1 before m_2. Since the *last_msg* propagates through the network like a normal message, from Theorem 9, both *new_p* and *old_p* must receive m_1 before the *last_msg*. Since *new_p* will start processing messages received directly from i only after it receives *last_msg*, *new_p* will process m_2 after m_1. Moreover, since *old_p* will receive m_2 through *new_p*, it follows that *old_p* will receive m_2 after m_1.

Case 2: For messages received by i from *new_p* or *old_p*:

Assume *new_p* sends a message m_1 for propagation at t_0, then it receives a *join_req1* and sends a *join_ack1* 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 i receives the messages via different paths (i.e., m_1 from *old_p* and m_2 from *new_p*), it will still process m_2 after m_1. Since i receives both m_1 and *join_ack1* from *old_p*, from Theorem 9, i must receive m_1 before the *join_ack1*. Since *new_p* will send m_2 to i only after *new_p* receives *join_req2*; and since i will send *join_req2* only after it receives *join_ack1*; then *new_p* will send m_2 only after i has processed m_1. The same argument holds for messages sent to i by *old_p* while the change is taking place. □

Theorem 11 *If a node i changes its parent from old_p to new_p, then it is guaranteed that no messages are lost as a result of the move, even if a failure occurs along the path from old_p to new_p while the reorganisation takes place.*

Proof: The proof is based on arguing that *old_p* and *new_p* will receive *all* messages i receives from below, and that i will receive *all* messages that *new_p* has received before the move or will receive after the move.

Node new_p: Assume i sends *leave_req* at time t_1 and receives *join_ack2* at time t_2. Every message generated by i or received from below at $t_1 \leq t \leq t_2$ will be buffered by i in $M\_Log_{new\_p}$. Since i will send the log to *new_p* at time t_2 and will then start sending messages directly to *new_p*, this means that *new_p* will receive directly from i every message received by i from below at time $t \geq t_1$. All messages received by i at $t < t_1$ from below are received by *old_p* directly from i, and *old_p* propagates them to *new_p* by the normal propagation mechanism. If a failure occurs along the path from *old_p* to *new_p* while messages are propagating, the transition and diffusion phases guarantee that no messages will be missed by *new_p*. Therefore, *new_p* receives *all* messages received by i from below.

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

Node i: Assume *new_p* receives i's *join_req1* at time t_1. We need to show that

(1) i receives every message received by *new_p* at $t \geq t_1$

(2) i receives every message received by *new_p* at $t < t_1$

Assume *new_p* receives i's *join_req2* at time t_2 ($t_1 < t_2$). Since *new_p* will buffer every message generated or received at $t \geq t_1$ and will send the log at $t = t_2$; and since it adds i as a correspondent at $t = t_2$, it follows that i will receive every message received by *new_p* at $t \geq t_1$ directly from *new_p*. To prove (2), we need to show that any message that *new_p* has received at $t < t_1$ will also be received by i through *old_p*. Assume that *new_p* has received a message m at $t < t_1$. Since *new_p* generates the *join_ack1* at $t = t_1$ and since i will still be receiving messages from *old_p* until i receives *join_ack1*, we need to show that *old_p* (and consequently i) will receive m before receiving *join_ack1*. Three cases can occur while m and *join_ack1* are propagating:

Case 1: while m and *join_ack1* are propagating, there were no failures nor reorganisation all along the path to *old_p*. Then, from Theorem 9, *old_p* will receive m before receiving the *join_ack1*.

Case 2: while m and *join_ack1* are propagating, a failure occurs in their path of propagation towards *old_p*. If the failure occurs after m reaches *old_p*, then whether *join_ack1* is sent during the *transition* phase or the *diffusion* phase, it will be received by *old_p* after m anyway. If the failure happens before m reaches *old_p*, there are four subcases:

Case 2.1: m and *join_ack1* are diffused by the *transition algorithm*
If *old_p* was one of the failed node's correspondents, then, upon sorting m and *join_ack1*, *join_ack1* will always come last because it is a control message. This guarantees that *old_p* will process m first, send it to i, and then process *join_ack1*. If *old_p* was not one of the failed node's correspondents, then *old_p* will receive m and *join_ack1* through normal propagation. Since each correspondent of the failed node sorts messages (such that *join_ack1* comes last) before processing and further propagating them, and since they maintain the H_FIFO order in propagation, then *old_p* will receive m first, and then *join_ack1*.

Case 2.2: m and *join_ack1* are sent during the *diffusion phase*.
Since the coordinator CO becomes a correspondent of the failed node's correspondents and each node maintains H_FIFO order, then from Theorem 9 *old_p* will receive m before *join_ack1*.

Case 2.3: m is diffused by the *transition algorithm* while *join_ack1* is sent during the *diffusion phase*.
Since the diffusion phase starts only after the transition phase is over, m will precede *join_ack1* in propagation.

Case 2.4: *join_ack1* is diffused by the *transition algorithm* while m is sent during the *diffusion phase*.
We will show that this case cannot occur. Assume that the failure occurs at time t_f. For this case to occur, at time $t < t_f$, one or more correspondents of the failed node should have received the *join_ack1* but not m. Again, from Theorem 9, this is not possible at $t \leq t_f$, and, therefore, there is a contradiction.

Case 3: while m and *join_ack1* are propagating, a reorganisation takes place.
From Theorem 10, since the causal order is preserved while a reorganisation occurs, *old_p* receives m before receiving *join_ack1*. $\square$

Bibliography

[Adly 93a] N. Adly. HARP: a hierarchical asynchronous replication protocol for massively replicated data. Technical Report 310, Computer Laboratory, University of Cambridge, August 1993.

[Adly 93b] 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*, pages 152-157, Princeton, NJ, October 1993.

[Adly 95a] N. Adly and M. Nagi. Maintaining causal order in large scale distributed systems using a logical hierarchy. In *Proceedings of the 12th IASTED International Conference on Applied Informatics*, pages 214-219, Innsbruck, Austria, February 1995.

[Adly 95b] N. Adly, J. Bacon and M. Nagi. Performance evaluation of a hierarchical replication protocol: synchronous versus asynchronous. In *Proceedings of the IEEE Second International Workshop on Services in Distributed and Networked Environments (SDNE'95)*, pages 102-109, Whistler, Canada, June 1995.

[Adly 95c] N. Adly and A. Kumar. HPP: a hierarchical propagation protocol for large scale replication in wide area networks. In *Proceedings of Seventh International Conference of Computing and Information (ICCI)*, pages 237-257, Peterborough, Ontario, Canada, July 1995.

[Adly 95d] N. Adly. Performance evaluation of HARP: a hierarchical asynchronous replication protocol for large scale systems. Technical Report 378, Computer Laboratory, University of Cambridge, August 1995.

[Agrawal 90] D. Agrawal and A. El-Abbadi. The Tree Quorum Protocol: an efficient approach for managing replicated data. In *Proceedings of the IEEE 16th International Conference on VLDB*, Brisbane, pages 243-254, August 1990.

[Agrawala 92] A. Agrawala and D. Sanghi. Networks dynamics: an experimental study of the Internet. In *Proceedings of the IEEE GLOBECOM'92*, pages 782-786, December 1992.

[Alonso 88] R. Alonso, D. Barbara and H. Garcia-Molina. Quasi-copies: efficient data sharing for information retrieval systems. In *Proceedings of the Second International Conference on Extending Data Base Technology*, Venice, Italy, March 1988.

[Alonso 90] R. Alonso, D. Barbara and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM TODS*, 15(4), September 1990.

[Amir 92] Y. Amir, D. Dolev, S. Kramer and D. Malki. Transis: a communication subsystem for high availability. In *Proceedings of the IEEE 22nd International Symposium on Fault Tolerant Computing*, July 1992.

[Ammar 91] M. Ammar and G. Rouskas. On the performance of protocols for collecting responses over a multiple-access channel. In *Proceedings of the IEEE INFOCOM*, Florida, pages 1490-1499, April 1991.

[Barbara 90] D. Barbara and H. Garcia-Molina. The case for controlled inconsistency in replicated data. In *Proceedings of the First Workshop on Management of Replicated Data*, Houston, Texas, pages 35-38, November 1990.

[Barbara 92] D. Barbara and H. Garcia-Molina. The demarcation protocol: a technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the Third International Conference on Extending Database Technology (EDBT)*, pages 373-388, Vienna, Austria, March 1992.

[Bernstein 84] P. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM TODS*, 9(4):596-615, December 1984.

[Bernstein 87] P. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[Birman 87] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):272-314, February 1987.

[Birman 91] K. Birman, A. Schiper and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):47-76, August 1991.

[Bowman 94] C. Bowman, P. Danzig, U. Manber and M. Schwartz. Scalable Internet resource discovery: research problems and approaches. *Communications of the ACM*, 37(8):98-114, August 1994.

[Carey 87] M. Carey, B. Lindsay and R. Obermarck. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Transactions on Software Engineering*, 13(12):1348-1363, December 1987.

[Carey 91] M. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM TODS*, 16(4):703-746, December 1991.

[Ceri 91] S. Ceri, M. Houtsma, A. Keller and P. Samarati. A classification of update methods for replicated databases. Technical Report STAN-CS-91-1392, Stanford University, October 1991.

[Chan 86] A. Chan and D. Skeen. The reliability subsystem of a distributed database manager. Technical Report CCA-85-02, Computer Corporation of America, 1986.

[Chang 84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computing Systems*, 2(3):251-273, August 1984.

[Cheriton 85] D. Cheriton and W. Zwaenepoel. Distributed process group in the V Kernel. *ACM Transactions on Computer Systems*, 3(2):77-107, May 1985.

[Cheung 90] S. Cheung, M. Ammar and M. Ahamad. The Grid protocol: a high performance scheme for maintaining replicated data. In *Proceedings of the IEEE Sixth International Conference on Data Engineering*, pages 438-445, 1990.

[Cheung 94] S. Cheung and A. Kumar. Efficient quorumcast routing algorithms. In *Proceedings of the IEEE INFOCOM-94*, pages 840-847, June 1994.

[Ciciani 90] B. Ciciani, D. Dias and P. Yu. Analysis of replication in distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):247-261, June 1990.

[Clark 89] D. Clark, V. Jacobson, J. Romkey and H. Salwen. An analysis of TCP processing overhead. *IEEE Communication Magazine*, June 1989.

[Cristian 91] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 6(4), 1991.

[Davcev 85] D. Davcev and W. Burkhard. Consistency and recovery control for replicated files. In *Proceedings of the ACM Tenth Symposium on Operating Systems Principles*, pages 87-96, 1985.

[Davcev 89] D. Davcev. A dynamic voting scheme in distributed systems. *IEEE Transactions on Software Engineering*, 15(1):93-97, January 1989.

[Demers 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 ACM Sixth Symposium on Principles of Distributed Computing, SIGACT-SIGOPS*, pages 1-12, August 1987.

[Dine 94] O. Zein El Dine. *Atomic broadcast in heterogeneous distributed systems.* PhD thesis, Old Dominion University, September 1994.

[Downing 90a] A. Downing, I. Greenberg and J. Peha. OSCAR: an architecture for weak consistency replication. In *Proceedings of the IEEE PARABASE-90*, pages 350-358, March 1990.

[Downing 90b] A. Downing, I. Greenberg and J. Peha. OSCAR: a system for weak consistency replication. In *Proceedings of the First Workshop on Management of Replicated Data*, Houston, Texas, pages 26-30, November 1990.

[El-Abbadi 85] A. El-Abbadi, D. Skeen and F. Christian. An efficient fault-tolerant protocol for replicated data management. In *Proceedings of the Fourth ACM SIGACT-SIGMOD, Symposium on Principles of Database Systems*, pages 215-228, Portland, OR, March 1985.

[El-Abbadi 86] A. El-Abbadi and S. Toueg. Availability in partitioned replicated databases. In *Proceedings of the Fifth ACM SIGACT-SIGMOD, Symposium on Principles of Database Systems*, pages 240-251, March 1986.

[El-Abbadi 89] A. El-Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM TODS*, 14(2):264-290, June 1989.

[Emtage 92] A. Emtage and P. Deutsch. Archie – An electronic directory service for the Internet. In *Conference Proceedings Usenix*, San Francisco, CA, January 1992.

[Ferrari 78] D. Ferrari. *Computer systems performance evaluation.* Prentice Hall, 1978.

[Fidge 88] C. Fidge. Timestamps in message-passing systems that preserve the partial order. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56-66, 1988.

[Fishwick 92] P. Fishwick. Simpack: getting started with simulation programming in C and C++. In *Proceedings of the Winter Simulation Conference*, pages 154-162, Arlington, VA, December 1992.

[Fowler 90] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proceedings of the IEEE Tenth International Conference on Distributed Computing Systems*, pages 134-141, May 1990.

[Garcia-Molina 82] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48-59, January 1982.

[Garcia-Molina 91] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242-271, August 1991.

[Gifford 79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the ACM Seventh Symposium on Operating System Principles*, December 1979.

[Golding 91] R. Golding and D. Long. Accessing replicated data in large scale distributed systems. *International Journal in Computer Simulation*, 1(2), June 1991.

[Golding 92a] R. Golding and D. Long. Quorum oriented multicast protocols for data replication. In *Proceedings of the IEEE Eighth International Conference on Data Engineering*, June 1992.

[Golding 92b] R. Golding. A weak-consistency architecture for a distributed information services. Technical Report UCSC-CRL-92-31, University of California, Santa Cruz, July 1992.

[Golding 92c] R. Golding. *Weak-consistency group communication and membership.* PhD thesis, University of California, Santa Cruz, December 1992.

[Goodman 83] N. Goodman et al. A recovery algorithm for a distributed database system. In *Proceedings of the Second ACM SIGACT-SIGMOD, Symposium on Database Systems*, pages 8-15, Atlanta, GA, March 1983.

[Gray 88] J. Gray. The cost of messages. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 1-7, August 1988.

[Heddaya 89] A. Heddaya, M. Hsu and W. Weihl. Two Phase Gossip: managing distributed event histories. *Information Sciences*, 49(1):35-57, January 1989.

[Herlihy 87] M. Herlihy. Concurrency vs. availability: atomicity mechanisms for replicated data. *ACM Transactions on Computer Systems*, 5(3):249-274, August 1987.

[Hwang 88] D. Hwang. Constructing a highly-available location service for a distributed environment. Technical Report MIT/LCS/TR-410, MIT Laboratory for Computer Science, Cambridge, MA, January 1988.

[Jajodia 87] S. Jajodia and D. Mutchler. Dynamic voting. In *Proceedings of the ACM International Conference on Management of Data*, pages 227-238, 1987.

[Jajodia 90] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining consistency of a database. *ACM TODS*, 15(2), June 1990.

[Kaashoek 89] M. Kaashoek, A. Tanenbaum, S. Hummell and H. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5-19, October 1989.

[Kahle 91] B. Kahle and A. Medlar. An information system for corporate users: wide area information servers. Technical Report TMC-199, Thinking Machines Corporation, April 1991.

[Kantor 86] 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.

[Kistler 92] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[Krishnakumar 92] N. Krishnakumar and A. Bernstein. High throughput escrow algorithms for replicated databases. In *Proceedings of the IEEE 18th International Conference on VLDB*, pages 175-186, Vancouver, Canada, September 1992.

[Kumar 88] A. Kumar and M. Stonebraker. Semantic based transaction management techniques for replicated data. In *Proceedings of the ACM SIGMOD Conference, Chicago*, June 1988.

[Kumar 90] A. Kumar. An analysis of borrowing policies for escrow transactions in a replicated environment. In *Proceedings of the IEEE Sixth International Conference on Data Engineering*, pages 446-454, 1990.

[Kumar 91] A. Kumar. Hierarchical Quorum Consensus: a new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9):996-1004, September 1991.

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

[Ladin 89] R. Ladin. *A method for constructing highly available services and an algorithm for distributed garbage collection.* PhD thesis, MIT Laboratory for Computer Science, Cambridge, MA, 1989.

[Ladin 90] R. Ladin, B. Liskov and L. Shrira. Lazy replication: exploiting the semantics of distributed services. In *Proceedings of the ACM Ninth Symposium on Principles of Distributed Computing*, pages 43-57, Quebec City, CA, August 1990.

[Ladin 92] 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.

[Lamport 78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[Lamport 82] L. Lamport, R. Shostak and M. Pease. The Byzantine general problem. *ACM Transactions on Programming Languages and Systems*, 4(3), July 1982.

[Lamport 89] L. Lamport. The part-time parliament. Technical Report 49, DEC System Research Centre, 1989.

[Lampson 86] B. Lampson. Designing a global name service. In *Proceedings of the ACM Fifth Symposium on Principles of Distributed Computing, SIGACT-SIGOPS*, pages 1-10, Vancouver, CA, August 1986.

[Lazowska 84] E. Lazowska, J. Zahorjan, G. Graham and K. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*, Prentice-Hall, 1984.

[Liskov 86] B. Liskov and R. Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the ACM Fifth Symposium on Principles of Distributed Computing, SIGACT-SIGOPS*, pages 29-39, Vancouver, CA, August 1986.

[Liskov 91] B. Liskov et al. Lazy replication: exploiting the semantics of distributed systems. *ACM Operating Systems Review*, 25(1), January 1991.

[Little 90] M. Little and A. Shrivastava. Replicated k-resilient objects in Arjuna. In *Proceedings of the First Workshop on Management of Replicated Data*, pages 53-58, Houston, Texas, November 1990.

[Long 92] D. Long. A replicated monitoring tool. In *Proceedings of the Second Workshop on the Management of Replicated Data*, Monterey, California, November 1992.

[Ma 92] C. Ma. *Designing a universal name service.* Phd thesis, Computer Laboratory, Cambridge University, 1992.

[Macedo 92] R. Macedo, P. Ezhilchelvan and S. Shrivastava. Implementing robust multicast protocols using causal blocks. Technical Report, CS Dept, University of Newcastle upon Tyne, 1992.

[Mattern 89] F. Mattern. Time and global states in distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, North-Holland, Amsterdam, 1989.

[Melliar-Smith 90] P. Melliar-Smith, P. Moser and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17-25, January 1990.

[Mishra 89] S. Mishra, L. Peterson and R. Schlichting. Implementing fault-tolerant replicated objects using Psync. In *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, pages 42-52, Seattle, WA, October 1989.

[Obraczka 93] K. Obraczka, P. Danzig and S.H. Li. Internet resource discovery services. *IEEE Computers*, 26(9):8-22, September 1993.

[Obraczka 95] K. Obraczka. *Massively replicated services in wide-area internetworks.* PhD thesis, University of Southern California, 1995.

[O'Neil 86] P. O'Neil. The escrow transactional model. *ACM TODS*, 11(4):405-430, December 1986.

[Oppen 83] D. Oppen and Y. Dalal. The Clearinghouse: a decentralised agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems*, 1(3), July 1983.

[Peterson 89] L. Peterson, N. Buchholz and R. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217-246, August 1989.

[Pu 91a] 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.

[Pu 91b] C. Pu and A. Leff. Epsilon-Serializability. Technical Report CUCS-054-90, Columbia University, New York, NY 10027, January 1991.

[Pu 91c] C. Pu, F. Korz and R. Lehman. A measurement methodology for wide area networks. Technical Report CUCS-044-90, Columbia University, New York, NY 10027, March 1991.

[Queinnec 93] P. Queinnec and G. Padiou. Flight plan management in a distributed air traffic control system. In *Proceedings of the International Symposium on Autonomous Decentralised Systems*, Kawasaki, Japan, March 1993.

[Rabinovich 92] M. Rabinovich and E. Lazowska. Improving fault tolerance and supporting partial writes in structured coterie protocols for replicated objects. In *Proceedings of the ACM SIGMOD*, pages 226-235, June 1992.

[Raynal 91] M. Raynal, A. Schiper and S. Toueg. The causal order abstraction and a simple way to implement it. *Information Processing Letters*, 39:343-350, 1991.

[Raynal 95] M. Raynal and M. Singhal. Logical time: a way to capture causality in distributed systems. Technical Report TR-900, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes, France, January 1995.

[Renesse 93] R. van Renesse. Causal controversy at le Mont St.-Michel. *ACM Operating Systems Review*, 27(2), April 1993.

[Renesse 95] R. van Renesse, K. Birman, B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd and W. Vogels. Horus: a flexible group communications system. *Submitted to the Symposium on Operating Systems Principles*, 1995.

[Ricciardi 91] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. Technical Report TR91-1188, Department of Computer Science, Cornell University, February 1991.

[Satyanarayanan 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.

[Schiper 89] A. Schiper, J. Eggli and A. Sandoz. A new algorithm to implement causal ordering. In *Distributed Algorithms, Lecture Notes in Computer Science*, volume 392, pages 219-232, 1989.

[Schlichting 83] 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.

[Schroeder 84] M. Schroeder, A. Birrell and R. Needham. Experience with Grapevine. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[Schwarz 94] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, (7):149-174, July 1994.

[Singhal 92] M. Singhal and A. Kshemkalyani. An efficient implementation of Vector Clocks. *Information Processing Letters*, 43, Amsterdam, North-Holland, 1992.

[Son 88] S. Son. Replicated data management in distributed database systems. *ACM SIGMOD Record*, 17(4):62-69, December 1988.

[Stonebraker 79] 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.

[Tait 92] C. Tait and D. Duchamp. An efficient variable-consistency replicated file service. In *Usenix File System Workshop Proceedings*, Michigan, May 1992.

[Terry 95] D. Terry. Towards a quality of service model for replicated data access. In *Proceedings of the IEEE Second International Workshop on Services in Distributed and Networked Environments (SDNE'95)*, pages 118-121, Whistler, Canada, June 1995.

[Thomas 79] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM TODS*, 3(3):180-209, June 1979.

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

[Wuu 84] G. Wuu and A. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the ACM Third Symposium on Principles of Distributed Computing*, pages 233-242, August 1984.

[Zhang 94] Y. Zhang. *Communication experiments for distributed transaction processing – from LAN to WAN*. PhD thesis, Purdue University, December 1994.