

Policy Driven Replication

by

Dmitry D. Brodsky

B.Math. University of Waterloo, 1997

M.Sc. University of Alberta, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University of British Columbia

February 2005

© Dmitry D. Brodsky, 2005

Abstract

From the inception of digital storage, ensuring that data is not lost due to user error, malicious acts, and hardware failure has always been, and still remains, a challenging open problem. This problem is exacerbated by the exponential increase in storage capacity, the proliferation of new digital media, and our growing reliance on digital storage. Today, a typical user stores financial and medical records, music and movie libraries, photo albums, etc, the loss of some of which can be catastrophic.

The advent of large robust networks has made it possible to replicate data on remote hosts to protect data from loss. Unfortunately, the growth of network bandwidth is far outstripped by both the growth of storage capacity and our ability to fill it. Thus, most replication systems that uniformly replicate all the data are incapable of protecting the ever increasing amount of data.

One important observation is that not all data is created equal. Data such as commercial music and movie libraries can be, given time, rebuilt. Data such as personal, health, and financial records, are much more difficult to reconstruct. Since resources such as network bandwidth are limited, they should be used to protect the important data.

In this thesis we propose a Policy Driven Replication (PDR) system that prioritizes data replication according to user-defined policies that specify what data is to be protected, from what failures, and to what extent. By prioritizing what data is replicated, the system conserves limited resources and protects high-priority data from high-probability failures.

PDR is a userlevel process that hooks into the file system. It is notified of file creation and modification events, and replicates the data to the hosts specified in the file's policy. In addition, the replica nodes specified in the policy are monitored for liveness to ensure the policy is followed.

PDR provides a model to describe replica nodes and a generic plug-in interface that facilitates the creation of appropriate user interfaces to manage replication policies and to translate these policies into a set of replica nodes. Replica node selection is sensitive to the system topology so that hotspots and message storms are not created.

Contents

Abstract	iii
Contents	v
List of Tables	ix
List of Figures	xi
Acknowledgements	xiii
1 Introduction	1
1.1 Motivation	2
1.2 Issues	3
1.3 PDR overview	6
1.4 Thesis	8
1.5 Contributions	9
1.5.1 Replication policies	9
1.5.2 Replica node selection	10
1.5.3 The PDR prototype	11
1.6 What follows	12
2 Background	13
2.1 Nomenclature	14
2.2 Storage system types	14
2.2.1 Block	15
2.2.2 File system	15
2.2.3 Object store	15
2.2.4 Backup	16
2.2.5 Summary of storage system types	16
2.3 Interfaces of storage systems	17
2.3.1 Application	17
2.3.2 Stub	18
2.3.3 Inter-position	18

2.3.4	Device	19
2.3.5	File systems	19
2.4	The taxonomy of replication	20
2.4.1	Consistency	20
2.4.2	Granularity	24
2.4.3	Replica node selection	26
2.4.4	Replication strategies	28
2.4.5	Update responsibility	30
2.5	Summary of replication systems	31
3	Selective replication	33
3.1	Design overview	33
3.2	Replication policies	36
3.3	Communication manager	38
3.3.1	Sending messages	38
3.3.2	Receiving messages	39
3.4	Replicator	40
3.4.1	File system controller	41
3.4.2	Replication control	41
3.4.3	Replica store	43
3.5	Policy oracle	45
3.6	Normal operation	47
3.6.1	Client operations	47
3.6.2	Replica operations	50
3.6.3	Policy oracle operations	52
3.7	Failure operations	55
3.7.1	Transient errors	56
3.7.2	Stale replication policy data	56
3.7.3	Client failure	58
3.7.4	Client failure to contact a replica	59
3.7.5	Replica failure	59
3.8	Data recovery	61
4	Replication policy	65
4.1	Overview	66
4.2	Node attributes	66
4.3	Node selection constraints	69
4.4	Policy specifier	70
4.5	Policy plug-in	70
4.5.1	Plug-in example	71
4.6	User interface	71

4.7	Highlevel replication policies	72
4.7.1	ptool	73
4.8	Translating policies	73
4.9	Remapping node attributes	77
5	Topology aware replica node selection	79
5.1	Overview	80
5.2	Algorithm	82
5.2.1	Perspective	82
5.2.2	Definitions	84
5.2.3	Policy creation	85
5.2.4	Node replacement	90
5.2.5	Periodic maintenance	94
6	Evaluation	95
6.1	Micro-benchmarks	96
6.1.1	Redirector overhead	98
6.2	Replica node selection	98
6.2.1	Simulation	99
6.2.2	Metrics	100
6.2.3	The three algorithms	101
6.2.4	Scalability	104
6.2.5	Policy creation	107
6.2.6	Splitting and Periodic maintenance	110
6.2.7	Comparison to DHTs	112
6.3	Communication costs	113
6.3.1	Propagating updates	114
6.3.2	Heartbeat mechanism	114
6.3.3	New policy creation	115
6.3.4	Policy Update	116
6.3.5	Failure messages	116
6.3.6	Summary	117
6.4	Applicability	117
6.4.1	Department of Computer Science, UBC	117
6.4.2	Silicon Chalk	120
7	Conclusion and future work	123
7.1	Conclusion	123
7.1.1	Summary	124
7.2	Contributions	126
7.3	Future work	126
7.3.1	Further improvements	127

7.3.2	Future enhancements	127
7.3.3	Open questions	128
	Bibliography	131

List of Tables

- 2.1 Storage system types 17
- 2.2 The replication taxonomy 20
- 2.3 Storage systems 32

- 4.1 Property types 68
- 4.2 Constraint function $\phi()$ 69

- 6.1 Micro-benchmarks 96
- 6.2 Communication overhead 117
- 6.3 File system trace 119

List of Figures

3.1	PDR overview	34
3.2	Structure of PDR	35
3.3	Metadata - replication policy	37
3.4	Communication manager	38
3.5	PDR replicator	40
3.6	PDR replication control	42
3.7	PDR replica store	44
3.8	PDR policy oracle	45
3.9	Policy oracle data structures	46
4.1	Default node attributes	67
4.2	Node attribute	68
4.3	Node selection constraints	69
4.4	Policy specifier	70
4.5	ptool	73
4.6	ptool node selection constraint	74
4.7	ptool $\phi()$ functions	74
4.8	ptool policy specifier	74
4.9	Slider interface policy specifier	75
5.1	Random verses TopSen	81
5.2	Independent node selection	83
5.3	Cut vertex	84
5.4	Replication policy	85
5.5	Policy creation - phase I	86
5.6	Policy creation - phase II	87
5.7	Policy creation - phase III	88
5.8	Replica node replacement - phase I	90
5.9	Replica node replacement - phase II	92
5.10	Replica node replacement - phase III	93
5.11	Periodic maintenance algorithm	94
6.1	Node degree verses Time	101

6.2	Node membership in cliques verses Time	103
6.3	Number of components verses Time	103
6.4	Node degree and system size with constant load	104
6.5	Node degree and system size with proportional load	105
6.6	Node membership in cliques and system size with constant load	106
6.7	Node membership in cliques and system size with proportional load	106
6.8	Number of components and system size with constant load	107
6.9	Node degree and policy creation frequency	108
6.10	Node membership in cliques and policy creation frequency	109
6.11	Number of components and policy creation frequency	109
6.12	Node degree and benefits of splitting	110
6.13	Node membership in cliques and benefits of splitting	111
6.14	Number of components and benefits of splitting	112
6.15	Distributed hash tables	113

Acknowledgements

You can fight
Without ever winning
But never ever win
Without a fight

— *Rush*

Coffee should be black as hell,
strong as death,
and as sweet as love.

— Turkish proverb

The research presented in this thesis is not the work of a single person. It is the embodiment of a great many contributions of a great number of people. Their thoughts, ideas, and criticisms provide the glue and cohesiveness for this work. Without their support and participation this research would not be possible. I want to thank my supervisors Norm Hutchinson and Mike Feeley. Norm for his abundant amount patience, insight, and encouragement. Mike for his uncanny ability to take on a birds-eye view of any situation. Both, for their guidance through the world of operating systems, for continuously challenging my ideas, forcing me to hone, focus, and improve them into solid pieces of research, and for making me a better researcher.

To two great friends, Matt and Bettina, thank you for your never ending support and friendship throughout the years. Thank you for all the dinners, the port (Taylor Fladgate), and the accompanying conversations. You two are amazing idea generators, forever cutting edge, never boring, always stimulating. You have opened my eyes to the big world out there and for that I am forever grateful!

To my ultimate friend Alex, thank you for being you. Thank you for always being there, for allowing me to bounce my ideas of someone, and for always being able to blow off steam — sorry for all the lumps and bruises, I know you had more than your fair share. Thank you for all your ideas, all your input, your energy as a proof reader, and many other innumerable things. You are a model friend, brother, and researcher. Thank you!

The DSG is a wonderful place to work in. It is a special place where someone is always ready to answer your questions, proof read your papers, or just simply chat. Thank you Joon, Yvonne, Chamath, Geoffrey, Kan, and the rest of the DSG folk for creating such a great environment that is conducive for both work and play.

I would like to say a big big thank you to the faculty and staff in the Department of Computer Science for making my five and some years most enjoyable. Thank you to my fellow grads and friends, Andrea, Steph, Martin, Lisa, Brian, Karyn, Andrew, Asher, Micheline and many more for your friendship. And to the faculty, Alan, Buck, Wolfgang, Gail, Will, and others; thank you for all your sage advice and your guidance during my time at UBC.

Since September of 1999 my home away from home has been 3843 and the orphanage. The time spent at “The House” has been fantastic bar none. I would like to thank Adrian, Goose, Jordan, Paul, Florence, and Veronique for a very special and magical time during the Fall and Winter of 2001/2002. Thank you for the movie nights, the dinners, and the house discussions that went into the wee hours of the night, those evenings shall forever be cherished. And to the recent orphan clan, Ajanta, Asher, Dan, Dave, Dustin, and Scott: first and foremost thank you for your friendship. Thank you Ajanta and Asher for the chats, and to the rest, thank you for the various and always captivating, dinner and late night, conversations, and especially for making me feel “young” again!

Finally, and most importantly I thank my family for their continuous support and their never failing belief in me. Thank you to my parents, Laris and Boris, to my grandmother, Bella, to my twin, Alex, and to everyone else for their love, support, and faith that I would finish one day.

Over many lunchroom conversations the question of “Will I ever finish” has been discussed an uncountable number of times, from many different angles. I too have participated in these discussions and was skeptical at various points of my graduate career. Now, having come to the end, I can say in hindsight that there is a light at the end of the tunnel. We all get there, we would not be here, now, otherwise.

I would like to dedicate this thesis to my grandfather, a person who always listened, gave sound advice, and was always there. You are greatly loved and dearly missed.

DMITRY D. BRODSKY

The University of British Columbia
February 2005

Chapter 1

Introduction

The price of reliability is the pursuit of the utmost simplicity.
It is a price which the very rich find the most hard to pay.

— Sir Antony Hoare, 1980

The primary motivation for the *Policy Driven Replication* (PDR) system is based on several assumptions. First, backup is a tedious, a cumbersome, and an expensive process. Second, all data is not created equal and thus has different replication needs. From the inception of digital storage, ensuring data is not lost due to user error, malicious acts, and hardware failure has always been, and still remains, an open problem.

In addition, the improvements in storage density combined with a proliferation of new digital media are dramatically changing what people store on disk. Gone are the days when the average home computer stored only letters and spreadsheets. A typical PC today stores financial records, tax returns, music and movie libraries, photo albums, and much more.

Computer storage is moving to the centre of people's lives. As it does, the consequences of a data-destroying failure become increasingly catastrophic. The digital-storage revolution thus requires not just that storage be cheap, but also that it be reliable.

Not all data is created equal. Data such as commercial music and movie libraries are, given time, easily reobtainable. On the other hand, data such as personal, health, and financial records, are much more difficult to reconstruct if lost. Naturally, resources should be focused on the data that is difficult to restore if lost.

Protecting data from loss is not only difficult, it is also expensive. Even though the cost of

resources such as network bandwidth and storage is decreasing, they are still not free. In addition, the cost of administration is constantly increasing. Thus, the level of protection afforded to data should correspond to the data's importance.

1.1 Motivation

Current reliability techniques typically fail to adequately protect data, are very expensive, or both. Reliable-storage administration, for example, is an order of magnitude more expensive than physical storage itself [58, 80]. A key reason for this high cost is that many existing techniques such as tape and optical-disk backup require too much human intervention to scale to modern disk capacities and reliability standards. RAID [57, 92] and related distributed-disk solutions [20, 29, 39] provide a significant degree of automation, but do not protect data from site failures and are often expensive and complex. High-end commercial file systems [5, 44] protect data from site failure by tightly coupling to an off-site mirror, using a specialized high bandwidth connection. Some research systems provide similar protection [4, 47, 86] without requiring a specialized connection, but they also tightly couple primary and secondary sites. This tight coupling is a major source of complexity that inhibits scalability, increases cost, and makes the system less resilient.

Recent research has examined the use of a collection of peer-to-peer nodes to replicate or to erasure encode data and thus protect it from failure [10, 13, 18, 67, 69]. This approach has the potential advantage of low cost, loose coupling, and low complexity. Most recent peer-to-peer systems are organized as a distributed hash table (DHT) [68, 83] that stores a file, or file block, on the node whose ID is closest to the file's. Multiple copies of a file are stored on the i nodes closest to the file's ID. The fact that node IDs are assigned randomly means that replica nodes are each stored on randomly chosen nodes with presumably independent failure modes. A limitation of this approach, however, is that when a new node is added, many of the files stored on its immediate successor must be copied to the new node, or at least redirection pointers must be used. The DHT based systems also assume that the cost of replicating to a node and its reliability is uniform across the system, and that its failure modes are independent from others. Although this may be true in an ideal world, these assumptions are generally over simplified [43, 89]. In addition, users never explicitly know on which nodes their data is stored, which is problematic for sensitive data.

To overcome the limitations of the DHT based peer-to-peer systems, several peer-to-peer sys-

tems [1, 20, 17, 73] take a different approach to replication. Farsite [1] designates a subset of nodes to co-operate to store a hierarchical index of the file system. A file's index entry lists the IDs of the nodes that store it, which are initially chosen at random from the entire system. As the namespace grows, it is automatically partitioned and assigned to non-overlapping sets of nodes. In addition, Farsite continuously moves metadata and file data from low availability to higher availability machines to improve data availability. Pangaea [73] stores files on nodes that frequently access them. A subset of these nodes are charged with ensuring that the file's *replication level*, the number of copies, is maintained. It uses information such as available disk space, location, and network connectivity to select the nodes responsible for maintaining a file's replication level. Starfish [20] replicates at the block level. It predominantly uses monitoring to ensure that replica nodes are alive and automatically re-replicates the data if a replica node fails. Unfortunately, the input of a system administrator is necessary to select replacement nodes. Pastiche [17] is similar to Farsite except that replica nodes store chunks rather than files and it uses the Pastry [68] substrate to locate suitable nodes. A node is deemed suitable if it already holds a significant percentage of the chunks; this technique was originally introduced by Muthitacharoen et al. [48] in the *Low Bandwidth File System*.

The techniques presented above strive to automate the process of backup and recovery, but they are insensitive to the costs of replication. As storage capacities increase, cost of physical media decreases, and users create an ever increasing amount of data, it becomes infeasible to replicate everything uniformly and ignore the cost of replication. PDR, however, provides a system that is resource conscious and focuses the replication resources where they are most needed and provide the most benefit.

1.2 Issues

The primary goal of any backup system is to protect data against loss due to user error, malicious acts, hardware failure, and environmental disasters. In the majority of cases these systems replicate the data, that is, make one or more copies of it, and store the copies on different drives, on separate hosts, and/or on remote sites. The number of copies and their location determines the *level of replication* of the data, and hence the level of protection.

Protection can be further decomposed into *availability* and *durability*. Availability describes how easy is it to access the replicated data in the event of a failure. For example, in a RAID system

if a disk fails the data on the mirrored drive is automatically and immediately available. In tape backup systems, however, users generally have to wait for the appropriate tape to be mounted and for the data to be restored by an administrator before they can access the data again. Durability describes the data's persistence or the robustness of the system. For example, the failure of a RAID controller could cause all the data on the RAID array to be corrupted and lost. Whereas optical media, once written and verified, lasts almost forever (ignoring media deterioration).

To ensure availability and durability there must be at least one copy of the data at all times, and the failure of a system component must not cause total data corruption. Ensuring high availability is even more difficult because a copy must always be readily available. If a copy fails, that copy must be re-created in a reasonable amount of time. A reasonable amount of time is a period in which the likelihood of all copies of the data disappearing is extremely small or nil. Increasing the availability and durability generally increases the complexity of the system. As the number of copies increases, the cost of keeping them consistent increases, which can hinder scalability. In addition, increasing availability further increases complexity since the system must react to failures faster to decrease the window of unavailability.

Maintaining the consistency of replica nodes is the primary responsibility of a replication system. There are many different approaches to maintaining consistency. To characterize the different approaches a five dimensional taxonomy, similar to the one created by Saito et al. [70, 71], is created and used; the five dimensions are *when*, *where*, *who*, *how*, and *what*. The *when* dimension characterizes how quickly updates are propagated. In *pessimistic* replication updates are propagated immediately to the replica nodes when data is modified. This approach usually tightly couples replica nodes and processes the updates in a transaction like-manner. *Optimistic* replication, on the other hand, propagates updates asynchronously in a relaxed and best effort manner. Optimistic replication does not require the tight coupling of replica nodes or the use of expensive algorithms to maintain strict consistency as in pessimistic replication, and thus is less complex. Unfortunately, optimistic replication is unable to provide the consistency guarantees of pessimistic replication because replica nodes are usually not consistent at the moment of data modification, but become consistent eventually as the system quiesces.

The second dimension used to differentiate replication approaches is *where* the replicated data is stored; including how the replica nodes are chosen. In many distributed hash table (DHT) based

peer-to-peer systems, replica nodes are selected based on node IDs that are randomly assigned, and thus the data is always located on a random subset of nodes. Although this is beneficial to provide replica node diversity, it brings forth several security concerns. Although blocks can be encrypted, it does not prevent the administrator of the node to copy the data and break the encryption; one has to implicitly trust all the nodes in the system. Another approach is to rely on a single replica node, a *primary* copy, to receive and distribute all updates. This approach reduces system complexity and alleviates some of the security concerns, but it introduces a single point of failure and can reduce the availability of the data if the primary copy becomes unavailable and no secondary copies exist. Thus, it is important to ensure that multiple copies remain and to actively re-replicate data as replica nodes fail. Partitioning the file system and assigning the responsibility of replicating a subtree to a subset of replica nodes removes the single point of failure but increases complexity. In addition, as subtrees grow it may be necessary to perform replica node reassignment which is additional administrative overhead.

The third and fourth dimensions characterize *who* and *how* updates are propagated. Replication systems consist of two parties, the *client* is the creator and modifier of the data and the *replica nodes* store copies of the client's data. Either the client, *active replication*, or the replica nodes, *passive replication*, are responsible for update propagation, and updates are either *pushed* or *pulled*. An update being pushed means that the update holder is responsible for distributing the update to the replica nodes. Whereas, pulling means that replica nodes need to contact the update holder and retrieve it. The simplest approach is for the client to push the updates to the replica nodes. Unfortunately, this approach introduces complexity for maintaining consistency when the client fails. The consistency problem is alleviated if the client pushes the update to a replica node, and then the replica node pushes the update to the other replica nodes, but the complexity of the replica nodes increases since they have to handle a larger number of failure modes. A middle ground is for the replica nodes to pull the update from a primary copy. This distributes the load of propagating updates but introduces the need to monitor for new updates.

The final dimension that is used to characterize the systems is the granularity of replication, or *what* is replicated. This is an important dimension because there is a direct trade-off between efficiency and complexity when choosing the granularity of replication. If the granularity is large, at the granularity of a volume, then maintaining consistency, locating data, and performing recovery

is relatively straightforward. The drawback to replicating at a coarse granularity is that all data in a volume is replicated, both useful and useless. Thus, significant amounts of resources are wasted and there are no controls to reduce this waste. Reducing the granularity enables one to be more selective as to what is replicated but at the cost of increased complexity. Not only does the complexity of the system increase (the mechanism) but also the complexity of the user interactions with the system; that is, the user may need to specify more parameters or remember more information with respect to what and where their data is stored.

Designing a replication system is an optimization problem involving maximizing efficiency, minimizing complexity, and minimizing cost. Maximizing efficiency involves reducing the overhead associated with replication and reducing the amount of communication needed for control and coordination within the system. Complexity includes both the system view and the user view. By minimizing system complexity the system inherently becomes easier to scale, to maintain, and to understand. From the user's point of view, if the system is difficult to use and reason about, then it is not going to be used. Finally, the cost is the overall operating cost of the system, including network bandwidth, storage, and administration. Efficiency influences costs because the amount of communication overhead is proportional to the network bandwidth. Complexity influences costs because a system that is easy to use and understand usually requires less administration.

Security is an issue in any system that stores or manipulates private user data. A large amount of work has been done in keeping private data private and thus security is not directly addressed in this thesis. The structure of PDR is such that many different security measures could be used depending on the needs of the user.

1.3 PDR overview

Policy Driven Replication (PDR) is a peer-to-peer replication system. The key novel features of PDR are that it uses user-specified, file-grain policies to direct replication and that it uses a non-random approach to selecting storage nodes.

The PDR system is a collection of independently operating nodes (physical machines). There are no centralized services and all nodes run the same software. A node is either a client, a replica, or both. Client nodes only push data to replica nodes and are not responsible for storing replicated data or ensuring that replication policies are followed. Replica nodes, as the name suggests, store

replicated data for client nodes, and ensure that replication policies are followed even when nodes fail, re-replicating data when policy violations occur.

Each node comprises two parts: the *replicator* and the *policy oracle*. The *replicator* replicates data, and the *policy oracle* oversees policy creation. These tasks are separated to help PDR manage state. The replicator manages the file system state and the metadata associated with each file. The policy oracle manages the inter-node connectivity; in particular, it tracks system topology created by the replication policies.

The *replicator* process runs on every node and is responsible for replicating data; it interprets, executes, and enforces replication policies by re-replicating data if a node fails. It is integrated with the local file system and receives upcalls whenever a file is created or modified. The replicator invokes the local policy oracle when node selection is needed and communicates with replicators on other nodes for replication or recovery purposes.

The *policy oracle* is responsible for the majority of the decision making in PDR. First, the policy oracle selects the replica nodes to satisfy both new replication policies and existing policies when a replica node has failed. Second, the policy oracle communicates with other policy oracles to maintain up-to-date connectivity (network topology) information and discover new replica nodes.

When and where a file is replicated is specified by a replication policy. Users specify replication policies using either a command-line tool or a graphical interface. PDR provides a plug-in interface¹ that enables administrators of the system to create a set of tools, appropriate to the user base, to create, set, and modify replication policies. The immediate replication policies specified by the users are termed as *highlevel*. The plug-in is responsible for translating highlevel replication policies into a form that the replicator understands; these are termed as *lowlevel* replication policies. The lowlevel policy is a list of pairs consisting of a node and a time specifying when to replicate to the node. The policy oracle stores these lowlevel policies and the replicator is guided by them.

To ease the mental workload of setting replication policies a mechanism is provided to create a set of default replication policies that are automatically assigned when a file is created. This set of default policies are assigned to a directory and are automatically inherited by subdirectories when they are created.

¹The plug-in is a module that is dynamically loaded by PDR. The plug-in and PDR provide a set of functions that each other uses to manage policies.

Replica node selection is performed in a systematic way so as to maintain a good topology. When a policy is created a set of virtual links are established between the replica nodes listed in the policy. These links represent the dependencies with respect to monitoring and failure handling. If a replica node has many links then its failure will affect a large number of other nodes. Thus, one goal of selecting replica nodes is to minimize the number of links any replica node has, thus minimizing the effect of its failure on other nodes. Another goal is to select replica nodes that provide the desired level of protection. This is accomplished through a set of common and domain specific node attributes. The domain specific attributes are created as part of the policy plug-in creation process. A node is assigned these attributes when it is introduced into the system.

1.4 Thesis

Today it is common to replicate data to a variety of replica node types: local workstations, local storage servers, tape backup libraries, and remote sites across a wide area network. Unfortunately, the size of the data continues to increase at a rate much greater than the storage capacity of these systems and the throughput of the network. In addition, as the number of storage alternatives increases, the complexity of managing them (remembering where and when data was replicated) also increases. Although people are storing more data, the amount of important, irreplaceable, data is growing significantly slower than the amount of easily replaceable, unimportant data². Thus, in the foreseeable future, uniformly replicating data is at best wasteful, because all data is not equally valued or equally sensitive, and at worst infeasible.

A system where replication operates at the granularity of files and is specified by users can be built with only a small increase in complexity and overhead. This system reduces the amount of resources wasted, in terms of network bandwidth and storage, compared to existing replication techniques.

This thesis makes several assumptions: first, network bandwidth is not free, storage is cheap (but not free), and its administration is expensive. Second, the network bandwidth is an order of magnitude or more smaller than that of physical storage. Three, the network bandwidth increases at

²Users are storing a lot more music and movies which are several orders of magnitude larger than a Word document or an Excel spreadsheet.

a rate that is several orders of magnitude less than the rate at which storage capacity increases and the rate at which people create new data [58, 80].

There is no single or uniform way to explicitly define the value of data. The value is not simply defined by the number of hours it took to create it, or the amount of money it took to acquire it. Other subjective factors, such as sentimental value, greatly influence the perceived importance of the data. The adage “*beauty is in the eye of the beholder*”, applies to digital data as well.

In addition, user’s perspectives on backup, replication, and redundancy vary in three ways. First, user’s views vary on the protection level (the safety) afforded by the different backup and replication options. Second, the value of each option (how much one is willing to pay) is different for each user. Third, not every user knows about all the available options. Thus, there does not exist an absolute scale for either the protection level or the value for the different backup and replication options that is uniformly applicable to all users. Given this diversity in perspectives about data value and replication level a generic plug-in interface is provided that enables one to insert modules that provide the required interpretations for data value and replication level.

Replicating at the granularity of files and having users specify the level of protection can greatly reduce the cost of replication because only important data is replicated, and thus network bandwidth and storage is not wasted on unimportant data. These savings are traded for a small increase in both system and user complexity that arises from having to keep track of more information and requiring users to specify the desired protection level.

1.5 Contributions

The research presented in this thesis makes three main contributions. First, a system and a framework were developed that enables a user to specify how their data is to be replicated. Second, a replica node selection algorithm was created that maintains good topologies. A good topology is defined as one where a node failure or a major system event only affects a small number of nodes. Third, a prototype of PDR was designed, implemented, and evaluated.

1.5.1 Replication policies

PDR is unique in that replication is driven by user specified policies. This functionality is provided by a framework that consists of a plug-in architecture and an extensible model to describe replica

nodes.

The plug-in architecture provides a simple way to create a variety of user interfaces for the management of replication policies. An administrator of a PDR domain can easily create an interface that is suitable for the domain's user base. Also, this plug-in architecture aids in exploring the space of possible interfaces and hence improve our understanding as to what is an appropriate interface to manage replication policies. With the plug-in architecture PDR is not tied to any specific user interface or highlevel definition of policies, making it more portable and applicable to a wider variety of users and systems.

A model to describe replica nodes was defined to enable the translation of highlevel replication policies into sets of replica nodes. This model is extensible and expressive, allowing administrators to add domain specific, or custom, node attributes that can describe any physical aspect of a replica node. The plug-in translates protection or node attributes specified by a user through the user interface into an intermediate representation called *node selection constraints* which is a set of node descriptions. Given a set of node selection constraints it is straightforward to create a set of replica nodes by matching the required node attributes.

With the model for describing replica nodes it is possible to fully describe where data is to be replicated. This method cannot specify when data is to be replicated. For example there is no way to specify that a file should be replicated if it is modified by a particular application, or that file *A* should be replicated if file *B* is modified. Currently it is only possible to specify when, in terms of time, data is replicated.

1.5.2 Replica node selection

A replica node selection algorithm was created that is sensitive to the topology of the system. The algorithm is used to select replica nodes for new policies and to replace failed nodes. The selection process is guided by the node selection constraints and the system topology.

Selecting nodes solely based on the node selection constraints may create undesirable topologies. A replication policy establishes links, or dependencies, between the participating nodes. A replica node participating in many policies is linked to many nodes, and thus may encounter a significant amount of overhead during major system events such as replica node failure or obtaining consensus amongst the replica nodes.

The replica node selection algorithm maintains a good topology using two strategies; the idea is to maintain the topology as a large collection of small, disconnected components. First, when creating new replication policy the algorithm strives to create a small, highly connected component that is not connected to any other component or reuse an existing component. Second, when selecting a replacement node the algorithm attempts to split components into smaller components or reuse a replica node which is already connected. The general goal is to introduce the smallest number of new inter-node connections.

The replica node selection algorithm does not require complete knowledge of the entire system topology to operate. The more of the topology the algorithm knows about the better its selections become. Furthermore the algorithm not only maintains a good topology, but strives to improve it by moving data to allow inter-node connections to be removed. The replica node selection algorithm is evaluated using simulation in Section 6.2. The evaluation demonstrates both the effectiveness of the algorithm and its scalability.

1.5.3 The PDR prototype

A prototype of the PDR system was designed and implemented to validate the ideas presented in this thesis. The system is mostly portable, it is integrated with the local file system, and requires no kernel level modifications. PDR is a userlevel process that uses the Coda [36, 77] redirector to hook into the local file system. The system is portable to any operating system that supports Coda and requires only minor modifications to operate with a different redirector. PDR contributes some amount of overhead to file system calls. For applications such as document processing and software development this overhead is not noticeable. For file system intensive tasks that create a large number of files and directories there is a two to three times slowdown.

The prototype was evaluated in two ways. In Section 6.1 micro-benchmarks were used to determine the overhead induced by PDR on the file system. In Section 6.3 communication costs, number of messages sent and received, were used to evaluate the cost of replication and failure recovery. The analysis of the communication costs was also used to show the scalability of PDR.

1.6 What follows

Chapter 2 presents and discusses the research and the replication systems that have been designed and built over the last 30 years. The next three chapters present the bulk of the research in this dissertation. Chapter 3 introduces the Policy Driven Replication (PDR) system. It presents the architecture and discusses the design of the system. This chapter predominantly focuses on the mechanisms for replicating the data and ensuring that it stays replicated. The policy aspects of the system are presented in Chapter 4. This chapter discusses the plug-in mechanism and interface, how nodes are described and compared, and how policies are set, translated, and the effects they have on the overall system. Chapter 5 discusses the effects of replica node selection on the network topology in a system such as PDR, and how replica nodes should be selected to ensure that the topology remains good. PDR is evaluated in Chapter 6. Micro-benchmarks, simulation, and communication costs are used to evaluate the performance and scalability of the system. Finally, Chapter 7 concludes and presents future work.

Chapter 2

Background

Plurality should not be assumed without necessity.

— William of Ockham, 1330

A *storage system* stores data for users and usually uses replication to increase the data's availability and durability. Over the last three decades a large number of these systems have been proposed and built. Their designs vary widely along with the levels of availability and durability they provide; some are general purpose while others are application specific.

A storage system can be categorized by its *type*, how it interfaces with the operating system and the user, and how it replicates data. The *type* is predominantly determined by the unit of data, or granularity, the system works with and how the system reacts to, or handles, data modifications. To further aid comparison, and to position PDR within the realm of existing research, a taxonomy of replication is created. This taxonomy is then used to categorize storage systems based on various attributes of replication.

In the rest of this chapter, Section 2.1 presents the terms used throughout this chapter and the rest of the thesis to discuss replication. In Section 2.2 the four main types of storage systems are presented. Section 2.3 describes the different interfaces these systems use to interface with the user and the operating system. Section 2.4 presents the existing storage systems and PDR with respect to the taxonomy of replication.

2.1 Nomenclature

The terms used throughout the rest of this chapter and thesis are defined here. Although there do exist storage systems that do not replicate data (RAID-0, only striping), it is assumed that the term *storage system* refers to a system that does use replication. The term *host* or *node* refers to a machine or a device that can either create or modify data, store it, or both. The term *client* denotes the entity that resides on a host which creates or modifies data that is to be replicated. The term *replica* denotes the entity that stores a copy of the data for a client. A replica node can either be a host, a disc, or another storage device; unless explicitly specified a replica node is a host. It is possible that both a client and a replica node reside on the same host.

An *update* is data that was created or modified by a client and that has to be propagated to one or more replica nodes. The term *coupling* is used to denote a dependence amongst two or more nodes; the coupling ranges from *tight* to *loose*. Tightly coupled nodes are highly dependent on each other, a failure of a node is a major event that is handled immediately. Loosely coupled nodes are mostly independent and a failure of a node is handled at the system's leisure.

Throughout this thesis the *complexity* of a system is discussed. Complexity refers to several different aspects of the system. The first is the physical requirements of the system, which are elements such as the network connection, storage, special hardware, and administrative input. The second is the complexity of the software, and is measured in terms of the amount of state that is maintained, communication overhead with respect to the number of messages sent, and the algorithmic complexity. The third is the complexity of the storage system interface to the operating system. This is measured as the amount of work needed to interface the storage system with the operating system. The work entails modifying the kernel, writing a device driver, or modifying system libraries or applications. The fourth is the complexity of the user interface, and is measured by the learning curve of the system.

2.2 Storage system types

This section presents the four main types of storage systems. Most research systems are of type *block*, *file* system, or *object* store. A database is also a type of storage system. They are similar in nature to object stores and thus are not explicitly discussed here. Most commercial systems are of

the purely *backup* variety.

2.2.1 Block

Block type (block level) storage systems are equivalent to large virtual discs. They store data in equal and constant size chunks that are accessed by specifying a block number. These systems usually provide a simple interface consisting of a `GetBlock()` and a `PutBlock()` method. No additional metadata is maintained and the data is untyped. These systems usually present themselves as a large block device to the operating system, onto which any file system can be placed.

2.2.2 File system

The file system is the most popular type of storage system. It usually allows both the user and the operating system to access data through the standard set of system calls such as `open`, `close`, `read`, `write`, etc.

File-system-type storage systems usually provide availability and durability through replication and provide the functionality and behaviour of a local file system. They store both metadata and file data and must maintain the consistency between the two. In addition, they must handle concepts such as permissions, complex file system operations such as *lookup*, and concurrent data access.

2.2.3 Object store

Object stores take a different approach by storing key–value pairs similar to a database. When data is written to the store a unique key is generated and assigned to the data. To retrieve the data one has to provide the key, like in a database.

Data in an object store is unstructured and there is no set size for an object (data). The main difference between block and an object store is in the way data is accessed. In block storage the block numbers are assigned sequentially and are independent of the data stored in the block. Accessing block n accesses a specific location in the block store and that location always remains the same. In an object store the key is dependent on the data. For example, accessing an object with key k and then modifying it produces a new object with key k' . If the key computation is dependent solely on the data then the same key is returned for multiple store requests of the same data. If the key

computation includes a time stamp then a different key is returned for multiple store requests of the same data.

The interface to an object store is similar to that of block storage. There are two methods `PutObject()` and `GetObject()`. The main difference is that `PutObject()` also returns the key for the data.

2.2.4 Backup

The last type of storage system, termed *Backup*, is usually a userlevel application that is only responsible for replicating modified data. There is no integration with the operating system and the system simply scours the file system looking for changed data since the previous scan; once all the changed data is found it is replicated.

This type of system is becoming less and less feasible as the size of file systems continue to grow. Scouring the file system is an expensive and lengthy operation that cannot be performed often.

2.2.5 Summary of storage system types

Table 2.1 presents the relevant research storage systems categorized by type. The majority are of the file system type because the concept of the file system has been around for a long time. In addition, the file system is the appropriate place to implement additional functionality such as *versioning* [75], *snapshots* or *checkpointing* [30, 63], and *journaling* [28, 34], which provides additional protection and faster recovery times from a failure. File system type storage systems also tend to be more complex than their block and object store counterparts because they also have to provide all the functionality and semantics of a local file system.

The backup systems listed in Table 2.1 are all userlevel applications except for HSM [79] (Hierarchical Storage Management). HSM is a policy-based approach to managing file backup and archiving. An HSM usually consists of a hierarchy of different storage media, such as RAID, optical storage, and tape. Each level of the hierarchy represents a different level of cost and availability (speed of retrieval). For example, as a file ages in an archive, it is automatically moved to a slower but less expensive storage media. The administrator of the HSM creates the policies that describe what data is archived at what level of the hierarchy.

Block	File system	Object store	Backup
RAID [57, 92] Petal [39] CFS [18, 83] Starfish [20]	AFS [32, 47] Archipelago [33] Cedar [21] Coda [36, 77] Farsite [1, 11] Ficus [56, 66] Fragiapani [86] Intermezzo [62] Ivy [49] NFS V4 [46, 59] Pangaea [73] Pastiche [17] Plan 9 [63] Porcupine [72] xFS [4, 88] Zebra [29]	Bayou [61, 84, 85] OceanStore [37, 67] Past [69, 68]	Computer Associate's ArcServe [6] HP Omniback [53] HSM [79] Legato Networker [52] Netbackup [51]

Table 2.1: Storage systems categorized based on their type.

2.3 Interfaces of storage systems

There are five main ways that storage systems interface with the operating system, three are at userlevel and two are in the kernel. The userlevel interfaces are termed *application*, *stub*, and *interposition*. The kernel interfaces are termed *device* and *VFS (Virtual File System)*.

2.3.1 Application

The *application* interface is probably the simplest, but also the least efficient, of all the interfaces. The client is a userlevel application that is responsible for determining what data was modified and replicating the data. It is the least intrusive of all the interfaces requiring no modification to the operating system. Unfortunately, this benefit is obtained at the expense of efficiency. The only way to determine what data was modified is to scour the entire file system. *Backup* type storage system are the only ones to employ this interface.

2.3.2 Stub

The *stub* interface is used in applications that have very specific storage needs. The interface to the storage system is a proprietary library or a stub that is linked into the application at compile time. A benefit of this interface is that it requires no modifications to the operating system or the environment.

A drawback of this interface is that to use this system an application has to be modified and recompiled. In addition, if the interface library or stub changes then the application may have to be recompiled for the changes to take effect; a recompilation is necessary if the API changes or if the library is linked in statically. This interface is usually used by applications that have specific storage needs and they usually interface to object store type systems [37, 84].

2.3.3 Inter-position

The *inter-position* interface requires no modifications to the operating system or the recompilation of applications to connect and use any block, file system, or object store type storage system. Inter-positioning works by overriding the standard file system functions with versions that call into the storage system. For inter-positioning to work the operating system must support dynamic libraries and dynamic linking. A dynamic library is created with the overloaded functions and the library loading order is adjusted such that any previously defined file system functions are overridden by those of the storage system. Thus, when an application performs a file system call it is automatically redirected to the storage system.

Mostly file system type storage systems employ this interface, but usually only during the development and testing stages of the system. The reason this approach is not used in production environments is that it is difficult to control access. A system that interfaces in the kernel can use the access control mechanism provided by the operating system; this is not possible with inter-positioning. Also, applications that are statically linked cannot use this interface and some operating systems still do not support dynamic libraries and dynamic linking. Finally, there is a small performance penalty for using this approach.

2.3.4 Device

The first kernel interface is the *device*, in particular a block device, and which is especially appropriate for all block type storage systems (see Table 2.1). The storage system attaches itself to the operating system through a device driver that is compiled into the kernel or is dynamically loaded as a kernel module. Most modern operating systems support kernel modules, and thus this interface has become less intrusive since it is no longer necessary to recompile the operating system.

The operating system still needs to trust the device driver since it is operating within the kernel, with no protection, and can intentionally or unintentionally modify kernel data structures. Writing a device driver can be a tedious and onerous chore because extra care must be taken not to introduce bugs that make the operating system unstable and crash.

A significant benefit of this interface is that it allows any file system to be installed on top of the block device. From the point of view of the user this is a good approach because one does not have to adapt to a new file system with new semantics and new tools.

2.3.5 File systems

The second kernel interface is the *VFS*. This interface requires the modification of the operating system and users may have to adapt to a new file system. In addition, the developer of the storage system not only needs to have an understanding of the operating system kernel, but they also need to possess an intimate understanding of file systems to properly handle issues such as file data and metadata consistency and concurrent data access.

Most modern operating systems have abstracted away the file system layer into what is called the *virtual file system VFS*. *VFS* is an object oriented approach to implementing file systems. There is a required set of functions that are implemented and inserted into the kernel as an object. When a file system is accessed the kernel retrieves the file system's object and uses the provided functions to perform the request operation. This permits a file system to be written as a kernel module without needing to modify the operating system most of the time. Unfortunately, the kernel module is not portable and must be written for each supported platform.

A slightly different approach is to implement a thin veneer layer in the kernel that pushes file system calls to a userlevel server; AFS [32, 47], Coda [36, 77], Intermezzo [62], and PDR use this approach. The benefit of using the veneer layer is that the amount of code that is not portable

strict	←	consistency	⇒	eventual
volume	←	granularity	⇒	block
primary copy	←	replica node selection	⇒	random
passive	←	replication strategies	⇒	active
client	←	update responsibility	⇒	replica

Table 2.2: The replication taxonomy.

and specific to an operating system is reduced, the drawback is that regular file system calls take longer to execute because there are additional user–kernel boundary crossings and inter-process communication (IPC) overhead.

2.4 The taxonomy of replication

The replication taxonomy is based on five attributes that are commonly used to describe and compare storage systems. These attributes create a five dimensional space that provides a meaningful frame of reference for categorizing, comparing, and contrasting existing systems. The five attributes are the consistency model, the granularity, replica node selection, replication strategies, and update responsibility. This taxonomy is presented in Figure 2.2, each line is a dimension or characteristic with the end points of its range.

2.4.1 Consistency

The first dimension positions a storage system based on its *consistency model*, or the *level of consistency*, which ranges between *strict* and *eventual*. In the strict consistency model data is replicated immediately, or *eagerly*, upon its modification and systems that keep strict consistency are described as performing *pessimistic* replication. On the other end of the spectrum there is eventual consistency. The eventual consistency model does not require that updates be propagated immediately. Update propagation is either scheduled [17], performed during low system activity [56, 66], or when stale data is accessed otherwise [73]. Whereas strict consistency guarantees that replica nodes always hold current data, eventual consistency guarantees that the replica nodes are consistent when the system quiesces.

The level of consistency can be measured on the real line by using the *window of inconsistency*, which is defined as the maximum amount of time a copy is allowed to differ from the original. The

level of consistency is inversely proportional to the window of inconsistency and in most cases is directly proportional to the coupling of nodes; in general the tighter the coupling the higher the level of consistency.

Maintaining strict consistency imposes a non-trivial amount of overhead and forces most of the systems to tightly couple their nodes. The benefit of maintaining strict consistency is that it is easier to reason about the system, performance tends to be better (smaller access latency), and failure recovery is easier. The performance and failure recovery aspects are especially important for block level storage systems such as RAID [57, 92] (Redundant Array of Inexpensive Discs) and Petal [39]. Since these systems present a virtual disc to the system, the performance of the virtual disc should be similar to that of a regular disc. This is not an issue for RAID since it creates a large virtual disc from a number of local discs connected by a highspeed bus. Petal on the other hand uses a set of machines connected by a highspeed network to achieve the same goal. In both cases failure recovery is vital because the loss of a single block could render the entire file system useless. Recovery is simpler with a strict consistency model because recovery usually involves performing a simple copy of the data from an existing replica node to a new replica node.

Starfish [20], a recent block level storage system, showed that not all replica nodes need to be tightly coupled. Unlike Petal and RAID levels 5 and 6, Starfish does not stripe the data across a set of nodes or discs, but mirrors it (RAID level 1). They showed that to provide 99.999999% or more of reliability all that is necessary is for one local replica node and for one remote replica node to be tightly coupled to the client by a highspeed link, a third replica node can be connected by a high latency, low bandwidth link without degradation to reliability.

Another benefit to storage systems that employ strict consistency is that they do not require mechanisms to resolve conflicts during update propagation. Since updates are propagated immediately, synchronously, in a transaction like manner, it is not possible for updates to be interleaved or for two conflicting updates to be generated. A number of local area network (LAN) and wide area network (WAN) file system type storage systems [1, 29, 32, 49, 17, 86, 88] rely on this behavior. Block type storage systems interface with a single producer of updates, usually the operating system, and thus conflicts do not occur. Object stores either store immutable objects [37, 69] and thus updates create new objects and no conflicts are created, or the storage system [84] provides for mechanisms to resolve conflicts. File systems on the other hand, have people as the producer

of updates. Even though concurrent write sharing is rare [31, 54] it may still occur and thus must be handled. Most file systems already provide some sort of mechanism to resolve such conflicts, for example in *FFS* [45] the *last writer wins* rule is used¹. As long as the communication substrate guarantees that updates are delivered and applied in order and atomically, that is one-copy serializability [8] is achieved, then additional mechanisms to resolve conflicts are unnecessary.

As storage systems expand to the wide area, maintaining strict consistency becomes prohibitively expensive and it hinders scalability. A number of systems [36, 56, 85] reduce the need for strict consistency by introducing mechanisms to resolve conflicts. Coda [36, 77] enables clients to randomly connect and disconnect from the system. The client caches data, has the ability to modify it, and then at reconnection propagates the updates to the replica nodes. This type of behavior can create conflicting updates and to combat this problem Coda introduced conflict resolvers. Conflict resolvers are pieces of code that are either supplied by the system or are inserted by the user that are executed when a conflict is detected. The choice of resolver to execute depends on the type of the data. Object store systems are more likely to have resolvers because they store explicitly typed data.

Ficus [56, 66] loosely couples its replica nodes and new updates are propagated via a gossip style protocol [7, 19, 87]. Although the use of the gossip protocol considerably reduces the overhead introduced by other update propagation algorithms (see Section 2.4.4), it is much more likely that updates can be delivered out of order, or conflict if concurrent updates occur on two different replica nodes. For this reason, Ficus also has default conflict resolvers and allows users to register their own.

On the other end of the consistency spectrum are systems that employ an eventual consistency model which *lazily* propagate updates and perform what is termed *optimistic* replication [70, 74]. The biggest benefit of optimistic replication is that replica nodes can be loosely coupled. Since an update need only eventually arrive at a replica node, the unavailability or transient failure of a node is not a significant system event. Replicas need not be constantly monitored. Only if a replica node is unavailable for an extended period of time are recovery operations started. Another advantage of looser coupling is that it is easier to scale the system to a large number of nodes and into the wide area since the consensus algorithms that ensure strict consistency and require the tight coupling are not used. A drawback is that conflicts become much more common and there is an increasing danger of reading stale data [24]. In addition, recovery becomes more difficult because replica nodes are

¹Only the effects of the writer with the latest timestamp are seen.

potentially no longer identical, and thus recovery involves determining what has to be replicated and from where it should be replicated rather than just the simple duplication of a replica node.

Since updates are not eagerly propagated it is necessary to control the staleness of a replica node. Algorithms by Saito [72] (Porcupine) and Ladin [38] guarantee that if there are no updates for a period of time and each replica node is available at some point then the replica nodes' contents converge. *View consistency* [23] provides stronger consistency than eventual consistency by ensuring that a client never sees data older than the data already seen. Pu et al. [64] look at creating staleness metrics for controlling the staleness of data at a replica node. In their model environment, updates are guaranteed to be delivered, but they are not immediately committed. Pu et al. analyze the characteristics of updates, and how many uncommitted updates there could be before a request fails.

PDR uses an eventual consistency model but the issue of replica node staleness is less important because data is only accessed on the replica nodes if the client fails. Thus for PDR it is important to have up-to-date replica nodes for recovery purposes rather than for accessibility. Clients propagate the updates to the replica nodes, and the replica nodes are responsible to monitor the clients during update propagation. In the event of a client failure the replica nodes are responsible to bring each other up-to-date.

Pangaea [73] takes the following approach to maintaining consistency and ensuring replica nodes do not become stale. First, the replica nodes are partitioned into two sets, *gold* and *bronze*. The number of gold replica nodes is usually on the order of three or four and they continuously monitor each other, if a gold replica node fails it is immediately replaced. When a client modifies data, the update is propagated to one of the replica nodes. If it is a gold replica node then the update is propagated to the other gold replica nodes and the bronze replica nodes are informed of a new update that they are responsible to retrieve. If the update is propagated to a bronze replica node then the bronze replica node propagates the update to a gold replica node and the update propagation process continues from there. In this manner the gold replica nodes are always current and they hold the definitive answer as to the state of the data. The gold replica nodes are also responsible for maintaining the replication level. Depending on the number of gold and bronze replica nodes, if a bronze replica node fails it may or may not be recovered. If the system sees that the number of bronze replica nodes falls below a certain threshold then it chooses a new bronze replica node

and re-replicates from a nearby gold or bronze replica node. A subtree of the file system is the responsibility of a set of gold and bronze replica nodes and thus it is straightforward to determine what has to be re-replicated and where the data is located.

Monitoring is predominantly used in PDR. Replicas monitor each other and are exclusively responsible for performing recovery as in Pangaea. When a failure is detected all the remaining replica nodes are brought up to the current state, this work being performed by the most current replica node, and then a new replica node is selected and brought into the fold (see Section 3.7). In both PDR and Pangaea it is the responsibility of the new replica node to obtain the data from an existing nearby replica node.

2.4.2 Granularity

The *granularity* of replication is the second dimension and describes the unit of replication. The granularity can easily be discretized based on the type of the storage system, thus there are four granularities, block, object, file, and volume. The smallest granularity is the block, and is the unit predominantly used by block level type storage systems [20, 39, 57, 92]. CFS [18, 83] is a *distributed hash table DHT* based peer-to-peer storage system (see Section 2.4.3 for more detail) that replicates at the granularity of blocks. Interestingly though, CFS has all the properties and characteristics, and interfaces in the same manner, as other object store systems [67, 69, 85]. Ivy [49] uses CFS as the storage substrate and creates a file system that interfaces by the inter-positioning approach. Replicating at the granularity of blocks is a double-edged sword. On one hand there is no metadata to maintain and recovery is very systematic. Systems such as RAID [57, 92] and Petal [39] use block placement functions that are only dependent on the *BlockID*, and thus locating and re-replicating the blocks is straightforward. Clients can easily locate the lost data and the re-replication of a failed replica node is straightforward; RAID does it in hardware. Unfortunately, in most cases all the blocks have to be recovered for the data to be useful, since a missing block could render the file system unusable.

The majority of the file system type storage systems [1, 21, 29, 32, 33, 17, 56, 59, 62, 63, 73, 86, 88] replicate at the granularity of volumes. A volume is usually a subtree in a hierarchical file system, it can be as large as the entire file system or as small as a single directory. Replicating at the granularity of a volume means that the subtree is replicated wholly on the same set of replica nodes.

This approach greatly simplifies recovery since the client can go to any replica [32, 56] storing the volume to recover lost data, and the recovery of a replica node simply involves re-replicating the volume [1, 73]. Another benefit is that creating a *snapshot* or a *checkpoint* [21, 63] of a volume is simplified since the snapshot can be created from any replica node and only one replica need be involved if it stores the whole replica node. A drawback of replicating at the granularity of files is that creating a snapshot of a volume is significantly more complex. A *checkpoint* or *snapshot* is a read-only copy of a volume taken at a point in time. Restoration with snapshots is trivial since it simply involves copying the snapshot back and with a series of snapshots one can create a history of the volume. To create a snapshot of a volume one simply needs a single replica node, whereas with file granularity replication one may need to contact several replica nodes to build the complete snapshot of a volume. In general the larger the granularity the easier it is to perform recovery and to reason about consistency in terms of inter-file relationships; for example, a source tree.

Reducing the granularity of replication and replicating at the granularity of files brings a different set of benefits. First, it permits one to be selective as to what to replicate. Because the system can focus on individual files, given some user input, the storage system can focus its resources on providing availability and durability where it is necessary rather than spreading it thinly everywhere.

Replicating at the granularity of files also reduces or eliminates the scalability issues that arise in storage systems that replicate at the granularity of volumes. Many of the older systems such as AFS [32, 47], Archipelago [33], Coda [36, 77], Ficus [56, 66], and xFS [4, 88] require manual intervention by the system administrator to partition the file system into volumes and to re-partition the volumes as they grow in size. This approach does not scale as the system grows in both the number of nodes and volume size. Systems such as Farsite [1, 11], Pastiche [17], and Pangaea [73] have gone one step further and automatically re-partition the volumes as they grow and become too large; but this adds considerable system complexity.

Replicating at the granularity of files brings some complexity with respect to recovery, for both the client and the replica nodes. The client has to keep track of the replica nodes for each of its files rather than for a single volume, and recover lost data on a per file basis. This problem is easily solved by employing a database that maps replica nodes to files and files to replica nodes, and replicating the database on a set of well known and trusted nodes. Re-replicating a failed replica node is also not as simple as re-replicating a replica node storing a volume. A replica node stores a

collection of unrelated files for a set of unrelated clients. Thus, with high probability there does not exist a replica node that is a mirror of the failed node and thus a simple copy cannot be performed to recreate the replica node. Each client, with their associated replica nodes needs to coordinate to determine what data the failed replica node stored and re-replicate it from the available replica nodes.

Although storage is very inexpensive, it is not free and its administration cost is high. For this reason Pastiche [17] uses techniques based on LBFS [48] to reduce storage use. LBFS and Pastiche store distinct data chunks rather than blocks or files; chunks are variable size blocks. That is, if two files have similar data then they share the common data blocks rather than duplicating them and in this way the system saves on storage.

Object store systems replicate at the granularity of objects and achieve many of the benefits and drawbacks of systems that do file grained replication. Since an object, is stored based on its key, the client must have all the keys to the objects to obtain them from the object store. Again, the solution involves a level of indirection by maintaining an object that maps keys to objects and objects to keys and thus the client only needs to ensure that it does not lose the key to that object. In many object store systems [18, 37, 69] the location of the object is based on its key. In these systems each node is assigned a unique node ID and an object is placed on a replica node that has the closest matching node ID to its key. Past [69] for example replicates an object to the n nodes that most closely match the object's key. On a failure, the other $n - 1$ nodes select a node whose node ID is the closest to the key and re-replicate the data to that node. Again, the need for some cooperation and coordination between the replica nodes is the extra overhead incurred compared to replicating at the granularity of a volume, but Past allows the object creator to specify how large n should be for a particular object, which is a start of selective replication.

2.4.3 Replica node selection

The third dimension in the replication taxonomy describes replica node selection: what nodes store replicated data and how they are selected. On one end of the spectrum is *primary copy* [2]. In this scheme, used by systems such as NFS V4 [46, 59], Intermezzo [62], and AFS [32, 47], the client always replicates to and restores from the same replica node which is usually selected by the administrator. If there are multiple replica nodes then it is the responsibility of the primary copy

replica node to propagate the updates to them. Although this scheme is simple, the primary copy, unfortunately, becomes a single point of failure in the system. If there are multiple replica nodes then they are usually tightly coupled to the primary so in the event that the primary fails one of the secondary replica nodes can immediately take over the primary's responsibility.

In a number of systems [1, 20, 29, 39, 56, 73, 88] the clients select the replica nodes. In systems such as Ficus [56, 66] and Starfish [20] the set of potential replica nodes is predetermined, and changes to it require the intervention of the administrator. Although this approach reduces the complexity of discovering and selecting replica nodes, it also inhibits scalability and increases the cost of administration.

In Petal [39], Zebra [29], and xFS [4, 88] all the nodes are potential replica nodes but a single manager node directs the client as to where to place the replica node. This scheme is also limited in scalability and potentially has a single point of failure, the manager node. xFS does provide for multiple manager nodes, but each manager node is responsible for a volume in the file system and if the manager fails then that volume becomes inaccessible.

Farsite [1, 11] improves on these approaches by automatically creating the replica node sets. Farsite designates a subset of nodes to store a volume. These subsets are non-overlapping and are initially chosen at random. In the event of a node failure Farsite simply selects a random replacement replica node from the set of available nodes. Although all this management is automated, Farsite depends on perfect knowledge of the system, that is, Farsite knows about all the nodes in the system. This aspect makes Farsite limited to mostly LAN use or single domain use where such information is possible to maintain.

Pangaea [73] further improves the set selection and creation process. Pangaea, like Farsite, automatically creates replica node sets for subtrees of the file system. Instead of randomly, nodes are selected based on proximity, *TCP TTL* (Time To Live), information, available space, and amount of down time. All this information is disseminated using a gossip protocol. In addition, the gossiping of information permits the discovery of new replica nodes and thus there is no requirement, as in Farsite, that each node know about all the other nodes in the system.

The approach taken by Pangaea is very similar to that taken by PDR. PDR also employs other node attributes and system topology to perform more restricted node selection. This has the added benefit of lowering the cost for recovery since the failure of a replica node affects a smaller number

of nodes.

A large amount of recent research has examined the use of a distributed hash table (*DHT*) [68, 83, 93] to build peer-to-peer storage systems [18, 37, 49, 69] (object stores). Each node is assigned a quasi-random ID. Data, objects, files, and blocks are assigned keys, by a known hash function, from the same space as the node IDs and are stored on n nodes with the closest node IDs to the key. Thus, the replica node selection and data placement is more or less performed randomly as in Farsite. The main difference is that there is a relationship between the key assigned to data and the node IDs the data is stored on. Hence, one knows given the key the replica nodes on which the data resides, while in Farsite one needs to query the system to determine this. The benefit of placing data on random replica nodes is that diversity is attained probabilistically, but not guaranteed, in terms of location and reliability, as long as there is no relationship between the proximity of the nodes and the closeness of the node IDs. The drawback is that one has implicitly to trust all the nodes in the system and assume that all the nodes are identical, which is wrong [43, 89].

2.4.4 Replication strategies

The fourth dimension of the replication taxonomy describes how updates are propagated. Replication strategies range from *passive* to *active*, and can either be *eager* or *lazy*. In passive replication [60] the client propagates the update to a replica node and then the replica node propagates the update to the other replica nodes. Eager passive replication is known as *primary backup* [25]. The client–replica node relationship is equally described by primary copy and primary backup; primary backup further describes the replica node–replica node relationship. In active replication [60, 78] the client propagates the update to all the replica nodes. Eager strategies tightly couple the nodes and propagate and apply updates in a transaction like manner. These systems usually employ protocols such as *abcast* [27] (atomic broadcast) to keep state synchronized and consistent.

The choice of replication strategy is influenced to some extent by the required level of consistency. Systems that have a strict consistency model use either eager passive or eager active replication.

Storage systems such as NFS V4 [46, 59], Intermezzo [62], and AFS [32, 47] use the pure eager passive replication strategy. Both replicate to a single master replica node, the primary copy, and that replica node propagates the updates to one or more other replica nodes. This is a natural strategy

to use given the design, structure, and consistency model of these systems.

Active replication is predominantly used by storage systems such as Coda [36, 77], Farsite [1, 11], and Starfish [20]. Starfish uses a combination of both eager and lazy active replication. Usually one local and one off-site replica node are propagated to eagerly and the others lazily, thus minimizing the number of replica nodes that must be tightly coupled. In Coda there is no strict notion of eager or lazy update propagation since clients can operate in connected or disconnected mode. While the client is disconnected no updates are propagated, but on connection the client eagerly-propagates the updates to all the Coda servers (replica nodes).

A number of storage systems, Fragiapani [86], Petal [39], xFS [4, 88], and Zebra [29], use a version of eager active replication that is termed as *guided*. In these systems the client directly replicates to one or more replica nodes but the set of replica nodes is dependent on the data. These storage systems have a replica node manager that the client initially contacts to determine which replica nodes the data should go to. This approach is similar to eager passive replication except that the actual work of replication is off-loaded onto the client.

The Ficus [56, 66] and Pangaea [73] storage systems employ both eager and lazy replication strategies. Ficus uses a passive replication strategy that is both eager and lazy. In Ficus the client selects a replica node from a predetermined set of nodes and eagerly propagates the update to that node; this is known as *multi-master*. Then that replica node gossips to the other replica nodes about the existence of a new update. It is then up to the other replica nodes to contact the replica node with the new update to retrieve it. Pangaea is similar except that the receiving replica node eagerly propagates the update to the gold replica nodes and lazily, via gossip, to the bronze replica nodes.

PDR uses both lazy active and eager passive replication strategies. During normal operations it uses lazy active replication to propagate the updates to the replica nodes. The replication of a file is a time-based, per replica node, scheduled event that is specified by the file's replication policy. If there is a failure then the replica nodes eagerly propagate all their new updates to the other replica nodes. In this way during normal operations the systems does not have to maintain tight coupling. But in the event of a failure all replica nodes quickly become consistent so that recovery operations become simpler.

2.4.5 Update responsibility

The fifth and final dimension defines who is responsible for propagating the update. If the creator or the originator of the update is responsible for the propagation then the update is *pushed* to the receiver. Note, if a node is responsible to propagate an update, even if it did not physically create that update, then it is still considered the originator of the update. If the receiver of an update is responsible for initiating the propagation then the receiver is said to *pull* the update from the creator or the originator. The main benefit of pushing updates is that it is easier to maintain consistency since the update can be pushed immediately after it is created. Pushing though places a large responsibility and potentially a large amount of overhead onto a single node to replicate to a potentially large number of replica nodes. By having updates pulled it is possible to reduce this overhead by reducing the amount of state that a replica node must maintain and distributing the responsibility onto the other replica nodes. The availability of a new update can be disseminated either directly, or by a gossip protocol, and then it is the responsibility of the receivers of this information to ask for the update. The drawback is that it is hard to bound when the receivers will ask for the update and thus it is impossible to maintain strict consistency.

Storage systems such as NFS V4 [46, 59], Archipelago [33], Intermezzo [62], and AFS [32, 47] rely on the client to push the update to the primary replica node. Once the update hits the primary, its safety becomes the responsibility of the storage system. The primary replica node is responsible for pushing the update to the other replica nodes. Ficus [56, 66] also relies on the client to propagate the update to a replica node in a set of replica nodes. Then the other replica nodes in the set are responsible for pulling the update from that replica node. This approach distributes the load and responsibility, and as a result reduces the coupling among the nodes in the set.

Other storage systems, such as Coda [36, 77], Farsite [1, 11], Pangaea [73], Pastiche [17], and Starfish [20], the client pushes the update to all of the replica nodes. This has the benefit of removing the overhead and responsibility of update propagation from the replica nodes, but it adds overhead onto the client and potentially makes recovery more difficult, especially if the client fails in mid-stream. In the event of a failure the replica nodes take on the responsibility of propagating the updates to ensure consistency. This is also the approach taken in PDR.

An approach taken by some systems is to only rely on the client to push the update. This approach is popular with systems that usually have a few replica nodes and employ a manager

node to tell the client where to replicate; for example Fragiapani [86], Petal [39], xFS [4, 88], and Zebra [29]. The nodes are tightly coupled and are connected by a high bandwidth, low latency connection and update propagation is assumed to be an atomic operation. Thus, a failure of a client in the middle of an update propagation is assumed to be a failed transaction, and the entire update is lost.

2.5 Summary of replication systems

Table 2.3 summarizes the storage systems presented in Section 2.4. Each system, and PDR, is presented with respect to the five dimensions of the replication taxonomy. The first column presents the consistency model used by each system. Although consistency is on a continuous spectrum, it is quantized as *strict*, *schedule*, and *eventual*. Systems that are classified as strict immediately propagate updates upon creation or modification of data. Eventual classified systems lazily propagate updates based on some criteria that differs from system to system. Schedule consistency is lazy propagation of updates, but the criteria for propagation is a time based schedule.

The second column is the granularity of replication. This dimension is quantized into the four main granularities described in Section 2.4.2. The third column presents the replica node selection dimension of the replication taxonomy. This dimension is quantized into six categories; *primary*, *manager*, *DHT*, *set-manual*, *set-random*, and *set-smart*. Storage systems that use primary backup replication are classified as primary. Systems whose replication is directed by a manager node are classified as manager. Peer-to-peer systems that are based on distributed hash tables are classified as DHT. For systems that are classified as set-[manual,random,smart] the clients replicate to one or more replica nodes in a chosen set of replica nodes. How these replica nodes are chosen is described as manual, random, or smart. Manual selection implies the input of an administrator is necessary, random selection means the nodes are selected at random, and smart selection means that nodes are selected based on some criteria, such as node and network attributes.

The fourth column summarizes the replication strategy that is taken by the storage systems. The format of the description is a tuple consisting of two letters: **E** for eager or **L** for lazy and **A** for active replication, **P** for passive replication, or **M** for manager replication. Manager replication means that a single manager node is responsible for directing the replication process. If two different replication strategies are used between clients and replica nodes and between replica nodes then a second tuple

	Consistency	Granularity	Replica Selection	Replication Strategy	Update Responsibility
PDR	schedule	file	set-smart	LA-FEP	CP-FRP
AFS [32, 47]	strict	volume	primary	EP	CP-RP
Archipelago [33]	strict	volume	primary	EA	CP
Bayou [84, 85]	eventual	object	random	LA-LP	CP-RP
CFS [18, 83]	strict	block	DHT	EP	CP-RU
Cedar [21]	eventual	volume	primary	LA	CP
Coda [36, 77]	strict	volume	set-manual	EA	CP
Farsite [1, 11]	strict	volume	set-random	EA	CP
Ficus [56, 66]	eventual	volume	set-manual	EP-LP	CP-RU
Fragiapani [86]	strict	volume	manager	EM	CP
Intermezzo [62]	strict	volume	primary	EP	CP-RP
Ivy [49]	strict	volume	DHT	EA	CP
NFS V4 [46, 59]	strict	volume	primary	EP	CP-RP
OceanStore [37, 67]	strict	object	DHT	EA	CP
Pangaea [73]	eventual	volume	set-smart	EA-LP	CP-RU
Past [69, 68]	strict	object	DHT	EA	CP-FRP
Pastiche [17]	schedule	volume	set-random	LA	CP
Petal [39]	strict	block	manager	EA	CP
Plan 9 [63]	strict	volume	primary	EP	CP-RP
Porcupine [72]	eventual	volume	set-manual	LA	CP
Starfish [20]	strict	block	set-manual	EA	CP-FRP
xFS [4, 88]	strict	volume	manager	EM	CP
Zebra [29]	strict	volume	manager	EM	CP

Table 2.3: Summary of storage systems, presented within the replication taxonomy.

is used to describe the strategy between the replica nodes. If the second tuple is preceded by an **F** then the strategy is only used during failure mode recovery operations.

The fifth column summarizes who is responsible for propagating the updates in the system. The format of the description is similar to the one used in column four. The first letter in the tuple specifies whether it is the client, **C**, or the replica node, **R**. The second letter specifies whether updates are pushed, **P**, or pulled, **U**. If the method differs between client–replica node and replica node–replica node update propagation then there is a second tuple. If the difference is only during recovery operations then the second tuple is preceded by an **F**.

Chapter 3

Selective replication

Simplicity is prerequisite for reliability.

— Edsger W. Dijkstra, 1975

The design of PDR is decomposed into three components: selective replication, policy creation and instantiation, and replica node selection. The name *selective replication* is given to the mechanism that determines what data to replicate, when to replicate it, and where to replicate the data. In this chapter the design and operation of the selective replication mechanism is presented. Both normal and failure operations are discussed, along with the client-to-replica node and the replica node-to-replica node interactions. For the purpose of simplicity and clarity of this discussion the replication policies are in their lowlevel representation (see Section 3.2); how they are set, the user interface, and how they are translated into the lowlevel form is discussed in Chapter 4. In addition, the node selector is a black box for this discussion; how nodes are selected is described in detail in Chapter 5.

3.1 Design overview

PDR is a peer-to-peer system. Nodes are independent operating physical machines, there are no centralized services, and all nodes run the same software (see Figure 3.1). A node is either a client, a storage, or both. Clients create and push updates to replica (storage) nodes. Storage nodes store data for client nodes and ensure that replication policies are followed.

PDR is predominantly structured as an event driven system. Threads block waiting for events

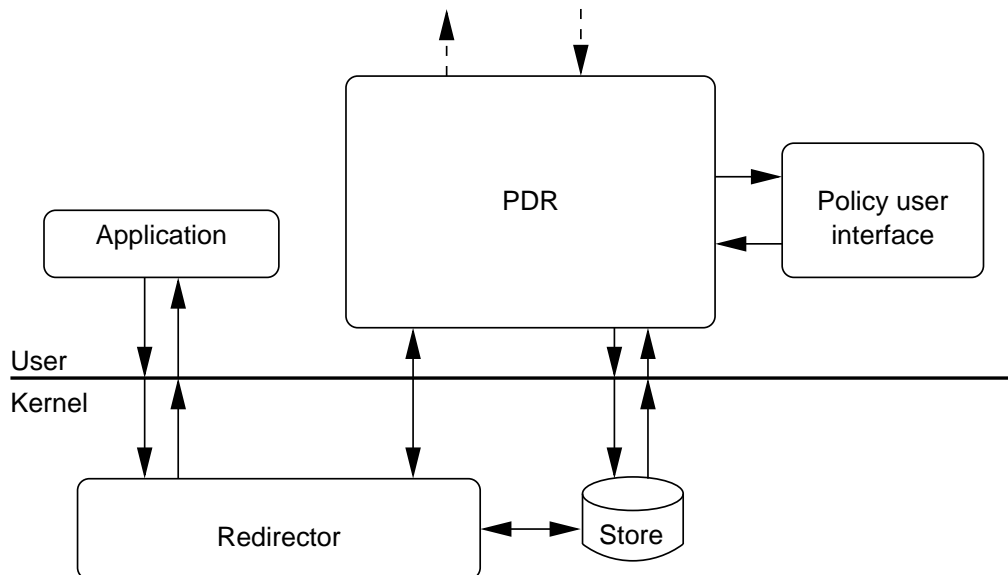


Figure 3.1: Global view and interaction of PDR with the operating system.

that arrive either via queues or sockets; inter-thread communication is primarily done using queues. Each thread is responsible for a particular stage in the replication process, making this architecture similar to SEDA [91]. This architecture provides for a lot of parallelism and asynchronous processing. In addition, a lot of complexity that is usually introduced by concurrency is removed since data structures are mostly not shared between stages or threads. The need for locks is reduced, thus making the system also easier to reason about.

All PDR nodes, client and storage, are implemented by the same userlevel process. Client nodes also use an in kernel file system *redirector* that hooks into the file system via the *VFS* layer and notifies the userlevel process of file system events (see Figure 3.1). The userlevel process is responsible for all operations in PDR that are related to replication, copying file data, recovering failed nodes, etc. PDR does not require any special functionality from the underlying operating system or file system. PDR stores file data and metadata as regular files in the local file system.

The userlevel process is composed of three main modules, the *replicator*, the *policy oracle*, and the *communication manager* (see Figure 3.2). The *replicator* is responsible for all operations related to the replication of data. The *policy oracle* is responsible for overseeing the creation and translation of replication policies. These tasks are separated to help PDR manage state. The *communication manager* provides support for all inter-node communication. The replicator manages the file sys-

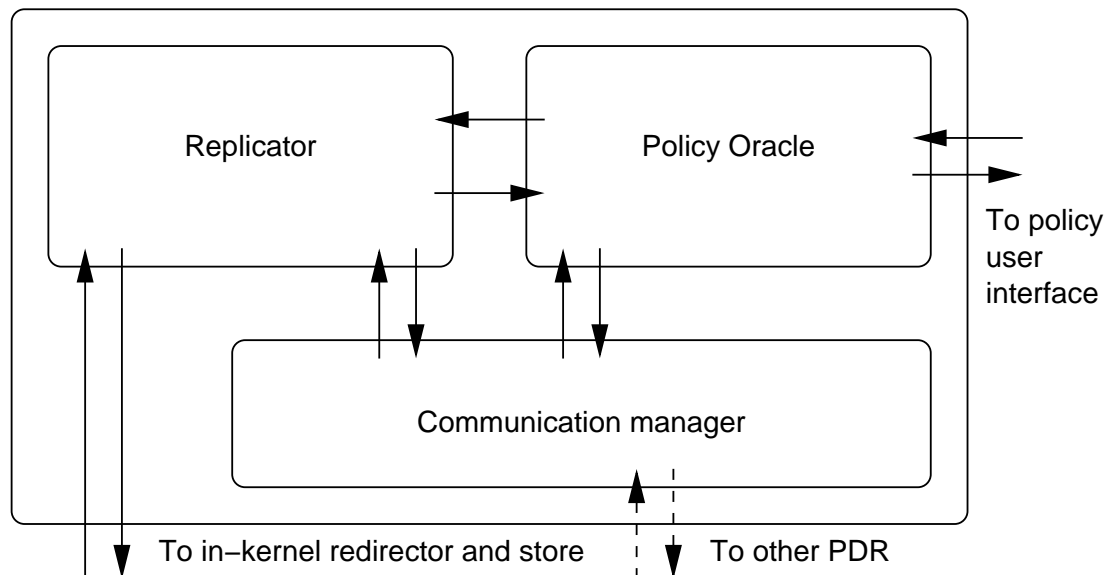


Figure 3.2: The components that make up the userlevel PDR process.

tem state and the metadata associated with each file. The policy oracle manages the inter-node connectivity; in particular, it tracks system topology created by the replication policies and actively discovers other nodes in the system 3.6.3.

The *replicator* process is responsible for replicating data; it interprets, executes, and enforces replication policies by re-replicating data when a node fails. It is connected with the local file system and is upcalled whenever a file is created or modified. The replicator invokes the local policy oracle for replica node selection, and communicates with replicators on other nodes for storage and recovery purposes.

The *policy oracle* performs the majority of the decision making in PDR. Primarily, the policy oracle is responsible for selecting replica nodes to satisfy both new replication policies and existing policies when a replica node has failed. It communicates with other policy oracles to maintain up-to-date connectivity (network topology) information and discover replica nodes.

The *communication manager* provides support for all inter-node communication. It provides for asynchronous message delivery, retransmission, and failure handling for undeliverable messages.

3.2 Replication policies

Replication policies control where and when data is replicated. In PDR there are two notions of replication policies, highlevel and lowlevel. Users specify highlevel policies and assign them to files. PDR provides a generic interface that enables users to specify highlevel replication policies in a wide variety of ways. A highlevel policy usually consists of two types of information. The first describes the desired attributes of the replica nodes, such as location, available storage space, and network connection. The second is a replication budget or the desired protection level. An example of a highlevel policy is: *replicate the data on local and offsite storage nodes, the staleness of the document should be no greater than t minutes, and the budget is c a month.* This information is transformed into an optimization problem that maximizes the protection level given a budget or minimizes the cost given the desired protection level. The policy oracle performs the translation and stores them as part of a file's metadata. The replicator uses these lowlevel policies to guide replication. The translation process is a multi-step process and is described in greater detail in Chapter 4.

The lowlevel representation of a policy, shown in Figure 3.3, consists of a number of fields and lists. The *replica node list* is a set of pairs $\langle \text{host}, SF \rangle$ consisting of a hostname and a *staleness factor* ((SF)). The staleness factor bounds the datedness of data for a replica node. A staleness factor of x seconds means that data should be replicated no later than x seconds after it is modified. The staleness factor can also represent a specific time, in a 24 hour period (e.g., 3:13am), when replication of modified data should occur.

In addition, each lowlevel replication policy stores a number of other pieces of information. First there is a version number that the policy oracle increments when a user or PDR modifies a policy; PDR modifies the policy when a replica node fails and is replaced. Second there is the *policy specifier* \mathbb{L} which is the intermediate representation of a highlevel policy. The policy specifier is used by the replica node selection algorithm to select the storage nodes and create the lowlevel replication policy. The policy specifier \mathbb{L} is a set of pairs $\langle \mathbb{K}, k \rangle$ that specify the *node selection constraint* \mathbb{K} and a non-negative integer k for the number of nodes required. A *node selection constraint* \mathbb{K} is a set of *node attributes*, such as location, CPU, and network connection, that describes the physical attributes of a replica node. Policy specifiers, node selection constraints, and node attributes are presented in greater detail in Chapter 4. The policy oracle and the replica node selection algorithm

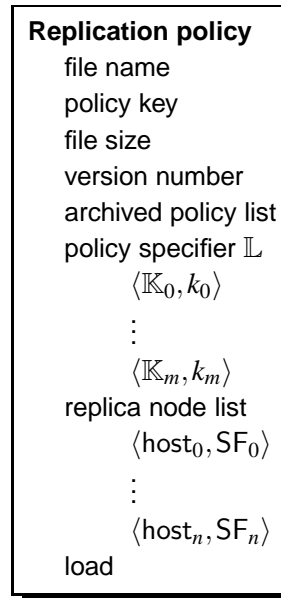


Figure 3.3: The replication policy as stored as part of the metadata.

use this information to select nodes.

In PDR replication policies are only applied to files; directories are not explicitly replicated. Since replication occurs at the granularity of files it is unclear what it means to replicate a directory. If replicating a directory involves replicating its contents then the benefit of file grained replication is lost. This approach has the benefit that directory metadata does not need to be replicated, which reduces system complexity.

Directories are implicitly replicated, from the root to the file's parent directory, when a file is replicated. A directory structure provides location information and context for file data; a directory that never had or will have file data is pointless and does not need to be replicated. If there is a need to preserve an empty directory structure it is possible to instantiate a special replication policy to replicate an empty directory.

In PDR default replication policies are created by assigning them to directories. A directory stores a list of policies that are keyed by file type, which is currently determined by the extension of a file; the extension is the string that follows the last dot in the filename. When a file is created it automatically inherits the appropriate default replication policy from the parent directory. Directories inherit the entire list of default policies from their parent directory.

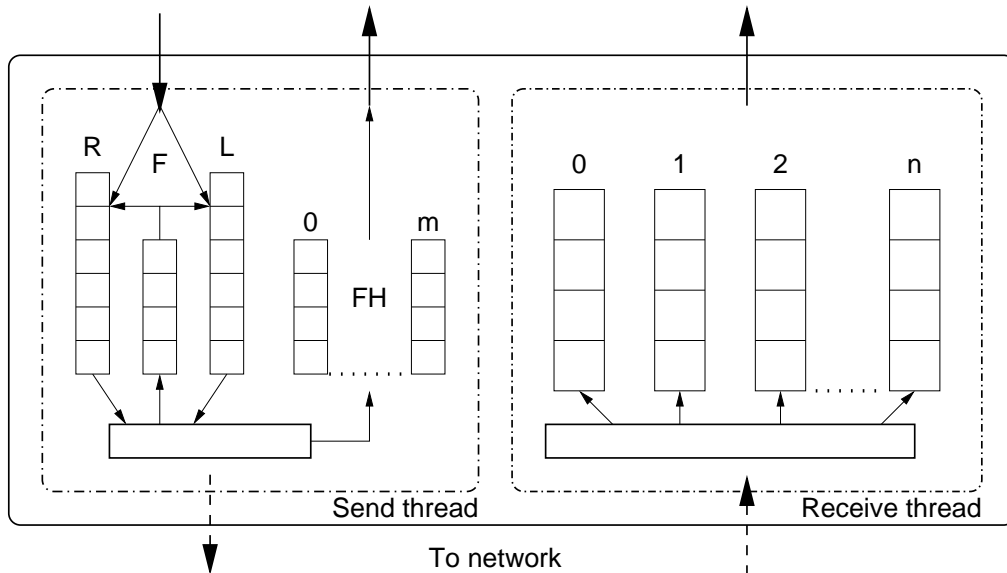


Figure 3.4: The communication manager manages all the communication in PDR.

3.3 Communication manager

The design of the *communication manager* focuses on providing a simple and efficient asynchronous message delivery system that handles both retransmission and undeliverable messages. The communication manager employs standard BSD sockets, two threads (one for sending and one for receiving), and a number of queues to provide the desired behavior (see Figure 3.4). The threads enable the communication manager to operate independently of the rest of the system and the queues enable the communication manager to provide asynchronous delivery and notification of failure.

3.3.1 Sending messages

The communication manager uses a thread and a number of queues to enable its clients to asynchronously send messages; a client is a thread within the process. Clients submit messages to be sent by enqueueing them onto a queue of regular priority or a queue of low priority. There is a queue of regular and low priority because the gossip messages for discovering new nodes and keeping the topology up-to-date are less important than replication and recovery messages and thus should not impede their transmission. The send thread blocks on these queues until there is a message to send.

When a message becomes available the send thread attempts to deliver the message. If suc-

successful the thread goes to deliver the next available message or blocks if the queues are empty. If message delivery is unsuccessful (a connection could not be established or the message was partially delivered), the message is put onto the failed queue. After a period of time, which is globally defined, the failed message is placed back on the appropriate message queue so that another attempt can be made at delivery. After a number of failed attempts the message is said to be undeliverable and is put on the client's undeliverable queue. Currently the number of attempts is specified globally but it is straightforward for the number of delivery attempts to be specified on a per message basis.

A client of the communication manager provides a queue into which undeliverable messages are placed. Usually a client uses a separate thread that waits on the queue and handles the undeliverable messages as they are returned by the communication manager. Messages are typed to enable the communication manager to place the undeliverable message into the right queue.

3.3.2 Receiving messages

To receive messages the communication manager uses a separate thread. On startup a client registers with the communication manager and specifies the incoming port and the message type of the messages it is expecting to receive. For each pair of port and message type a receive queue is created within the receive thread.

The receive thread listens on all the registered ports. When a message arrives it is placed on a queue that corresponds to the port number it arrived on and the message type. To retrieve the message the client queries the appropriate queue.

Currently only TCP type connections are supported and a connection is established and torn down for each message. Although, this approach is inefficient when it comes to sending many small messages, it is sufficient to show proof of concept for the PDR prototype and removes the necessity and complexity for implementing an efficient and reliable transmission protocol on top of UDP or other connectionless protocol. The communication manager is sufficiently extensible such that adding support for different types of connections and connection behaviors is straightforward. The client would need to register with the communication manager for both sending and receiving messages and specify the type of connection desired. Error messages would be propagated as messages via the undeliverable message queue and the receive queue for that particular client.

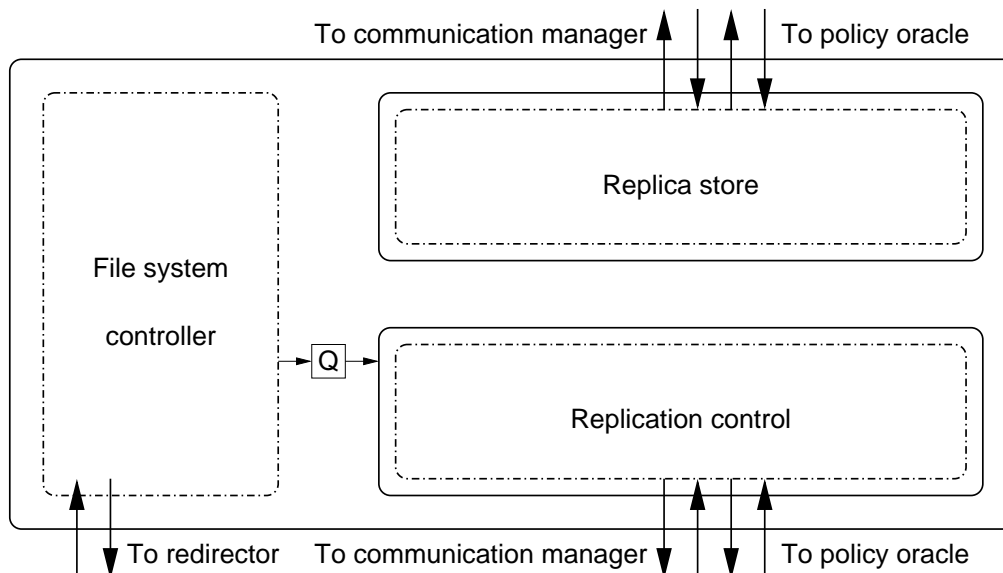


Figure 3.5: The structure of the replicator component in the PDR userlevel process.

3.4 Replicator

The *replicator* provides two distinct functions, one for client nodes and one for replica nodes. First, running on a client node, the replicator is responsible for processing the file modification events it receives from the kernel redirector, interpreting the lowlevel replication policies, and scheduling replication events. Second, running on a replica node, the replicator is responsible for processing storage requests from other PDR nodes, monitoring other nodes that replicate files it stores, and assisting or coordinating recovery of failed nodes.

The replicator (see Figure 3.5) is composed of the *file system controller*, the *replication control*, and the *replica store*. The file system controller communicates with the kernel redirector and processes the file creation and modification events. The file system controller informs the replication control when a file is created or modified and the replication control then executes the lowlevel policy for the file and schedules it for replication. The replica store receives requests from other PDR nodes to store file data. In PDR a node participating in a replication policy is responsible to monitor and to assist in recovery of all nodes listed in the policy.

3.4.1 File system controller

The *file system controller* (see Figure 3.5) is responsible for processing upcalls from the in kernel redirector. The redirector propagates all file system calls, except for `read` and `write`, to the file system controller. The interface between the redirector and the file system controller is operating system dependent. In Linux the file system controller communicates to the redirector through a pseudo device instantiated by the redirector. This device has similar properties to that of a unix pipe [82]. The communication is synchronous and is the standard send-receive-reply pattern.

The file system controller is thus in the critical path of all file system operations. To induce the smallest amount of overhead and latency it is necessary to reduce the amount of processing performed in the file system controller. To this end, the file system controller is run on a separate thread from the other parts of the replicator to decouple the file system operations from the replication operations.

When the file system controller receives a `close` for a modified file it creates a replication request, and inserts it into a queue for the replication control to pickup and process. In this manner replication is taken out from the critical path of regular file system operations and the induced overhead and latency is minimized.

3.4.2 Replication control

The replication control is responsible for interpreting lowlevel replication policies, scheduling replication based on the policies, and propagating updates. When the replication control receives a replication request it must retrieve the file's replication policy from the policy oracle and determine where and when the update should be sent. The replication control then creates a number of replication events that are inserted into a time based priority queue. The replication control employs the services of the communication manager when it is time for the update to be sent to a particular replica node.

The replication control is composed of three threads (see Figure 3.6), two for normal operations and one for handling failed updates. By placing failure handling on its own thread and decoupling replication and failure handling the process of replication is not retarded and thus data is not in danger of being lost due to possible delays in replication.

The replication process is partitioned into two threads, a *replication scheduler* and an *update*

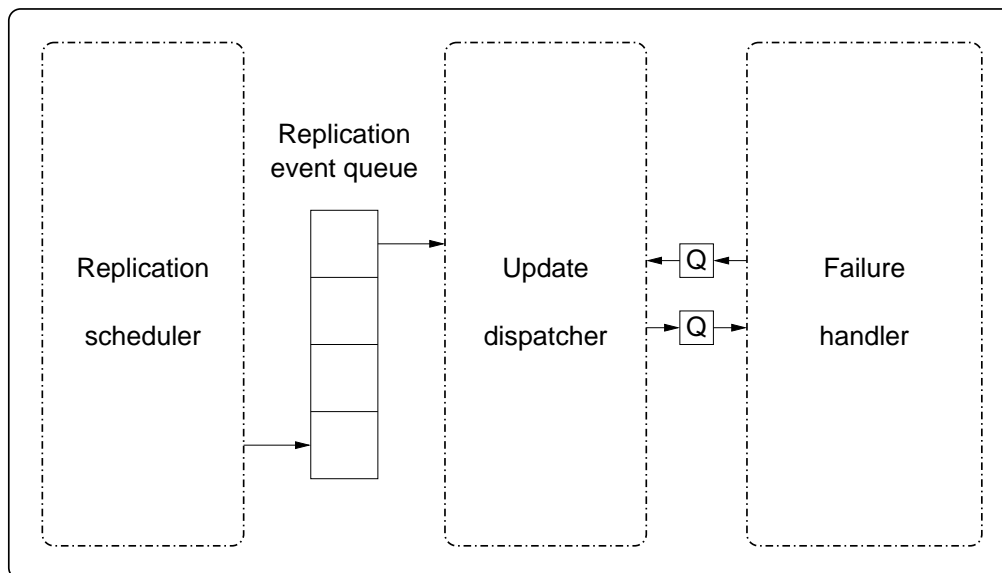


Figure 3.6: The structure of the replication control component in the replicator.

dispatcher. The tasks are partitioned because the operations of scheduling replication and dispatching updates are disjoint and can be overlapped to achieve parallelism. The *replication scheduler* waits for file modification events from the file system controller. On reception, the lowlevel policy for the file is retrieved from the policy oracle and the replication events are created and inserted into the *replication event queue*; if similar events already exist then new events are not created. Two events are considered similar if they only differ by their dispatch time. Similar events are not created because a replication event always replicates the current version of the file at the time of dispatch. Thus the second replication event becomes unnecessary since the current version is replicated within the required staleness factor. In addition, the replication events are written to a persistent queue so that if the node crashes temporarily then replication events are not lost and data is not in danger of being lost.

The replication event queue is a priority queue that sorts replication events based on the scheduled time of replication. The *update dispatcher* blocks until the next scheduled replication event. If the replication scheduler creates a replication event that needs to be dispatched before the next scheduled replication event then the replication scheduler unblocks the update dispatcher, and the update dispatcher re-evaluates the next scheduled replication event. The dispatching of an update involves replicating the latest version of the file specified in the event to the replica nodes listed in

the event. This mechanism enables PDR to avoid all unnecessary replication since not every version (modification) of a file is replicated.

The progress of the replication control is never impeded due to a node failure. If an update cannot be delivered by the update dispatcher the update is passed onto the *failure handler* thread in the replicator. Depending on the replication policy the update is reinserted into the replication event queue a number of times for additional delivery attempts. After a number of delivery attempts (the number depends on the policy) the replica node is deemed to have failed and a new replica node is selected. The update is rescheduled and the new updated replication policy is propagated to all affected replica nodes, see Section 3.5 for more details on policy update propagation.

3.4.3 Replica store

Apart from replicating data, the replicator is also responsible for storing data for other PDR nodes and ensuring that it stays replicated; these duties fall upon the *replica store*. The replica store performs three tasks. The first involves processing the store requests themselves and storing the file data. To ensure that replication policies are followed a replica node participating in the policy monitors the other nodes listed in the policy for liveness, this is the second task. The third task entails assisting in the recovery of a failed node. These three tasks are split amongst three threads (see Figure 3.7) in the replica store.

The *replica store* receives requests from other PDR nodes to store data on their behalf. The data is stored in the same form and with the same path name as on the client. A replica node creates a separate file system hierarchy for each client for which it stores data, with the root being demarcated by a unique client ID. This approach simplifies both the placement and the recovery of data.

Each replication policy has a *leader* node that has several additional responsibilities. The leader is implicitly chosen as having the smallest staleness factor because it always receives the first update from a client and thus is most likely to have the most up-to-date file data and metadata. Leader election protocols are unnecessary. By examining a replication policy a replica node can directly determine the leader of the policy. The leader is assigned the responsibility of monitoring the client until the complete replication policy is executed. In this way, if the client fails the leader can assume responsibility for ensuring that the replication completes to every node. The benefit of this is two fold, first the data's protection level is brought up to the required level, and thus the probability of

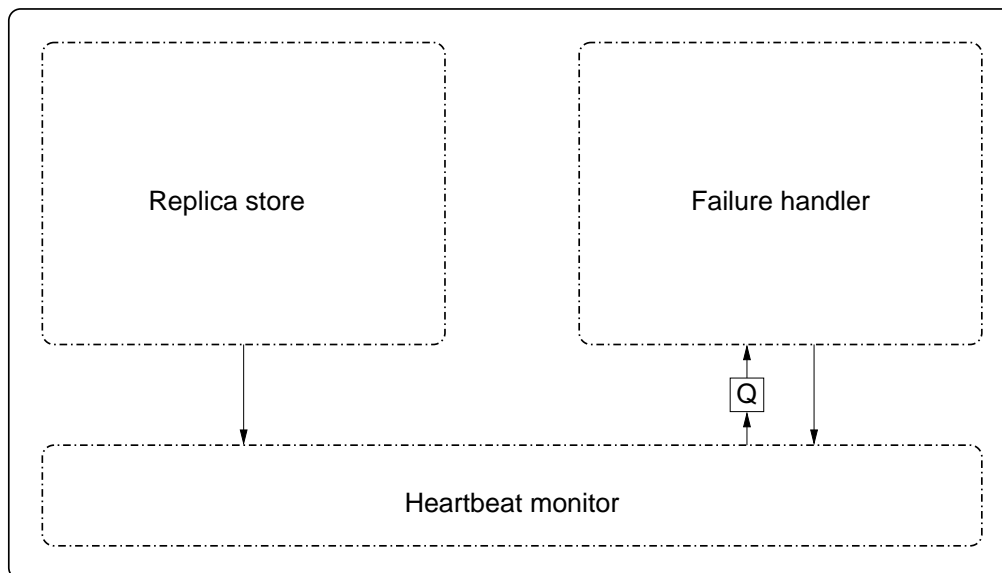


Figure 3.7: The structure of the replica store component in PDR.

losing data is reduced. Second, all the replica nodes become consistent and if necessary the client can restore from any replica. The other responsibility of the leader is to coordinate failure recovery in the event a replica node fails.

Clients are never leaders for their own replication policies because it is not known if they are highly mobile and if they have a high bandwidth connection (e.g. a laptop). A leader node should not be mobile and have a permanent connection so that it can ensure the replication policies are followed and to coordinate failure recovery.

The heartbeat monitor tracks the liveness of replica nodes that are linked to the node through the replication policies. For a node to be monitored it has to be registered. The registration is a three-tuple consisting of the host, the desired transition, and the frequency of the heartbeat. The transition is either from up-to-down or from down-to-up. The rate of the heartbeat messages depends on the staleness factor for that node, which helps to reduce the number of heartbeat messages. In addition, heartbeat messages are not duplicated for nodes that are participating in multiple policies.

The *failure handler* in the replica store either coordinates, if it is a leader node, or assists in the recovery of a failed node. Depending on the monitoring frequency of the failed node, the failure handler is notified of the failure by the heartbeat monitor or by another replica node. The reasons for decoupling the failure handling from the replica store are the same for decoupling replication

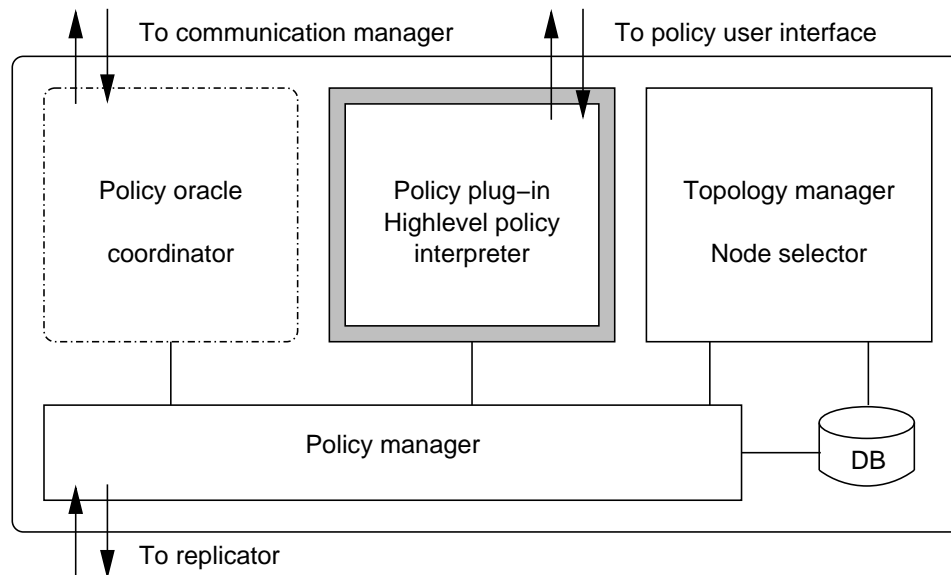


Figure 3.8: The structure of the policy oracle component in PDR.

and failure handling. These two tasks are relatively disjoint and the recovery of a node should not hinder the ability of other nodes to continue to replicate to it.

3.5 Policy oracle

The *policy oracle* has three primary tasks. First it selects replica nodes for new replication policies, second it selects replica nodes for the replacement of failed ones, and third it transforms highlevel replication policies to their lowlevel representation. The policy oracle also has a number of secondary tasks. It stores policy information for the node and provides an interface for the replicator to query this information. It also communicates with policy oracles on other nodes to gather and disseminate topology information.

Both clients and replica nodes have policy oracles. In the client, the policy oracle selects the nodes for new replication policies, translates highlevel replication policies to their lowlevel representations, and stores policy information that maps files to replica nodes. In the replica node the policy oracle selects replacement nodes for the replacement of failed ones and communicates with other policy oracles to maintain and track the topology. Figure 3.8 presents the structure of the policy oracle.

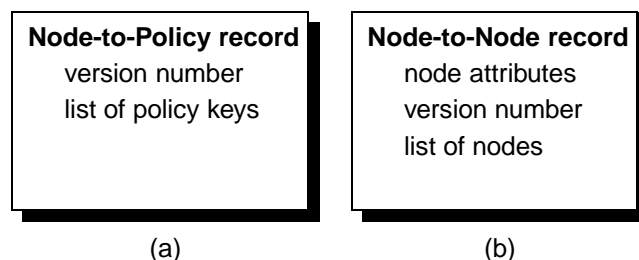


Figure 3.9: The policy oracle data structures, the node-to-policy and the node-to-node maps.

This section primarily discusses the *policy manager* and the *policy oracle coordinator* which are responsible for the secondary tasks. The primary tasks of policy translation and node selection are discussed in Chapter 4 and Chapter 5 respectively.

The policy oracle functions as the database for replication policy information. Each policy oracle has its own, local, database implemented by the Berkeley DB [81]; this database is configured to be thread safe, transactional, and recoverable¹ in order to provide consistency and fault tolerance. The database stores three types of data. First it stores the replication policies (see Figure 3.3). Each policy is keyed by a *policy key* which is a MD5 hash of the full path of the file, including the client ID for the root; there is an entry for every file. Second, the policy oracle stores two maps. The first (see Figure 3.9a) maps nodes to policies. For each node, it stores a record that contains a list of policy keys for the policies the node is responsible for. This map is predominantly used during failure recovery to quickly determine the policies a failed node was responsible for.

The second is a node to node map (see Figure 3.9b), the links between nodes. A link exists between two nodes if they are both named in a policy. Nodes are thus adjacent if they replicate the same file. This topology, or connectivity, is best represented as a graph where the vertices are nodes and edges are the links. Thus, the second map creates an adjacency list for this graph and is predominantly used by the topology manager to select nodes for policies.

The *policy manager* provides concurrent access to the information the policy oracle stores and is the interface to the node selection module. One of the primary clients of the policy oracle is the replicator and the policy manager is the interface. The policy manager is also the interface for the *policy oracle coordinator*. Both the replicator and the policy oracle coordinator employ the policy manager to create, update, and manage replication policy information.

¹The Berkeley DB uses journaling to aid recovery in the event of a crash.

The policy oracle coordinator is responsible for sending and receiving requests to and from policy oracles on other PDR nodes. The three types of common requests are for participation in a replication policy, for node information, and for topology information. Node and topology information is propagated in two ways. Nodes explicitly query for this information if it is required immediately either for policy creation or for node recovery. The information is also propagated via a gossip protocol [19, 87] in order to maintain up-to-date topology information on the PDR node. Maintaining current topology information improves the node's ability to perform node selection; as a secondary benefit less topology information needs to be explicitly retrieved.

3.6 Normal operation

The rest of this chapter focuses on the operations of PDR. There are three types of operations, those performed by a client node, those performed by a replica node, and those that are performed by the policy oracle. Sections 3.6.1 and 3.6.2 describe the operations of client and replica nodes and the interactions between them. Section 3.6.3 focuses on the operations of the policy oracle. Finally, Section 3.7 discusses the failure modes of all three types of operations.

3.6.1 Client operations

The client performs four main operations. It sets replication policies, services file system upcalls from the redirector, schedules replication events, and dispatches the updates.

Instantiating policies

Files acquire replication policies in two ways. One, a set of default replication policies are assigned to a directory. A file created in a directory inherits one of the default policies; files can only have a single policy. A subdirectory inherits the entire set of default policies from the parent directory. Two, users explicitly set and change the replication policies for files and directories.

Files inherit the replication policy from their parent directory based on the best matching type. Currently types are based on file extensions since traditional Unix file systems do not explicitly have the notion of file types. When a file is created the system examines the set of default replication policies available from the parent directory and selects the one that matches the extension of the file.

For example, if the extension of the file is “.txt” and there are policies for “.txt” and “.*” then the policy for “.txt” is selected. If there is no exact match then an empty policy is assigned to the file.

When a policy on a directory is modified the changes are not automatically propagated to the contents of the directory; only newly created files or directories inherit this new policy. To change all policies in a directory, the user must explicitly specify that the new policy should be recursively applied to all the existing policies in the directory. The semantics are identical to that of the *sticky* bit for directories.

Users also have access to a set of tools to enable them to set and change replication policies for files and directories (see Chapter 4).

File system upcalls

The file system controller receives all file system calls, except `read` and `write` which are serviced in the kernel, propagated by the Coda redirector. A replication policy is instantiated for a file or a directory when the replicator receives a `creat` or a `mkdir` file system call. The initial replication policy is either the default policy from the parent directory or empty. The file system controller marks a file as modified if it is newly created or opened for writing. On the reception of a `close`, the file system controller creates a replication request and passes it to the replication control if the modified flag is set.

Ideally, only the `close`, `remove`, `rmdir`, and `rename` file system calls should be intercepted by the replicator and only if the file has been modified, created, or deleted and a directory removed. This would not only reduce the file system call overhead and latency but it would significantly reduce the complexity of the replicator. One operation that would need to be modified is how default replication policies are set for directories. Currently, they are set on the reception of an `mkdir` call. One approach is to lazily set the default policies when a file is created for the first time in a directory. On file creation, the directories that make up the file’s path would be examined and default policies would be set if none were set before. This is a simple and elegant approach that would work well because directories are not replicated in PDR.

On the reception of a `rmdir` the replicator removes the associated entry for the directory from the policy oracle’s database. On the reception of a `remove` the entry for the file is marked as deleted

in the policy oracle's database. The rename does not just rename the file or directory. If a file or a directory is set to use its parent directory's default policy then on a rename the file or the directory acquires the default policies from the parent directory of their new location. If they had explicit policies set then those policies are maintained. The reasoning for this is that if the file or directory has the default policy set then the user has not focused on the file or directory and thus probably has the expectation that it is handled by the default policies. The converse is also assumed true that if the user focused on the file or directory then it must be special and the desired level of protection should be maintained as set by the user.

Scheduling replication

When the file system controller receives a `creat`² or an `open` call with the intent to modify the file, the file is flagged and a temporary copy of the file is created, so that previously scheduled replication events can proceed instead of waiting for the file to be closed. This decoupling means that no file system requests are ever blocked due to a pending replication or vice versa. The file is moved if the file is truncated on open, and copied otherwise. On receiving a `close` call the file system controller creates a replication request and passes it to the replication control.

Upon receiving the replication request the replication control (see Figure 3.5) retrieves the replication policy for the file. The replication control then examines the replication priority queue to determine if existing replication events for the file exist. A replication event is defined by a three-tuple $\langle filename, \langle host_i, SF_i \rangle, dispatch\ time \rangle$ consisting of the file, an entry from the replica node list (see Figure 3.3), and a dispatch time; the dispatch time is computed as the staleness factor plus the current time. The replication queue uses the dispatch time to order the replication events and schedule the sending of updates. The modification of a file generates zero or more replication events. A new replication event is not created if one already exists in the replication priority queue. Two replication events u and v are considered to be equivalent if $filename_u = filename_v$ and $\langle host, SF \rangle_u = \langle host, SF \rangle_v$.

The creation of a replication event is a two stage process. First, the event is written to a persistent queue so that in the event of a crash the replication events are not lost. Second, it is inserted into the replication priority queue that orders them based on dispatch time from soonest to latest.

²The `creat` function is the same as `open` with the `O_CREAT|O_TRUNC|O_WRONLY` flags.

Dispatching updates

Updates are propagated by the update dispatcher in the replication control. Dispatching a replication event involves sending the file to the indicated replica node. The update includes the modified file plus a small amount of metadata that reflects the changes to the file's attributes and the version number of the replication policy. Transmitting the entire file is not necessarily the most efficient approach but it is the simplest. One could propagate the modified blocks or chunks, as done in Pastiche [17] or LBFS [48], but that adds overhead and increases complexity. Upon the completion of a replication event, the update dispatcher removes the corresponding entry from the persistent queue and goes to sleep until it is time to dispatch the next event.

If an update cannot be delivered because the replica node is unreachable, the replication event is pushed to the failure handler. The failure handler holds the update for a period of time and then reinserts it back into the replication queue for another delivery attempt. The length of time the failure handler holds the update depends on the staleness factor for that replica. The number of retries is currently globally set but it can easily be specified on a per replication event basis. If after a number of delivery attempts the update cannot be delivered the failure handler asks the policy oracle for a replacement node. Failure recovery is then started for the failed node (see Section 3.7).

3.6.2 Replica operations

Replica nodes are primarily responsible for storing data and ensuring that replication policies are followed.

Storing data

Replica nodes receive *store* requests from clients. The *store* requests consists of the file data and a small amount of metadata that specifies the file's attributes and its replication policy version number. The version number is used to ensuring policy consistency between client and replica nodes.

On reception of a store request the replica node stores the file data in a temporary file and records the reception of the event in a persistent log. Again, this is done to simplify recovery in the event that a node crashes temporarily or is rebooted. The replica node retrieves the replication policy for the file and compares the version number of the local copy to the version number that is specified

in the store request. If the two numbers match then the replication policy state is consistent and the replica node moves the temporary file to replace the previous version of the file. Otherwise, either the client or the replica node is using a stale replication policy. The procedure to deal with stale replication policy data is described in Section 3.7.2.

On reception of a store request if the replica node is the leader for the replication policy then it starts to actively monitor the client. The monitoring continues until the client has finished propagating the update to all the replica nodes in the replication policy. The length of monitoring depends on the replication policy and the largest staleness factor. This monitoring is performed to quickly catch a failed client and complete the replication policy in the event of this type of failure (see Section 3.7.3).

Monitoring nodes

Replica nodes monitor their peers to quickly detect failure and minimize the amount of time a replication policy is not followed. The set of replica nodes that a node is responsible for monitoring is easily determined through the node-to-node map that is maintained by the policy oracle.

The set of nodes that need monitoring could be large and the potential overhead would be significant. This problem is mitigated in two ways, first, through intelligent node selection (see Chapter 5) the number of nodes that have to be monitored can be maintained at a reasonable number; roughly equal to the number of nodes in an average policy³. Two, unnecessary heartbeat messages are suppressed. If two nodes are communicating then there is no need to send a heartbeat message to determine liveness. In effect, the heartbeats are piggybacked on top of regular communication. The heartbeat mechanism provides an API call to cancel the upcoming heartbeat message for a particular node.

In addition, nodes are not automatically monitored, they are only monitored when they are storing data to satisfy a replication policy. When a new replication policy is created, the nodes in the policy do not automatically start monitoring each other. Only when they receive the first store request for the policy do they start the monitoring. This optimization follows from the observation that a node that is not storing the data cannot help in the recovery process, since it has nothing to re-replicate from.

³From our simulation results, this is around 6-9 nodes.

The heartbeat mechanisms (see Figure 3.7) ping each other to determine liveness. Each client of the heartbeat mechanism registers the node to monitor, the frequency at which to send the heartbeat messages, the transition, and a callback function to call on the transition. The transition can either be from alive-to-dead or from dead-to-alive.

3.6.3 Policy oracle operations

The policy oracle has three duties. First, it performs node selection for new and existing replication policies. Second, it stores replication policies. Third, it maintains topology and connectivity information and propagates it to the other policy oracles to continually build up a consistent view of the system topology.

Replication policy store

The policy oracle uses the Berkeley DB [81] for its storage needs. The replication policies, the node-to-policy map, and the node-to-node map are stored in three separate databases. The replication policies are indexed by policy keys which are MD5 hashes of the file's full path. The other two maps are indexed by node ID, which is the node's IP address. The policy manager (see Figure 3.8) provides the access and the synchronization for these databases, which are used by both the client and the replica nodes; the client relies on this store to determine which files reside on which replica nodes.

New policy creation

In PDR, policy creation is always performed by the client and it is a multi-step process. For the rest of this discussion assume that a highlevel policy has already been translated to a lowlevel replication policy.

First, the client's policy oracle queries each replica node listed in the new replication policy and the current information about the node is obtained. In particular the oracle retrieves, the nodes attributes, the policies that the node is responsible for, and the nodes that are adjacent to it. In this way current and consistent node and topology information is obtained and maintained for the replication policies the node is participating in. In addition, if a node fails then this information aids in performing recovery.

Second, the client's policy oracle incorporates the newly obtained information with its own. The node-to-policy and the node-to-node maps are updated. Then the policy oracle sends the new replication policy and the updated topology to the replica nodes listed in the replication policy. Third, on receiving the new replication policy the replica nodes update their replication policy database, the node-to-policy, and the node-to-node maps.

Updating existing policies

To improve efficiency and to avoid unnecessary replication the updating of replication policies is a two phase process. First, the old policy is temporarily saved in the archived policy list (see Figure 3.3). Second, the new nodes being added to the replication policy are queried for their information. Third, the nodes being removed from the replication policy are removed but are not informed of their removal. Finally, the version number on the replication policy is incremented and the existing nodes and the new nodes are sent the updated replication policy. This is the end of the first phase. No data copying is performed at this time. If there is a failure before the file is modified again then the old saved replication policy is used. Once the file is replicated based on the new replication policy the old replication policy is removed and the nodes that were removed from the policy are informed of their removal. This approach allows for a smooth and efficient move to an updated replication policy without performing data copying, but still maintaining a consistent view as to where the data resides.

The saved replication policy is actually a list of replication policies. In most instances the list is of length one since policies are not modified often. In the event that a replication policy is updated and then updated again without the replication policy being fully executed then the affected file may be fully replicated under the old policy, partially replicated under the first modified policy, and partially under the last modified policy. Thus, to ensure that data is not lost due to node failure and can easily be found, all three policies are kept until the current replication policy is completely executed; there is no limit to the number of policies that can be in the archived policy list.

File creation and modification

The replication scheduler in the replication control (see Figure 3.6) uses the policy manager to get replication policies. When a file is created, the replication scheduler queries the policy manager for

the default set of replication policies assigned to the parent directory. The most suitable replication policy is chosen, based on the file extension, and it is applied to the file. The creation of a directory is similar except that the entire set of default replication policies is inherited by the new directory. Finally, the replication policy is sent to all the replica nodes. Since a new policy is not being created, instead an existing policy is being applied to a new file or directory it is not necessary to query the replica nodes listed in the replication policy for information. The replica nodes just have to update their replication policy databases, the node-to-policy and the node-to-node maps are unchanged. When a file is modified the replication scheduler queries the policy manager to obtain the file's replication policy to create the replication events.

Remote requests

The policy oracle coordinator (see Figure 3.8) processes two types of remote requests from other policy oracles. The first is a request for information, which consists of the node attributes, the replication policies the node is participating in, and the local topology or connectivity. The second is a replication policy creation or update request.

When a request for information is received, the policy oracle coordinator calls the policy manager to obtain the information. By maintaining the node-to-policy and the node-to-node maps the policy manager can quickly and efficiently gather this information. It is important for the policy manager to satisfy these requests quickly because the majority of this information is dynamic. Depending on the activity of the system the information can quickly change, and thus a node can receive a significant number of these requests for information.

Replication policy creation and update requests are quite similar but are treated differently. On reception of a replication policy creation request the policy oracle coordinator updates the node-to-policy and the node-to-node maps and inserts the new policy into the database. On reception of a replication policy update request the policy oracle coordinator updates the maps and the replication policy, but also archives the old replication policy until the new replication policy has been executed at least once.

Topology updates

The policy oracle is also responsible for distributing topology updates to the other nodes in the system. This enables the nodes to maintain a relatively consistent view of the topology or connectivity. A gossip style protocol [19, 87] is used to accomplish this task.

On a regular basis the policy oracle builds an adjacency matrix that represents the topology and sends it to a small, random subset of the nodes it knows about. In particular, it focuses on the nodes it recently discovered through gossip messages from other nodes. Along with the adjacency matrix, the static attributes of the nodes in the adjacency matrix are also sent. This approach reduces the amount of information sent, since a portion of it can become outdated relatively quickly, and if a node is chosen for a policy then its full attributes are obtained.

On reception of a topology update the policy oracle updates the node-to-node map. New nodes are added and inter-connections between existing nodes are updated.

Gossip messages also piggyback heartbeat messages but not vice versa and they are not a substitute for the monitoring. First, heartbeat messages are sent to everybody verses a random subset. Second, heartbeat messages are usually sent at a higher frequency than gossip messages. Failure would not be detected quickly enough if only the gossip protocol was used.

3.7 Failure operations

An important aspect of any distributed system is the ability to handle failure conditions; the design of PDR assumes that nodes fail in a fail-stop manner, and do not inject corrupted information into the system. It is important for PDR to quickly recover from failure conditions because it must maintain the replication level of the data and no redundant replication is performed.

Given that replication is performed asynchronously at the granularity of files and that PDR's architecture is peer-to-peer, special care must be taken to ensure consistency is maintained. PDR considers two general failure modes. The first is transient failures such as network partitions and temporary node failures. The second is permanent node failure.

For transient failures the main issue that needs to be addressed is the consistency of policy metadata. The design of PDR assumes that transient failures are short term and replication policies are always followed. To deal with transient failures, the system must handle updates, both data and

policy, that are late, lost, or delivered out of order.

For permanent failures there are three additional issues. First, the system must quickly handle the failure of a node so that replication policies are followed. Second, the corrective process should not put excessive load on the system. Third, the detection of a node failure could potentially cause a message storm during the recovery process. The topology of the system must be maintained to prevent such storms from occurring, or ensure that they are localized (see Chapter 5).

3.7.1 Transient errors

Transient failures, such as network partitions and rebooting of nodes, are handled by a combination of heartbeat messages and re-transmission of unsent messages. If a node A is unable to send a message to a node B, it initially retries several times. If after several attempts A is still unsuccessful, the heartbeat monitor is invoked to monitor B and inform A when B becomes reachable again. At some point, if B is still unreachable, then it is considered to have failed permanently. Similarly, if the heartbeat monitor does not hear from a node for a period of time then that node is considered to have failed permanently.

3.7.2 Stale replication policy data

During transient outages it is possible that some data updates arrive late or out of order. In particular, this can be problematic during replication policy updates. For example, a node is added to a replication policy but is not informed of this until it receives the first store request from a client.

Assumptions

The recovery procedures for stale replication policy data outlined below are predominantly for typical failure modes. There are several atypical scenarios that are briefly discussed here, along with accompanying brute force solutions, but in general it is assumed that these scenarios very seldom arise.

A primary assumption is that the composition of a replication policy, that is, the set of replica nodes, does not quickly and drastically change. Specifically, we require that at any time if a node A discovers that it has a stale policy then there exists a node B in both the old and the current policies that it can query to obtain the current policy. More formally, if \mathcal{P} is the set of nodes in the old policy

and \mathcal{P}' is the set of nodes in the current policy, then $|\mathcal{P} \cap \mathcal{P}'| \geq 1$. If the above is not true then the node can resort to broadcasting the policy key to find the current version of the replication policy and determine the policy's members.

It is felt that the above assumption is valid because in large organizations machines, such as workstations, are permanently connected and have uptimes in the range of hundreds of days [11] and even longer lifespans. In addition, the mean time before failure for present day hardware, such as disks, is rated for 300K to 1.2M hours. Servers have an even longer life span than workstations because they are generally built with higher grade components and are administered. Given a policy composed of three workstations, with an average lifespan of 300 days, the above assumption would be valid for about one and a half years before the node membership changes completely. The client would have to resort to broadcasting only if the data is not modified for that period of time.

The above assumption is more important for clients than for replica nodes. Whether or not a replica node possesses stale policy information or not it will eventually be contacted with either a policy update or a store request and its policy information will be brought up-to-date at that time. Clients on the other hand are much more problematic because they only push updates to the replica nodes and replica nodes do not explicitly push updates to them. Thus, without the above assumption it is possible that at some point the replication policy on the client and on the replica nodes diverge to the extent that the client does not know and is unable to find any of the current replica nodes in the policy. At this point the client would need to do a broadcast to attempt to locate the current replica nodes in the policy.

Stale replica

PDR relies on the version number contained in the replication policy to determine if the policy a replica node is holding is current; the policy version number is attached to all store requests. On receiving a store request, a replica node compares the received version number with the version number it already stores. If the two match then the store request and the replica node are using the same policy, which might, in fact, be stale. If the received version number is bigger, then the replica node contacts the leader or one of the other nodes in the policy and requests the current policy. This case relies on the assumptions presented above. The expected reply from the leader or one of the replica nodes is a replication policy with the version number that is equal to or greater than that of

the store request. If the received number is smaller, then the client is using a stale policy and needs to be updated.

Stale client

A client will hold a stale replication policy when the leader of the replication policy changes the policy due to a replica node failure. There are two scenarios that can occur when a client holds stale replication policy data. The first is when the client holds a stale replication policy but the replica node has the current version. In this case the replica node informs the client about the inconsistency and the client rereplicates based on the updated policy.

The second is when both the client and the replica node hold a stale replication policy. In this case, the client continues to replicate based on the stale replication policy until it encounters a replica node with the current replication policy; this is where the assumptions made above are extremely important. At this point the second case turns into the first scenario and the client rereplicates based on the updated policy.

Disseminating stale information

To avoid disseminating stale information the policy oracles do not send detailed node information or node attributes that are dynamic when gossiping to other policy oracles. If a node needs the attributes of another node it contacts that node directly. Although node attributes do not change often, by forcing nodes to obtain this information directly the potential to disseminate and use stale information is reduced.

3.7.3 Client failure

PDR handles both transient and permanent client node failures. Transient failures are those on which the PDR process has died and was restarted; for example a node was rebooted. Permanent failures are those on which the PDR process has died and never returns. There are three client node failure scenarios that need to be handled.

In the first scenario the client's replication queue is empty. There are no pending replication events and all replication activity is up-to-date. In this instance it does not matter if the client fails permanently or eventually restarts, no recovery actions are necessary.

In the second scenario the client's replication queue is not empty and no replication policies are partially satisfied. That is, one or more files were modified and replication events were inserted into the replication queue, but no replication events were dispatched before the node crashed. If the node fails permanently then the modifications are lost. If the node eventually restarts then the modifications are recoverable because the replication queue is persistent. On a restart the client re-builds the replication queue and immediately dispatches all replication events that were missed while the client was down.

In the third scenario the client's replication queue is not empty and some replication policies are partially satisfied. That is, one or more files were modified and replication events were inserted into the replication queue. In addition, several of the replication events were dispatched to the replica nodes before the node crashed. As described earlier, the leader node monitors a client node until in-progress replication policies are satisfied. On detecting a client failure, permanent or transient, the leader node immediately replicates the latest update to the other nodes in the policy. If the client returns, on restart all replication events that were part of a partially satisfied policy are removed from the replication queue since the leader node has already propagated the updates to the other replica nodes.

3.7.4 Client failure to contact a replica

When a client fails to contact a replica node it anticipates a policy change. If the failed replica node is not a leader then the client contacts the leader node to obtain the current replication policy. If the failed replica node is the leader then the client contacts the replica node that is elected the replacement for the leader (see Section 3.7.5). The leader node always holds the current replication node.

3.7.5 Replica failure

The bulk of the failure handling machinery is used to handle failed replica nodes.

Non-leader replica nodes

When the failure of a replica node is detected, the leader of the replica node group is informed by the node(s) that detected the failure. If the leader fails then a new leader should take over the

restoration process. In order to quickly detect the failure of a leader the other nodes in the policy start to monitor the leader at a higher frequency.

Upon receiving notification of a node failure the leader queries the policy oracle to find a replacement node. Once a new node is selected the leader sends a policy update message to all the nodes listed in the replication policy; a policy update message is sent for all replication policies affected by the node failure. Note, that the failure of a replica node may affect a number of replication policies, each of which may have their own leader. The leader in each replication policy may select the same or a different replacement node, depending on its topology and connectivity information. Thus, a failed replica node may be replaced by a number of nodes.

Once a replacement node has been selected and all affected replication policies are updated, the leader starts the re-replication process for the data that was hosted on the failed node. The leader queries the policy oracle for a list of files it is responsible for, and propagates the list to the newly selected node. The newly selected node pulls the required data from the available replica nodes and then informs the leader when finished. The newly selected node selects replica nodes from which to pull the data based on two criteria. First the replica node must have an up-to-date copy of the data. Second, if there is a choice the replica node that is closer, i.e. on a local LAN verses off-site, and has the higher bandwidth connection is selected. These properties are part of the default *node attributes* that each replica node possesses (see Chapter 4). If the newly selected node receives a store request for a file it is supposed to pull from a replica node, it accepts the store request and removes the file from the list of files that have to be pulled.

Leader nodes

The procedure for handling leader node failures is similar to that for regular replica node failures. When the failure of a leader node is detected, the node with the next smallest staleness factor is automatically selected as the temporary leader. Then the recovery process described in Section 3.7.5 is performed. As a last step, if the newly selected node has the smallest staleness factor then it takes over the duties of the leader. If there are multiple nodes with the same staleness factor then the current policy is to select the leader as the node that has the smallest IP address.

Client node notification

Client nodes are notified by replica nodes of changes to replication policies as a result of a replica node failures. These notifications are primarily done lazily when the client contacts the leader or some other replica node. Leader nodes immediately notify client nodes of replication policy changes if the leader is already monitoring the client node.

Changes to an in-progress replication policy affect both client and replica nodes. When the client node receives a notification that a replication policy has changed, it examines what part of the replication policy has been satisfied. If the change does not affect what already has been replicated, then the existing replication entries in the replication queue are simply updated. Otherwise, the client node immediately replicates the data to satisfy the updated policy.

Client nodes cannot be notified of changes to the replication policy if the leader node fails. Instead a client node is informed of changes to replication policies when it sends a store request to one of the other replica nodes. This scenario only occurs on clients that have in-progress replication policies that have already replicated to the leader node.

Failure during recovery

Finally, the system must handle recursive failures when a replica node fails while the system is in recovery mode. Currently, PDR simply re-computes and re-replicates. Some work could potentially be salvaged that was done in the previous attempt to recover, but this adds substantial complexity to the system.

It should be noted that transient failures, such as network partitions, are still expected to occur during recovery. These transient failures are treated in the same manner as when the system is not in recovery mode. Thus, it is assumed that the recovery process may take some time to complete during a storm of transient failures.

3.8 Data recovery

Depending on the severity of data loss, data recovery is either a one or two step process. In the first scenario file data is lost but the policy oracle database is intact; for example accidental deletion or minor file system corruption. In this case the user employs a userlevel tool to restore the lost

data. The most recent version available of the file is determined by contacting the leader node in the replication policy. In addition, the leader returns the nodes on which this version is available, since the current version could still be in the process of being replicated.

The userlevel restore tool communicates to the local policy oracle through the plug-in interface to obtain the necessary replication policy information. The tool then communicates to the replicator on the replica nodes directly to restore the necessary data.

The other scenario is the loss of both the file data and the policy oracle database; for example a total disc failure on the client (the loss of the database on a replica node is irrelevant because it is automatically rebuilt when the replica node is replaced). In this case the first step is to recover the database and the second step is to recover the file data. The policy oracle database is also replicated. During system setup the administrator and/or the user specifies a set of replica nodes to which the database is replicated. This set of nodes are well known, trusted, and reliable.

The replication policy for the policy oracle database depends on the replication of newly created files. When a file is created, and a new replication policy is written to the policy oracle, a replication event is scheduled for the policy oracle that corresponds to the first replication event for the newly created file. If a replication event already exists for the policy oracle database and a replication event is inserted for a newly created file that is to be replicated even sooner, then the replication event for the policy oracle is rescheduled for the earlier replication time. In this way, the information in the policy oracle database is current as to what is replicated. There may be file data that is listed in the policy oracle database that was not replicated but there is never an instance when there is replicated data that is not listed in the policy oracle.

To recover the policy oracle database the PDR userlevel application is started in recovery mode. In recovery mode the replicator retrieves the policy oracle database from one of the well known replica nodes and starts pulling file data for all the files listed in the database. The choice of replica nodes from which to pull the data is made as in the first scenario.

A client can recover a large portion of its data without recovering the policy oracle database but there is no guarantee that all file the client's data is recovered. To recover file data one can broadcast the client's root to the system. If a replica node has a directory hierarchy with that root then it stores some set of files for the client. At this point the replica node returns the list of files it stores for the client along with the replication policies. The client can then recover the files and rebuild the policy

oracle database. This approach is considerably slower than recovering the database, it adds more overhead to the client and the replica nodes, and the client is not guaranteed to restore all its data since a node may be down at the time of the broadcast and thus not receive the message.

Chapter 4

Replication policy

Raise a lot of money for me, I'll give you good architecture.
Raise even more money, I'll make the architecture disappear.

— Yoshio Taniguchi, MOMA architect

Replication policies drive the replication of data in PDR. These policies affect replication at the granularity of a file, and the level of protection (replication) that is provided is proportional to the value of the data. Users specify policies using an interface that is appropriate for them and the operating system. Unfortunately, specifying this information in a consistent and uniform manner is extremely difficult because users have a very widely varying perspective on the definition of protection level or replication, and on the value of data.

The adage “*beauty is in the eye of the beholder*” equally applies to the definition of data value as it does to art. The value of data is not simply defined by the number of hours it took to create it, or its monetary worth. Other subjective factors, such as sentimental value, greatly influence the perceived value of data. Thus, there is no single or uniform way to explicitly define the value of data.

Users’ understanding of what backup and replication options are available, how much protection (safety) is provided by each option, and their relative cost differs to the extent that there is not a single, well understood, representation for either of these variables. Given the greatly varying perspectives on these three factors it is extremely difficult to create a uniform interface to specify these two variables. Instead, PDR provides a generic plug-in interface that enables the administrator of the system to create a module that provides a custom user interface to create, set, and modify

highlevel replication policies. In addition, the plug-in provides the necessary mapping functions to map highlevel replication policies into lowlevel representations that PDR understands. In this way the system can be tailored to fit the needs of its users.

4.1 Overview

The translation of a highlevel policy into a lowlevel policy is a multiphase process. A user specifies a highlevel policy using the user interface provided by the policy plug-in. This highlevel policy usually consists of two types of information. The first explicitly describes the desired properties of the replica nodes. For example, for video editing it is desirable to have replica nodes that have high bandwidth connections. The second is a replication budget or the desired protection level. This information is transformed into an optimization problem that maximizes the protection level given a budget or minimizes the cost given the desired protection level. The highlevel policy is translated into a *policy specifier* \mathbb{L} which is a set of *node selection constraints*. This set \mathbb{L} of constraints is then used by the node selection algorithm to select a set of nodes for the lowlevel policy; the selection is based on the *node attributes* of the nodes.

Section 4.2 presents the *node attributes* that define nodes in PDR and are primarily used for replica node selection. Section 4.3 presents the *node selection constraints* and describes how replica nodes are selected using these constraints and node attributes. Section 4.4 presents the *policy specifier*. In Sections 4.5 and 4.6 the mechanics and the API of the policy plug-in are discussed. In Section 4.7 the concept of highlevel policies is defined and several examples of utilities to set, create, and modify highlevel policies are presented. Section 4.8 describes how a highlevel policy is translated to a lowlevel policy. Lastly, in Section 4.9 the remapping of node attributes between domains is discussed.

4.2 Node attributes

Replica Nodes in PDR are described by a set \mathcal{C} of properties called *node attributes*. These node attributes are used extensively in node selection for both new replication policies and replacement nodes when replica nodes fail.

The set \mathcal{C} consists of static and dynamic node attributes. For example, static properties such

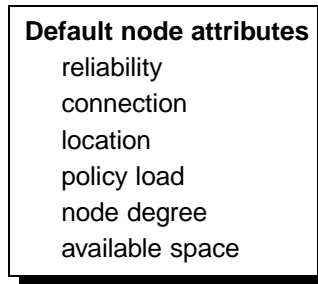


Figure 4.1: The default set of node attributes.

as the CPU or the network connection usually do not change during the life time of the node, while dynamic properties such as location, the available storage space, or node utilization change depending on the node's relative location or workload respectively.

There is a default set of node attributes that all PDR nodes possess (see Figure 4.1). *Reliability* is a static property that specifies the dependability of the node, and is one of the following values: a *workstation*, a *server*, or a *data center*. Workstations are desktop machines, they are abundant and inexpensive, but they are the least reliable since users may turn them off or reboot them at any time. Servers are machines that are locked away in a secure room and there is an administrator to take care of them. They are more reliable than workstations, but may still be affected by local network and power outages. Servers are generally more expensive to replicate to than workstations because of the administration overhead and they are less likely to be commodity machines. Data centers are the most expensive type of replica node but they are also the most reliable. Not only are they professionally administered but they are likely to be connected by multiple links, have uninterrupted power supplies, and have a large amount of redundancy.

Connection is a static property that describes a node's connection to the rest of the world in terms of the upstream/downstream bandwidth. This node attribute takes on the following values: *0/0*, *56Kb/56Kb* (modem), *128Kb/1.5Mb* (ADSL or cable), *1.5Mb/1.5Mb* (T1), or *45Mb/45Mb* (leased T3 or better). The *location* is a dynamic property that describes the relative location of a pair of nodes. *Local* nodes are co-located and are usually connected by a single high bandwidth LAN. *Remote* nodes are considered to be offsite and connected by a third party service provider that charges for the bandwidth. Thus, in general, remote nodes are more costly to replicate to because there is an additional cost of bandwidth. The node attributes *policy load*, *node degree*, and *available*

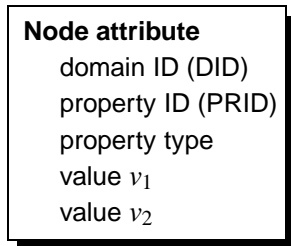


Figure 4.2: A node attribute.

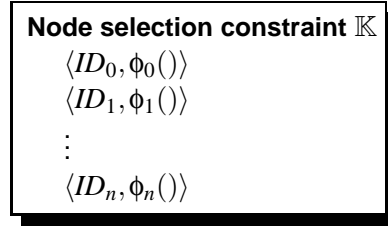
string	character string
enum	enumeration
int32	signed 32-bit integer
uint32	unsigned 32-bit integer
int64	signed 64-bit integer
uint64	unsigned 64-bit integer
float32	32-bit float
float64	64-bit float

Table 4.1: The eight property types.

space are all dynamic properties, and describe the number of policies the node is responsible for, the number of nodes that are dependent or connected to the node, and the available storage space on the node, respectively.

The system administrator of a PDR domain may also add custom node attributes. For example, a more precise location may be required and thus a property that specifies the node's GPS coordinates can be added. If an organization has offices in multiple locations then it may be beneficial to also specify the node location with respect to office location. Some node attributes, for example location, are relative to another node and are remapped as necessary (see Section 4.9). For example, if replica nodes *a* and *b* are on the same LAN, then from *a*'s point of view *b*'s location is *local*. If node *c* is attached by the Internet and knows about *a* and *b*, then *a* and *b* are listed as *remote* to *c*.

A node attribute (see Figure 4.2) is a five-tuple that describes a property of a replica node. The *domain ID* specifies the domain a particular property belongs to, or was defined in. The domain ID is a unique identifier across all instantiations or domains of PDR. The *property ID* uniquely identifies the node attribute within the domain. The combination of the domain ID and the property ID uniquely identifies a property across all domains of PDR. The domain ID and the property ID

Figure 4.3: A node selection constraints \mathbb{K} .

equal	$x = v_1$
not equal	$x \neq v_1$
less or equal	$x \leq v_1$
greater or equal	$x \geq v_1$
in the range	$v_2 \leq x \leq v_1$
out of the range	$x \leq v_2$ or $x \geq v_1$

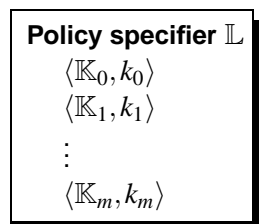
Table 4.2: Several examples of constraint functions ϕ .

are used to uniquely identify node attributes when they are remapped from one domain into another domain (see Section 4.9). The *property type* defines the type for v_1 and v_2 ; v_1 and v_2 are always the same type and v_2 is optional. The values v_1 and v_2 specify the value of the node attribute; v_2 is used to specify a range of values for a node attribute. The types are listed in Table 4.1.

4.3 Node selection constraints

In PDR *node selection constraints* specify the criteria by which replica nodes are selected for new replication policies and for replacement of failed replica nodes. A *node selection constraint* (see Figure 4.3) is a set \mathbb{K} of pairs $\langle ID, \phi() \rangle$ consisting of an *ID*, and a constraint function $\phi()$. The *ID* is a pair $\langle DID, PRID \rangle$. The constraint function $\phi()$ takes as input v_1 and v_2 , if v_2 exists, and returns true if v_1 and v_2 satisfy the constraint. Several examples of the constraint function $\phi()$ are listed in Table 4.2.

Determining if a replica node *satisfies* a node selection constraint is straightforward. Given a node x with node attributes C_x and a node selection constraint \mathbb{K} we say that x *satisfies* \mathbb{K} , denoted $C_x \subseteq \mathbb{K}$, if for all $k \in \mathbb{K}$ there exists $c \in C_x$ such that $\langle DID, PRID \rangle_c = ID_k$ and $\phi_k(\langle v_1, v_2 \rangle_c)$ is true. Similarly, given two node selection constraints \mathbb{K} and \mathbb{K}' , \mathbb{K} *satisfies* \mathbb{K}' , $\mathbb{K} \subseteq \mathbb{K}'$, if for all C , if

Figure 4.4: A policy specifier \mathbb{L} .

$C \subseteq \mathbb{K}$ then $C \subseteq \mathbb{K}'$.

4.4 Policy specifier

A *policy specifier* \mathbb{L} (see Figure 4.4) is the result of the translation of a highlevel policy by the policy plug-in. It is a set of pairs, $\langle \mathbb{K}, k \rangle$, consisting of a node selection constraint \mathbb{K} and a non-negative integer k . Given a set of replica nodes N , N satisfies policy specifier \mathbb{L} if there is a function that maps the nodes in N to the tuples in \mathbb{L} , such that for each $\langle \mathbb{K}, k \rangle \in \mathbb{L}$ at least k nodes satisfying \mathbb{K} are mapped to it.

4.5 Policy plug-in

The policy plug-in has three primary responsibilities. First, it provides the user interface for users to create and modify replication policies (see Section 4.6). Second, it maps the highlevel policies specified by users into a policy specifier \mathbb{L} (see Section 4.8). Third, it performs remapping of node attributes from one domain into another (see Section 4.9).

In most modern operating systems, application plug-ins are usually supported as dynamically linked libraries that are explicitly loaded by the application; the policy plug-in is a dynamic library that is loaded by the userlevel portion of PDR on startup. The name of the policy plug-in is explicitly specified which allows a single domain to use different plug-ins. The plug-in exports two functions; the *name* of the plug-in is prefixed to the function name. The *name_init* function, as the name suggests, initializes the plug-in. At initialization time the interface to the policy manager (see Section 3.5 and Figure 3.8) is passed into the plug-in to give it access to the services provided by the policy oracle.

The other function, *name_node_desc_rewrite*, is used by the policy oracle to remap the node attributes from a foreign domain to the local domain (see Section 4.9).

4.5.1 Plug-in example

A brief example of a simple plug-in is presented here; the example is based on the *ptool* plug-in (see Section 4.7.1). The two functions that the plug-in implements are *ptool_init* and *ptool_node_desc_rewrite*. When PDR loads the plug-in it maps the *ptool_init* and *ptool_node_desc_rewrite* functions and calls *ptool_init* with a reference to the policy oracle. This enables the plug-in to call the policy oracle to create and modify replication policies.

During initialization the plug-in loads a remapping, or translation, table that the *ptool_node_desc_rewrite* function uses to remap node attributes; this file is generated by the administrator. The file consists of a list of four-tuples $\langle \text{DID}_o, \text{PRID}_o, \text{PRID}_n, c_n \rangle$ where DID_o and PRID_o is the original domain ID and property ID, PRID_n is the new property ID, and c_n is the remapped node attributes. In addition, the default policy for the root is created if one does not exist. The *ptool* plug-in establishes a listening socket so that the command line portion of the tool can communicate with the plug-in.

The *ptool_node_desc_rewrite* function takes as input a set \mathcal{C} of node attributes. For each entry $c \in \mathcal{C}$ the function looks up the original ID_c , $\text{ID}_c = \langle \text{DID}_o, \text{PRID}_o \rangle$. If ID_c is in the remapping table then the corresponding new attribute c_n is added to \mathcal{C} . When the policy oracle discovers a new replica node from a foreign domain it calls *ptool_node_desc_rewrite* to remap the node's attributes from the foreign to the local domain.

4.6 User interface

The user interface for creating, setting, and modifying policies can either be part of the policy plug-in or a stand alone application that communicates with the plug-in. The decision as to where to place the interface depends on the operating system and the operating system's user interface. In Microsoft Windows there is a *system tray* where icons of constantly running applications, such as virus checkers or configuration panels, are placed. Thus, in Microsoft Windows it is appropriate to include the user interface as part of the plug-in and have it reside in the system tray.

In operating systems where the predominant user interface is command line, any Unix for example, it is more appropriate to create a stand alone application that houses the interface than to include

the user interface in the policy plug-in. If the user interface is included in the policy plug-in then the user interface would be constantly running since the plug-in is loaded at startup. This is not the desired mode of operations and does not conform to the user interface of the system. The approach taken by the example policy plug-in is to create a thread running in the plug-in that listens to a local socket and then have a standalone application communicate to the policy plug-in via this socket. Although there is some overhead for packetizing all of the communication between the plug-in and the user interface it creates a generic solution that enables different user interfaces to use the same plug-in with potentially different interfaces.

At the present time, to develop a policy plug-in the administrator or the user has to write it in C. In the future, depending on the platform and operating system it would be possible to use a higher level language such as Visual Basic, or any language that can be compiled into a dynamically linked library.

4.7 Highlevel replication policies

There are many different ways to specify a highlevel policy. How policies are specified greatly depends on the needs and ability of the users, the operating system, and the user interface. In general, the interface for specifying replication policies should be consistent with the user interface of the system and appropriate for the user base. For example, computer savvy users, such as faculty and grad students in a computer science department, tend to prefer to have more control. Thus, they could be asked to select the replica nodes for the replication policies, either by node type or explicitly specifying hosts. Less computer savvy users, would prefer a simpler interface, probably consisting of a single slider that relates cost to protection level.

Next, two different approaches, or user interfaces, are presented for managing replication policies. The first is a command line tool called `ptool` that was implemented as an example plug-in and interface for PDR. The second is a commercial package offered by Bell Canada.

The `ptool` utility is used as a running example for the rest of this chapter as a way to demonstrate the process of transforming a highlevel policy to a policy specifier `L`. In addition, an example of a simple user interface consisting of a slider that represents the replication budget is used.

```
ptool <filename> [-p t:number:type:staleness factor[:a]] [-f <policy file>]
ptool <filename> [-p s:host:staleness factor[:a]] [-f <policy file>]
```

Figure 4.5: ptool command line arguments. Multiple replica nodes are specified by multiple -p command line options. The :a option specifies that the time is absolute. User can also specify the policies in a flat text file using the -f option.

4.7.1 ptool

The ptool application is a command line utility to manage replication policies. Users specify replication policies in two ways. First, users specify a replica node type, the desired number of nodes of that type, and the staleness factor (see Figure 4.5). Currently the set of replica node types are *local workstation*, *local server*, *remote workstation*, *remote server*, and *data center*. The classes define the location and the reliability of a replica node. Second, users can explicitly specify a set of hosts and associated staleness factors.

4.8 Translating policies

A primary responsibility of the policy plug-in is to map highlevel policies into policy specifiers. The plug-in is not responsible for arriving at the actual set of replica nodes, that is the job of the node selection algorithm (see Chapter 5). The amount and type of mapping done depends on the information that the user is required to provide.

There are predominantly two types of conversions or mappings that are performed. The first is a direct mapping from the information a user supplies to *node selection constraints*. For example, the ptool utility (see Section 4.7.1) requires the user to supply the node type and the number of nodes of that type. From the node type it is straightforward to extract the *location* and the *reliability* properties. Node selection constraints (see Figure 4.6) are directly created from these two properties, one for the *location* property and one for the *reliability* property.

Each constraint has two entries and each entry has a defined constraint function $\phi()$. Figure 4.7 shows two such functions for the *local workstation* type. The first function checks the location and the second checks the reliability of the node.

<p>ptool Node selection constraint \mathbb{K}</p> <p>$\langle\langle \text{DID}, \text{PRID}_{location} \rangle_0, \phi_0() \rangle$</p> <p>$\langle\langle \text{DID}, \text{PRID}_{reliability} \rangle_1, \phi_1() \rangle$</p>

Figure 4.6: A node selection constraint \mathbb{K} for the ptool utility.

```

0  $\phi_0(v_1)$ 
1   IF (  $v_1 == local$  )
2     RETURN TRUE
3   ELSE
4     RETURN FALSE

0  $\phi_1(v_1)$ 
1   IF (  $v_1 == workstation$  )
2     RETURN TRUE
3   ELSE
4     RETURN FALSE

```

Figure 4.7: ptool $\phi()$ node selection constraint functions.

<p>ptool Policy specifier \mathbb{L}</p> <p>$\langle \mathbb{K}_{local workstation}, k_{local workstation} \rangle$</p> <p>$\langle \mathbb{K}_{local server}, k_{local server} \rangle$</p> <p>$\langle \mathbb{K}_{remote workstation}, k_{remote workstation} \rangle$</p> <p>$\langle \mathbb{K}_{remote server}, k_{remote server} \rangle$</p> <p>$\langle \mathbb{K}_{data centre}, k_{data centre} \rangle$</p>

Figure 4.8: A policy specifier \mathbb{L} for the ptool utility.

Since there are five node types, there can be up to five node selection constraints. Figure 4.8 shows the policy specifier \mathbb{L} that the ptool plug-in generates. This policy specifier is then used by the node selection algorithm to generate the lowlevel policy.

The second type of conversion or mapping is an optimization problem that maximizes the protection level given a budget or minimizes the cost given a desired protection level. For example,

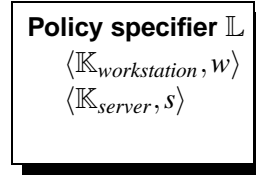


Figure 4.9: A policy specifier \mathbb{L} for the slider interface.

consider a user interface that consists of a slider that specifies the amount c a user wants to spend on replication; the level of replication is directly proportional to c . Assume that there are only two types of replica nodes, workstations w and servers s . The cost to replicate to a workstation is two and the cost to replicate to a server is seven. Furthermore, the reliability of a server is four times that of a workstation. Given these parameters the optimization problem becomes a simple linear programming problem.

$$\begin{aligned}
 & \text{MAX} && 4w + s \\
 & \text{such that} && \\
 & && 2w + 7s \leq c \\
 & && 4w = s
 \end{aligned} \tag{4.1}$$

Once the values for w and s are determined it is straightforward to create the policy specifier \mathbb{L} (see Figure 4.9). The node selection constraints for $\mathbb{K}_{workstation}$ and \mathbb{K}_{server} are the node attributes for the workstation and server replica nodes respectively. The number of each type of nodes is of course w for workstations and s for servers.

In general terms, the optimization problem can be expressed as follows. For each node type i , $i = 1 \dots m$, let c_i be the cost to replicate to node type i and p_i be the protection level afforded by node type i . The protection level p_i is a non-negative integer and protection levels are additive. We want to choose n_i nodes of type i to:

$$\begin{aligned}
 & \text{MAX} && \sum_{i=1}^m p_i n_i \\
 & \text{such that} && \\
 & && \sum_{i=1}^m c_i n_i \leq c_{total}
 \end{aligned} \tag{4.2}$$

The value c_{total} is the budget for the replication policy. It is the responsibility of the administrator to specify the replication cost c_i and the protection level p_i for each type of node.

One way to determine the cost c_i for each replica node type is to decompose the cost based on

the node attributes; for example, the cost of the administration, the bandwidth, and the hardware. A similar approach can be taken to determine the protection level p_i for a particular type of replica node. The level of reliability can be composed of several node attributes and depend on aspects such as the level of administration, the hardware, and the location.

Another guideline for determining c_i and p_i is to ensure that Equations 4.3 and 4.4 are satisfied. Let o_j be the number of nodes of type j ; o_j is a non-negative integer. If the protection level p_i of node type i can be expressed as a linear combination of other node types:

$$p_i = \sum_{j=1, i \neq j}^m p_j o_j. \quad (4.3)$$

Then the cost c_i for replicating to node type i should be:

$$c_i \leq \sum_{j=1, i \neq j}^m c_j o_j. \quad (4.4)$$

If this is not the case, then node type i is too costly for the specified level of protection it provides and will not be used.

The above two approaches for translating highlevel policies to lowlevel policies are orthogonal and thus can easily be combined and extended. Imagine an interface that is similar to the one with the simple slider except that there is also a checkbox, an entry field, and a pull-down list. The checkbox tells the plug-in to minimize the bandwidth usage and the entry field enables the user to enter their bandwidth budget b . The pull-down menu is a list of countries and allows the user to specify the country in which they would prefer the replica nodes to be located. In this scenario, the plug-in would create the initial policy specifier \mathbb{L} by casting it as an optimization problem and increase the cost of each node type by b based on its connection. Then for each $\mathbb{K} \in \mathbb{L}$ the plug-in adds a constraint to \mathbb{K} specifying the preferred country.

In some instances the cost of replication is a function of the update frequency of the file. That is, a file that is modified infrequently can be replicated right after modification to an expensive destination, but frequently changing files must either be replicated less often or to a less expensive location to keep the costs the same. That is, cost is a function of the bandwidth used and/or per update charge. At the present time PDR does not handle cost functions of this type. To handle such cost functions the system would need to monitor the update frequency, the file size, or both and adjust the replication policy accordingly. This would not only add complexity because of the monitoring,

but maintaining consistency of replication policies would be much more difficult because they could change much more quickly (see Section 3.7.2).

Currently only static cost functions can be used. Thus, part of the cost of replication is the cost of the network connection, rather than the bandwidth used. We also assume that all files are equally likely to be modified and thus the update frequency depends on the *staleness factor* (see Section 3.2). Hence, if the cost of replication is a function of the bandwidth used, then a constant multiplier, based on the *staleness factor*, is applied to the cost of replicating to a particular node type. The above approach is a reasonable simplification but is not completely accurate because as a file grows older it tends to be accessed less often [76]. A benefit of this approach is that it provides an upper bound to the cost of replication since the update frequency is dependent on the staleness factor.

The cost of translating a highlevel policy to a lowlevel policy by casting it as an optimization problem is not affected by the number of nodes in the system. Instead, the cost is affected by the number of node types, which is usually small. A well known algorithm for solving such optimization problems is the *simplex* method. Although it performs well in practice, its worst case is exponential. In 1979 Khachyan showed that these optimization problems can be solved in polynomial time by using the *ellipsoid method* (an *interior point method*). There are a number of free and commercial libraries available to solve the above optimization problems¹.

4.9 Remapping node attributes

The other responsibilities of the policy plug-in is to *remap* node attributes. When a policy oracle discovers a node, the node's attributes are *remapped* before it is inserted into the policy oracle's node-to-node map. Given a node a and a node b, when a discovers b, a *remaps* the node attributes of b so that they are relative to those of a's.

If the discovered node is part of the same domain then the remapping is usually straightforward. During the initialization of the policy plug-in, the plug-in may read in, or have hard coded, some remapping information. For example, an installation may consist of two sites connected by a lower bandwidth connection than the internal network at each site and each site is considered to be remote

¹For a partial list see <http://www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html#Q2>.

to the other. Thus, if *a* is from one site and *b* is from another site then when *a* discovers *b* it changes *b*'s *location* property to *remote* and *b*'s *connection* property to the external connection. There may also be some domain specific custom properties that are remapped.

There are no explicit rules in the policy oracle to remap node attributes. It is felt that the administrator of a domain knows best as to which node attributes should be remapped and which ones should not. A default plug-in could be provided to try to keep the *location* and *connection* properties relatively correct by computing the latency between the nodes using ICMP packets or other more complex approaches [26]. In addition, if there are properties that are a straight remapping from one to the other, then it is possible to have a generic remapper within the plug-in where the map is specified in a flat text file and read in during initialization.

Remapping node attributes of nodes from different domains is more complicated because there may be domain specific custom properties that have no remapping. Again the remapping is completely the responsibility of the policy plug-in. For all default properties the remapping is done as described in the previous paragraph. For the custom properties, if a remapping exists, the original custom property is retained and a new property is added. A remapped property is inserted under the remapper's domain ID. In this way the original node attributes of the node are maintained. This is important for two reasons. First, if the remapping changes, it is possible to change the previous remapping. Second, during the gossiping of node topology or connectivity only the original node properties are propagated, allowing for each policy oracle to do its own remapping.

Chapter 5

Topology aware replica node selection

To put it bluntly, we simply do not know yet what we should be talking about, but that should not worry us, for it just illustrates what was meant by “intangible goods and uncertain rewards”.

— Edsger Dijkstra, 'The End of Computer Science'

Over the last few years the popularity of peer-to-peer storage systems has increased at a phenomenal rate. Many systems [16, 22, 35, 50, 55] were created primarily to share information, while other systems [3, 15, 37, 42, 65, 90] were designed to function as deep archival repositories.

In general, with today's peer-to-peer storage it is possible to protect data at a level that is comparable to traditional replication systems but at reduced cost and complexity. Peer-to-peer storage systems provide the needed flexibility, reliability, and scalability to operate in present day environments, and handle present day loads.

Most recent peer-to-peer systems [18, 67, 69] select replica nodes using a quasi-random process. Systems that randomly select nodes assume that the cost of replication and the level of reliability are uniform across the system. In addition, they assume that the failure modes of the nodes are independent. Although, these assumptions may be true in the ideal world and are reasonable for simulation purposes, they are generally over simplified [43, 89].

To overcome the limitations of random node selection, several systems [1, 12, 20, 73] take a more systematic approach, selecting replica nodes based on node attributes such as replication costs, reliability levels, location, and bandwidth. These systems are thus able to effectively place replicas to maximize the data's safety while minimizing the overall cost of replication.

We believe that node selection should not only depend on node attributes, but also on the topology of the system, because a system's topology can greatly influence operation. The effect of poor replica node selection is twofold. First, the safety of data is jeopardized by placing it on poorly suited nodes. Second, a large number of inter-node connections are created which can greatly increase the cost of recovery. These inter-connections, physical [20] or virtual [12], are created when a set of replica nodes cooperate to store an object. Each object may potentially create new inter-node connections. Thus, as the system evolves (nodes fail and are replaced) and objects are created, every replica node may become connected to every other replica node, making group communication and recovery costly.

This chapter presents an algorithm, called *TopSen*, that performs replica node selection for a peer-to-peer storage system. *TopSen* selects replica nodes for data based on the topology of the system and node attributes. The algorithm selects replica nodes that satisfy user specified constraints, such that the number of new inter-connections between the nodes is minimized. This minimizes the potential increase in communication necessary to handle a failure. The driving goal of *TopSen* is that the number of nodes affected by a failure of a node should be minimized.

The *TopSen* algorithm is executed once the policy specifier is created for a new policy or a replacement replica node needs to be selected for a failed one. Replication policies enable users to specify a set of constraints as to when their data is replicated and what type of node stores it. Given these constraints, the *TopSen* algorithm selects replica nodes, using additional constraints, so as to maintain a good topology.

5.1 Overview

A peer-to-peer storage system consists of a collection of nodes (physical machines) that store data (files or blocks). Data is replicated or erasure encoded on a number of hosts to ensure its availability and reliability in the event of node failure. When a new object is created, the system selects a set of nodes to store the object. When a sufficient number of nodes have failed, the system selects replacement replica nodes for the objects on the failed ones, and re-replicates the data.

The number of inter-connections between a node and the other nodes directly impacts the number of nodes affected when the node fails; a large number of inter-connections implies a large number of nodes involved in the recovery. If node selection is performed haphazardly then eventually a

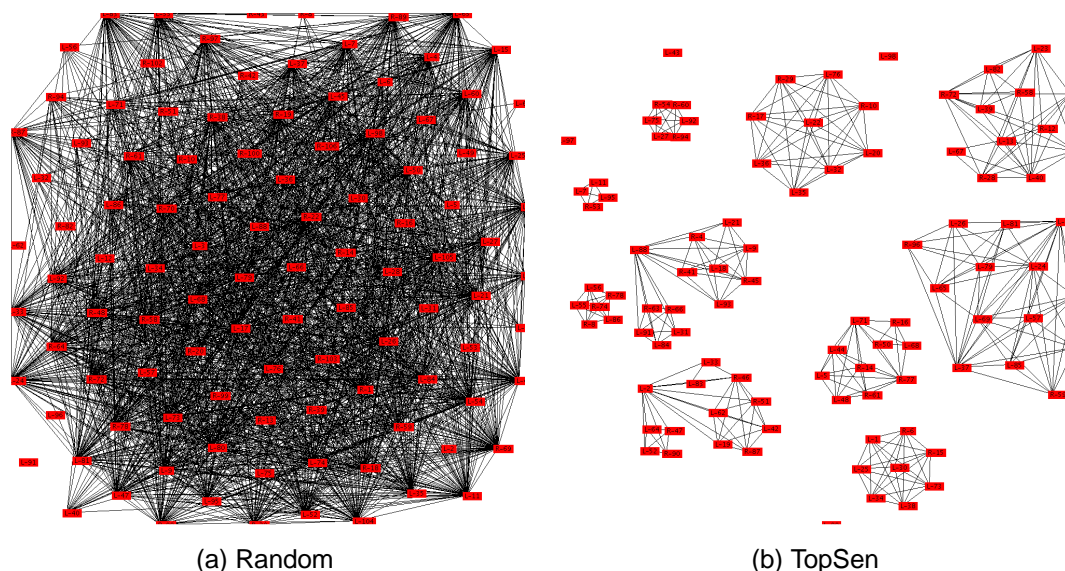


Figure 5.1: Topology after ten years of simulation of 100 nodes using Random and TopSen replica node selection algorithms.

single node failure could affect a very large number of nodes.

DHT based peer-to-peer storage systems [18, 69] fix the number of inter-connections by explicitly specifying the replica nodes a node replicates to based on node IDs. Unfortunately, this same mechanism does not allow these systems to select replica nodes based on node attributes; thus, they must assume that node reliability and failure modes are uniform across the entire system. Removing the assumption of uniform node reliability and failure modes results in a system, similar to [1], that more or less randomly selects replica nodes.

In a peer-to-peer storage system where replica nodes are selected at random (see Figure 5.1a) the topology after simulating 100 nodes for ten years is such that nodes are highly inter-connected. Thus, a single node failure affects a large number of nodes. In contrast, our approach, shown in Figure 5.1b, minimizes the inter-node connectivity such that a replica node failure affects only a small number of nodes.

Thus, the first goal of TopSen is to minimize the number of inter-connections among replica nodes so that the result is a set of small components. Nodes participating in a replication policy forms a clique; a component consists of one or more cliques (see Figure 5.1b). The second goal of TopSen is to balance the load. Otherwise, if all we wanted was to minimize the number of inter-node

connections then a clique the size of the maximum policy size would suffice.

We achieve the above goals in three ways. First, when selecting nodes to replicate a new file or to replace a failed replica node TopSen attempts to use nodes that are not connected to any other nodes (*singletons*). The selected nodes thus form a small disconnected component, under-utilized nodes are used, and new inter-connections to existing components are not created.

Second, if no singletons are available then an existing set of nodes is reused if they satisfy the replication criteria. Although, this imposes additional load on that set of nodes, it does not introduce any new inter-connections.

Third, TopSen splits apart components when replacing a failed replica node and during periodic topology maintenance. Components are sometimes joined to satisfy replication requirements. In many instances though, a single node x joins two or more mostly independent components. The topology can be improved by splitting these weakly connected components. Thus, if node x fails and there are singleton nodes available, then TopSen replaces x with two or more singletons; attaching each singleton to one independent component. In addition, the topology is periodically examined and weakly connected components are split. By combining these three methods the algorithm maintains a better topology than other simpler approaches as shown in Section 6.2.

5.2 Algorithm

This section describes the TopSen algorithm, which is presented as three separate algorithms. The first algorithm, presented in Section 5.2.3, is used to select the set of nodes to store a newly created file. The second algorithm, presented in Section 5.2.4, is used to select a replacement node for one that has failed. The third algorithm, presented in Section 5.2.5, is used to perform the periodic maintenance on the topology.

5.2.1 Perspective

The node selection algorithm is presented as an algorithm on a graph. The algorithm descriptions provide no explicit point of reference or context as to where the algorithm is run. In addition, the amount or quality of the topological information available to the algorithm is not specified. As described in Chapter 3, the policy creation portion is run on the client nodes when a replication policy is created. The second part of the algorithm that is responsible for selecting replacement

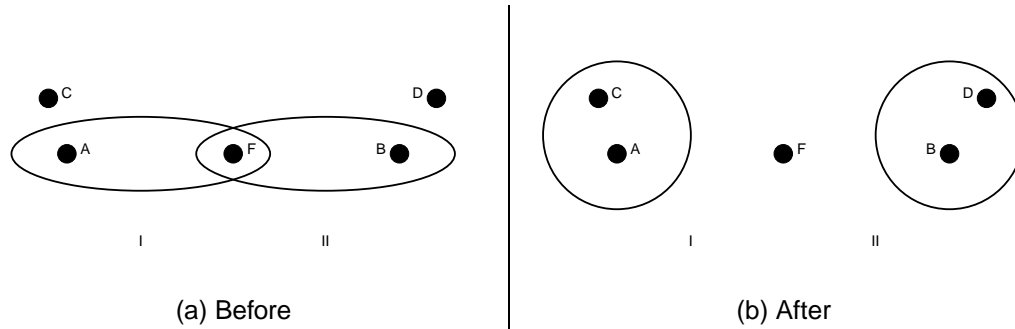


Figure 5.2: Before and after results of independent node selection.

nodes runs on both the client and the replica nodes. On client nodes the second part is executed when a replication policy is modified by a user. On replica nodes the second part is run by the leader of the replication policy when a failed node is detected and needs to be replaced.

Nodes do not possess a global view of the system topology. Each node's topology view depends on the replication policies it is participating in and the gossip information it has received so far. Node selection is performed independently, thus, given two nodes that are affected by the failure of the same node, their selection of a replacement node can be significantly different.

For example, Figure 5.2a shows two policies I and II that share a common node f . If f fails then the leader nodes a and b for I and II respectively must select a replacement node for f . Depending on the topology information they possess, they may not select the same node to replace f . In the scenario where each leader can replace the failed node with an idle replica node (see Figure 5.2b) then ideally the node selections for a and b should be different since it improves the quality of the topology by making it one step closer to separating a large component into two or more smaller components. In the scenario where no idle nodes are available the replacement node should ideally be the same for a and b so that components are not further intertwined by the new connections. This is further explained in Section 5.2.4

As described in Section 3.6.3 the approach taken to propagating updates ensures that all nodes know about all the other nodes in a component, in particular, they know the node-to-node connectivity and all the policies each node is responsible for. Possessing this knowledge enables a leader node not only to make intelligent decisions about the policies it controls, but to affect the topology of the component in beneficial ways. As mentioned, updates that are propagated by gossiping only contain static node attributes. Thus, before the node selection algorithm is run, the node obtains

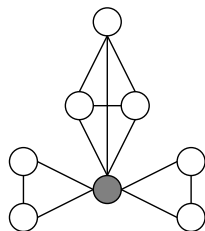


Figure 5.3: Cut vertex

the missing dynamic node attributes from all the nodes for which it does not have complete node attributes.

5.2.2 Definitions

We use the term *vertex* to denote a node. Edges (undirected) represent inter-connections between nodes. The term *topology* \mathcal{T} refers to the graph made by the vertices and the edges; $|\mathcal{T}|$ refers to the number of nodes in the system.

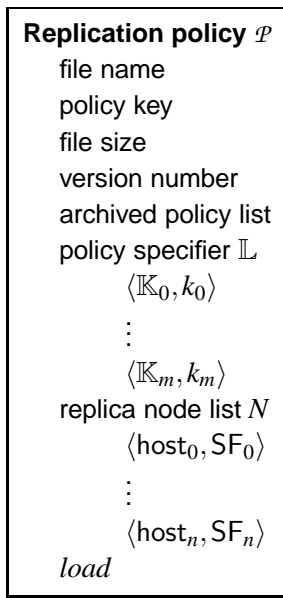
The term *singleton* is used to denote a node that is storing no data and is not connected to any other node. We use the term *cut vertex* (see Figure 5.3) to describe a node that if removed increases the number of connected components in the graph. A complete graph is created on the nodes in a replication policy. Every node, that is not a singleton, is part of at least one complete graph. Thus, a cut vertex must be part of two or more distinct policies.

Let \mathbb{K} be a *node selection constraint* and C a *node attributes*; see Section 4.2 for full definition.

A policy \mathcal{P} (see Figure 5.4) consists of a set N of nodes, a *policy specifier* \mathbb{L} , and a *load* property. The *policy specifier* \mathbb{L} specifies the composition of \mathcal{P} and is defined as a set of pairs, $\langle \mathbb{K}, k \rangle$, consisting of a node selection constraint \mathbb{K} and a non-negative integer k for each type of node required. The *load* specifies the number of files that use the policy. A set of nodes N satisfies policy specifier \mathbb{L} if there is a function that maps the nodes in N to the tuples in \mathbb{L} , such that for each $\langle \mathbb{K}, k \rangle \in \mathbb{L}$ at least k nodes satisfying \mathbb{K} are mapped to it. A policy specifier \mathbb{L} satisfies policy specifier \mathbb{L}' , denoted $\mathbb{L} \subseteq \mathbb{L}'$, if every set of nodes that satisfies \mathbb{L} also satisfies \mathbb{L}' . A policy \mathcal{P} satisfies policy \mathcal{P}' , denoted $\mathcal{P} \subseteq \mathcal{P}'$, if $\mathbb{L}_{\mathcal{P}} \subseteq \mathbb{L}_{\mathcal{P}'}$.

Two policies specifiers \mathbb{L} and \mathbb{L}' can be combined, denoted $\mathbb{L} + \mathbb{L}'$.

$$\mathbb{L} + \mathbb{L}' = \{ \langle \mathbb{K}, k + k' \rangle \mid (\langle \mathbb{K}, k \rangle \in \mathbb{L} \text{ or } k = 0) \text{ and } (\langle \mathbb{K}, k' \rangle \in \mathbb{L}' \text{ or } k' = 0) \}.$$

Figure 5.4: A replication policy \mathcal{P} .

Two policies \mathcal{P} and \mathcal{P}' can be combined, denoted $\mathcal{P} + \mathcal{P}'$, to create a new policy \mathcal{P}'' . $N_{\mathcal{P}''} = N_{\mathcal{P}} \cup N_{\mathcal{P}'}$ and $\mathbb{L}_{\mathcal{P}''} = \mathbb{L}_{\mathcal{P}} + \mathbb{L}_{\mathcal{P}'}$.

To compute the difference between two policy specifiers a distance function is defined between two policy specifiers \mathbb{L} and \mathbb{L}' is:

$$DL(\mathbb{L}, \mathbb{L}') = \sum_{\mathbb{K}} |k_{\mathbb{K}} - k'_{\mathbb{K}}|,$$

where

$$k_{\mathbb{K}} = \begin{cases} k & \text{if } \langle \mathbb{K}, k \rangle \in \mathbb{L} \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad k'_{\mathbb{K}} = \begin{cases} k & \text{if } \langle \mathbb{K}, k \rangle \in \mathbb{L}' \\ 0 & \text{otherwise} \end{cases}$$

5.2.3 Policy creation

The algorithm for creating policies follows the general recipe presented in Section 5.1 and consists of three phases. The input is a policy specifier \mathbb{L} , a set of existing policies $E_{\mathcal{P}}$, and the topology \mathcal{T} , the output is a set of nodes that satisfies \mathbb{L} . As mentioned earlier a node's knowledge about existing policies depends on the number of policies it is participating in and the information it has gathered through gossiped updates. Thus, existing policy knowledge, like the topology, is not uniform across all the nodes.

```

0 PROC_POLICYCREATE_I( S,  $\mathbb{L}$  )
1    $\mathbb{L}' = \mathbb{L}$ 
2   nodeset =  $\emptyset$ 
3   FOR EACH  $s \in S$ 
4     FOR EACH  $l \in \mathbb{L}$ 
5       IF (  $k_l > 0$  AND  $c_s \subseteq \mathbb{K}_l$  )
6          $k_l = k_l - 1$ 
7         ADD  $s$  TO nodeset
8         BREAK
9     IF (  $k_l == 0, \forall l \in \mathbb{L}$  )
10      RETURN  $\langle \text{TRUE}, \textit{nodeset} \rangle$ 
11 RETURN  $\langle \text{FALSE}, \textit{nodeset} \rangle$ 

```

Figure 5.5: Phase I of the policy creation algorithm.

Phase I – using singletons

In the first phase (see Figure 5.5) the algorithm attempts to satisfy the policy using only singleton nodes. The input is a set S of singleton nodes and the policy specifier \mathbb{L} . The output is a tuple consisting of a boolean and a set of nodes. If the set of nodes satisfies \mathbb{L} then TRUE is returned or FALSE otherwise. The variable *nodeset* stores the nodes selected for the policy.

The algorithm iterates through the set of singletons. For each node $s \in S$ the node's attributes are compared to the node selection constraint specified in \mathbb{L} (line 5). If the node attributes c_s of s satisfies one of $\mathbb{K}_l, l \in \mathbb{L}$, and the required number k_l of nodes of type \mathbb{K}_l is not yet reached then that node is added to the output set *nodeset* (line 7). If the required number of singletons are not found then the second phase of the algorithm (see Section 5.2.3) is executed.

The complexity of this phase is $O(|S| \cdot |\mathbb{L}|)$. In general, $|\mathbb{L}|$ tends to be small and constant, less than ten entries, thus the algorithm is mostly linear in the number of singletons.

This is a greedy solution and although it is linear the phase I algorithm may not return the optimal solution. For example, suppose there are two singleton nodes a and b and two constraints \mathbb{K}_0 and \mathbb{K}_1 . Let $c_a \subseteq \mathbb{K}_0$, $c_b \subseteq \mathbb{K}_0$, and $c_b \subseteq \mathbb{K}_1$. Then the optimal assignment is node a for constraint \mathbb{K}_0 and node b for constraint \mathbb{K}_1 . Depending on the order of $s \in S$ phase I may not arrive at the optimal assignment. One possible solution is to use *Flow Networks* which is guaranteed to

```

0 PROC_POLICYCREATE_II(  $E_{\mathcal{P}}$ ,  $\mathbb{L}$  )
1   FOR EACH  $\mathcal{P} \in E_{\mathcal{P}}$ 
2     IF (  $\mathbb{L}_{\mathcal{P}} \subseteq \mathbb{L}$  )
3        $O = \text{MIN\_SIZE}( Q, \mathcal{P} )$ 
4        $Q = \text{MIN\_LOAD}( O, \mathcal{P} )$ 
5    $load_Q = load_Q + 1$ 
6   RETURN  $Q$ 

```

Figure 5.6: Phase II of the policy creation algorithm.

arrive at the optimal solution but has a higher complexity in terms of runtime and implementation. This suboptimal selection does not affect the results presented in Section 6.2 because the above scenario of multiple node types satisfying the same constraint does not occur.

Phase II – reusing policies

If \mathbb{L} cannot be satisfied with singletons, the second phase of the algorithm (see Figure 5.6) is executed which attempts to satisfy \mathbb{L} with an existing policy \mathcal{P} . The input is a set $E_{\mathcal{P}}$ of existing policies and the policy specifier \mathbb{L} . The output is a reference to an existing policy \mathcal{P} .

The idea is to select \mathcal{P} such that \mathcal{P} satisfies \mathbb{L} and \mathcal{P} has the smallest number of nodes. The second constraint is used to balance the load in the system. Otherwise, a large policy could be used to satisfy both large and small policies causing it to be overloaded while under-utilizing the smaller policies.

Two functions are defined $\text{MIN_SIZE}(\mathcal{P}, Q)$ and $\text{MIN_LOAD}(\mathcal{P}, Q)$, both of which take two policies as input. The function $\text{MIN_SIZE}(\mathcal{P}, Q)$ returns the policy with the smallest number of nodes. The function $\text{MIN_LOAD}(\mathcal{P}, Q)$ returns the policy with the smallest load.

Each existing policy \mathcal{P} is compared to the policy specifier \mathbb{L} (line 2). If \mathcal{P} satisfies \mathbb{L} then it is compared to a previously selected policy Q and the policy with the smallest number of nodes is selected (line 3). If \mathcal{P} and Q have an equal number of nodes then the policy with the smallest load is selected (line 4). Once a policy is selected its *load* is incremented to reflect the increase in the amount of data the policy is responsible for. The complexity of this phase is $O(|E_{\mathcal{P}}|)$.

```

0 PROC_POLICYCREATE_III(  $S, E_{\mathcal{P}}, \mathbb{L}$  )
1    $policyset = \emptyset$ 
2    $nodeset = PROC\_POLICYCREATE\_I( S, \mathbb{L} )$ 
3   CONVERT_NODESET_TO_POLICY(  $nodeset, \mathcal{R}$  )
4    $nodeset = \emptyset$ 
5   FOR EACH  $\mathcal{P} \in E_{\mathcal{P}}$ 
6      $Q = \mathcal{P} + \mathcal{R}$ 
7     IF (  $\mathbb{L}_Q \subseteq \mathbb{L}$  )
8        $O = MIN\_SIZE( O, Q )$ 
9   IF (  $O$  )
10    EXTRACT SELECTED  $s \in N_{\mathcal{R}}$ , ADD TO  $nodeset$ 
11    RETURN  $\langle \mathcal{P}, nodeset \rangle$ 
12  FOR EACH  $\mathcal{P} \in E_{\mathcal{P}}$ 
13    IF (  $DL(\mathbb{L}, \mathbb{L}_{\mathcal{P}}) < DL(\mathbb{L}, \mathbb{L}_Q)$  )
14       $Q = \mathcal{P}$ 
15  ADD  $Q$  TO  $policyset$ 
16  LOOP
17    FOR EACH  $\mathcal{P} \in E_{\mathcal{P}} \setminus policyset$ 
18       $O = policyset + \mathcal{P}$ 
19       $Q = O + \mathcal{R}$ 
20      IF (  $\mathbb{L}_O \subseteq \mathbb{L}$  )
21         $u = MIN\_SIZE(u |, O)$ 
22      IF (  $\mathbb{L}_Q \subseteq \mathbb{L}$  )
23         $\psi = MIN\_SIZE(\psi, Q)$ 
24      IF (  $DL(\mathbb{L}, \mathbb{L}_O) < DL(\mathbb{L}, \mathbb{L}_{\psi})$  )
25         $\mathcal{W} = O$ 
26    IF (  $u$  )
27      RETURN  $\langle u, NULL \rangle$ 
28    IF (  $\psi$  )
29      EXTRACT SELECTED  $s \in N_{\mathcal{R}}$ , ADD TO  $nodeset$ 
30      RETURN  $\langle \psi, nodeset \rangle$ 
31    ADD  $\mathcal{W}$  TO  $policyset$ 
32    IF (  $E_{\mathcal{P}} \setminus policyset = \emptyset$  )
33      RETURN  $\langle NULL, NULL \rangle$ 

```

Figure 5.7: Phase III of the policy creation algorithm.

Phase III – combining policies

If no existing policy satisfies \mathbb{L} then the third phase (see Figure 5.7) is executed. To satisfy \mathbb{L} the algorithm first attempts to find the closest matching existing policy and extend it using singletons.

Second, it attempts to combine several existing policies, and third it attempts a combination of the two. The input is a set S of singletons, a set $E_{\mathcal{P}}$ of existing policies, and the policy specifier \mathbb{L} . The output is a set of existing policies and singletons that satisfy \mathbb{L} .

The algorithm presented below is greedy. It always selects the smallest existing policy to combine in the attempt to satisfy \mathbb{L} . One drawback of this approach is that it does not guarantee that the resulting component will have the smallest number of nodes. The optimal selection requires the examination of all combinations of all components. This is impractical because the algorithm becomes exponential in the number of existing policies and singletons.

First, phase I of the algorithm is used to build a set of nodes that can be used to extend an existing policy or a combination of existing policies (line 2). From this set of nodes a policy \mathcal{R} is created (line 3). The creation of \mathcal{R} provides for an efficient method to compute the contribution of the extension the singletons provide. Second, the algorithm iterates through all the existing policies to determine if there exists an existing policy \mathcal{P} such that if \mathcal{P} is extended by the singletons it is going to satisfy \mathbb{L} (lines 5-8). If such a policy \mathcal{P} exists (lines 9-11) then the singletons that form the extension to \mathcal{P} are extracted from the node set $N_{\mathcal{R}}$ in \mathcal{R} into the set *nodeset* (line 10); then the policy \mathcal{P} and the set *nodeset* of singletons are returned. Otherwise, a policy \mathcal{Q} is determined that partially satisfies \mathbb{L} the best (lines 12-15). That is, the difference between $\mathbb{L}_{\mathcal{Q}}$ and \mathbb{L} , is minimized. This is the starting point for the second part.

In the second part (lines 16-33) the algorithm loops through the existing set of policies looking for three types of combinations. Given the current set of policies, *policyset*, one, is there an existing policy \mathcal{P} such that *policyset* + \mathcal{P} satisfies \mathbb{L} (lines 20-21). Two, is there a set S of singletons such that *policyset* + \mathcal{P} + S satisfies \mathbb{L} (lines 22-23). Three is there $\mathcal{P}' = \text{policyset} + \mathcal{P}$ such that \mathbb{L} is not satisfied but $DL(\mathbb{L}_{\mathcal{P}'}, \mathbb{L})$ is minimized (lines 24-25). The selection of the policy is based upon the same two constraints described in the second phase of the algorithm. The contribution of the selected policy during each iteration should contribute the most to satisfying the policy specifier \mathbb{L} and have the fewest nodes necessary to do so.

If there exists a simple combination of policies that satisfies \mathbb{L} (lines 28-29) then that set *policyset* of policies is returned. Else if there exists a simple combination of policies plus singletons that satisfies \mathbb{L} (lines 28-30) then that set of policies and singletons are returned. Otherwise, the policy \mathcal{W} , that was chosen (lines 24-25), is added to the set *policyset* of policies and the process

```

0 PROC_REPLACE_NODE_I( n, S, T,  $\mathbb{K}$  )
1   match =  $\emptyset$ 
2   components = FindCutVertex( n, T )
3   IF ( |components| == 1 )
4     components = FindAuxiliaryCutVertex( n, T )
5   FOR EACH s  $\in$  S
6     IF (  $C_s \subseteq \mathbb{K}$  )
7       ADD s TO match
8       IF ( |components| == |match| )
9         BREAK
10  IF ( |match| == 0 )
11    RETURN failure
12  SORT_BY_SIZE( components )
13  FOR EACH c  $\in$  components
14    IF ( |match| > 0 )
15      match = match \ m, m  $\in$  match
16    ReplaceNode( c, n, m )
17  RETURN success

```

Figure 5.8: Phase I of the replica node replacement algorithm.

starts again. The loop iterates until there are no more existing policies to combine.

The main loop (line 16) iterates $|E_p|$ times. For each iteration of the outer loop the inner loop (line 17) iterates through $E_p \setminus \text{policyset}$ times, or $O(|E_p|)$. The overhead of determining if a policy satisfies a policy specifier is considerably greater than iterating through the singletons to create \mathcal{R} . Thus, the complexity of this phase is $O(|E_p|^2)$.

5.2.4 Node replacement

The algorithm for selecting a replacement for a failed node also consists of three phases and follows the general recipe presented in Section 5.1. The input to the three-phase replacement algorithm is the failed node n , the node selection constraint \mathbb{K} , a set of existing policies E_p , and the topology \mathcal{T} , the output is one or more nodes that replace the failed node n .

Phase I – singleton

In the first phase (see Figure 5.8) the algorithm attempts to replace the failed node n with one or more singleton nodes; more than one singleton is used if n is a cut vertex. The benefit of replacing n with multiple singleton nodes is that the topology is improved by splitting a large component into several smaller ones.

As stated previously, the goal of TopSen is to maintain a good topology by minimizing the number of inter-node connections so that a failure of a node does not affect a large number other of nodes. To this end, TopSen attempts to maintain a large number of small disconnected components. Although large components that have few edges do not explicitly create bad topologies, dealing with large components adds complexity to the algorithm. Our approach allows TopSen to operate at the granularity of a component verses edges and localize the effects on the topology.

As edges are added a component becomes a complete graph. If components are kept small then this is not a problem and the algorithm just has to avoid joining components. The size of a component is a natural repulsion force for adding edges because at some point no more edges can be added without combining components. This is not the case for large components. Imagine a topology that consists of a single loosely connected component. In this case the algorithm must operate at the granularity of edges and be able to calculate the affect of adding an edge on the topology.

The input to phase I is the failed node n , the connected component T that n was part of, a set S of singletons, and the node selection constraint \mathbb{K} . The output is success if the failed node n was replaced by one or more singletons. First the algorithm determines if n is a cut vertex (line 2), if it is not, the algorithm proceeds to find an auxiliary cut vertex n' ¹ (line 4); the result is one or more components stored in *components*. To determine if n is a cut vertex a breadth first *Minimum Spanning Tree* (MST) algorithm is used. If n is a cut vertex then multiple components, two or more, exist once n is removed. To determine if there exists an auxiliary cut vertex the algorithm iterates through the nodes in the component, removing each one and running the breadth first MST algorithm. If multiple components are created then there exists an auxiliary cut vertex. Next, the algorithm iterates through the available singletons and selects a number of nodes, up to the number

¹An auxiliary cut vertex n' is a vertex such that n and n' are part of the same component, and if n and n' fail then the component would become disconnected. This is done in anticipation of node n' failing, so that when n' does fail the component can be split into several smaller components.

```

0 PROC(  $T, E_p, \mathbb{K}$  )
1   FOR EACH  $\mathcal{P} \in E_p$ 
2      $U = \bigcup N_{\mathcal{P}}$ 
3    $A = T - U$ 
4   FOR EACH  $a \in A$ 
5     IF (  $C_a \subseteq \mathbb{K}$  )
6        $node = \text{MIN\_LOAD}( node, a )$ 
7   RETURN  $node$ 

```

Figure 5.9: Phase II of the replica node replacement algorithm.

of components, that satisfy the node selection constraint \mathbb{K} (lines 5-7). If no singletons are available the algorithm proceeds to the second phase.

The components are then sorted from largest to smallest in order to split off the largest components first; thus maximally reducing the overall size of the components. This is only important when the number of available singletons is less than the number of components. For each component, n is replaced by one of the selected singleton nodes (lines 13-16). When there is only one selected singleton left, it is used to replace n in the remaining components.

Determining the cut vertex and the auxiliary cut vertex dominates the runtime of this phase. To determine the cut vertex requires $O(|T|)$. To determine the auxiliary cut vertex, it is necessary to iterate through all the nodes in T , removing each one and running the cut vertex algorithm, thus the complexity is $O(|T|^2)$.

Phase II – internal component

When no singletons are available to replace the failed node n an attempt is made to find a replacement node from amongst the nodes in the component T . To increase utilization and to load balance the singletons are used first and this step is done second (see Figure 5.9). Although components would grow smaller if done in the first phase, the selected node would be responsible for storing more data, since it is already some storing data, and the singletons, that could replace n and store no data, would be under utilized.

The input is the failed node n , the connected component T that n was part of, the set E_p of

```

0 PROC(  $G, \mathbb{K}$  )
1   FOR EACH  $g \in G$ 
2     FOR EACH  $r \in g$ 
3       IF (  $C_r \subseteq \mathbb{K}$  )
4          $node = \text{MIN\_COMPONENT}(node, r)$ 
5   RETURN  $node$ 

```

Figure 5.10: Phase III of the replica node replacement algorithm.

policies that node n was responsible for, the node selection constraint \mathbb{K} . Let $N_{\mathcal{P}}$ be the set of nodes listed in a policy $\mathcal{P} \in E_{\mathcal{P}}$. The output is the replacement node.

First the nodes that cannot be selected as a replacement for n are determined. These nodes are easy to categorize as those that are adjacent to n and are determined by taking the union of all the nodes listed in all of the policies that the node n was responsible for (lines 1-2) and subtracting it from T (line 3). Next the algorithm iterates through A and selects nodes that satisfy the node selection constraint \mathbb{K} (lines 4-6). If there are multiple possible selections then the node with the smallest *load* is selected in order to distribute the load. At worst, this phase has to iterate through all the nodes in T , thus the complexity of this phase is $O(|T|)$.

Phase III – combining components

The third phase (see Figure 5.10) is executed only when the first two phases have failed to select a replacement node for n . In this final phase the algorithm joins two components to find a replacement node for n . The input to the algorithm is the failed node n , a set G of all the components in the system, and the node selection constraint \mathbb{K} . The output is a replacement node for n . This phase is performed as a last resort because it worsens the topology by joining components, making them bigger, and reducing their number; contrary to our initial goals. A function `MIN_COMPONENT` is introduced that given two nodes returns the node that is part of the smaller component.

In this final phase the algorithm iterates through all the nodes r that are listed in the components $g \in G$ to locate a node that can replace n . To avoid growing the component any larger than necessary, the node that replaces n is chosen such that it is part of the smallest possible component (line 4).

```

0 PROC(  $S, G$  )
1   FOR EACH  $g \in G$ 
2     FOR EACH  $r \in g$ 
3        $components = \text{FindCutVertex}( r, g )$ 
4       IF (  $|components| > 1$  )
5          $match = \text{NULL}$ 
6         FOR EACH  $s \in S$ 
7           IF (  $c_s \subseteq \mathbb{K}_r$  )
8             ADD  $s$  TO  $match$ 
9         IF (  $|match| > 0$  )
10          SORT(  $components$  )
11          FOR EACH  $c \in components$ 
12            IF (  $|match| > 0$  )
13               $match = match \setminus m, m \in match$ 
14              ReplaceNode(  $c, r, m$  )
15          RETURN success
16 RETURN success

```

Figure 5.11: Periodic maintenance algorithm.

The complexity of this phase is $O(|\mathcal{T}|)$ since it iterates through all of the nodes in the system.

5.2.5 Periodic maintenance

In addition to selecting replica nodes for policies and finding replacement nodes, an algorithm (see Figure 5.11) is provided to perform periodic maintenance to improve the topology. This algorithm is similar in nature to the first phase of the algorithm presented in Section 5.2.4 which selects replacement nodes. The algorithm finds cut vertices and splits components into smaller components. The input to the algorithm is a set S of singletons and a set G of components.

The algorithm iterates through the nodes r in all of the components $g \in G$ to find a cut vertex v . Once a cut vertex is found, the algorithm iterates through the singletons to see if there is at least one singleton s that satisfies the node selection constraint of v . One singleton is sufficient since at least one component changes from using v to s and the others can continue to use v . Once the singletons are selected the process of component splitting is performed, similar as in the first phase of the replica node selection algorithm (see Section 5.2.4). The complexity for this part is $O(|\mathcal{T}| \cdot |r|)$.

Chapter 6

Evaluation

The wonderful thing about standards is that there are so many to choose from.

— unknown

The evaluation of PDR attempts to demonstrate that the overhead of the system is reasonable, that the system is scalable, and that there is a benefit to using PDR. To show that the overhead added by PDR is reasonable, micro-benchmarks of the common file system calls are used (see Section 6.1). In addition, two common tasks are performed, untarring and compiling, to demonstrate that normal file system usage is not hindered by PDR.

First, system scalability is demonstrated by evaluating the node selection and topology maintenance algorithms in Section 6.2. Next, the cost of replication and failure recovery is evaluated in terms of the communication costs, i.e., the number of messages, in Section 6.3. Micro-benchmarks are not used to evaluate these costs for several reasons. First, both replication and failure recovery are handled off the critical path and thus they do not directly add any latency to file system operations. Second, the incurred overhead depends on a number of factors, such as the network bandwidth, the workload of the replica nodes, and any other factors that are dependent on the operating environment rather than the implementation. Thus, an evaluation metric is needed that is invariant to these factors and just focuses on the implementation. The communication costs, the number of messages, is just such a metric. It is invariant to factors such as the network bandwidth and the workload of the replica nodes, and focuses on the algorithms used to perform replication and failure recovery. This metric is only dependent on the number of nodes that are participating in the operation. Thus, not only can the overhead be evaluated, but also the scalability of these

Operation	PDR (μ s)	EXT3 (μ s)
statfs	283.0	243.6
creat	602.5	186.9
open (truncate)	200.6	36.0
open (append)	119.8	5.2
open (read only)	118.9	4.8
close	90.8	1.8
mkdir	511.0	215.5
rmdir	528.8	50.4
remove	690.7	55.5
rename	1601.5	304.5

Table 6.1: Micro-benchmarks for statfs, creat, open, close, mkdir, rmdir, remove, and rename file system calls.

operations with respect to system size.

Finally, in Section 6.4 the applicability of PDR is discussed. A qualitative analysis is performed on file system traces and on several usage scenarios to determine the savings that PDR can provide compared to traditional backup approaches.

6.1 Micro-benchmarks

A combination of micro-benchmarks and real-world tasks are used to demonstrate that the overhead of PDR is reasonable. The micro-benchmarks are acquired on a Pentium III PC running at 650MHz with 512MB of memory. The machine was running Linux with kernel version 2.6.9. All numbers reported as the median of 1000 trials on an otherwise unloaded machine.

PDR is compared to the standard Linux in-kernel EXT3 file system. Table 6.1 presents measurements for the basic file system operations. The statfs operation is the simplest operation in PDR which just calls statfs and returns the information. This operation is used to determine the overhead of using an in-kernel redirector and performing an upcall for most file system operations; there is about a 40μ s cost for this context switch. All operations except statfs and close implicitly perform a lookup operation. The lookup operation consists of an upcall that performs a stat on the file or directory being looked up. Thus for most operations there is an additional cost of an upcall plus the cost of a stat, 3.2μ s, for a total of about 43μ s.

Most file system calls require that metadata be modified in PDR. For operations such as `creat`, `mkdir`, and `rename` there is an additional $33\mu\text{s}$ overhead for creating a directory entry. Directory entries are stored in flat files containing *dirent* structures. The Coda [36, 77] redirector accesses these entries by requesting an open file descriptor to this file. For operations such as `remove`, `rmdir`, and `rename` there is an additional $280.5\mu\text{s}$ overhead for deleting a directory entry.

File system calls such as `creat`, `mkdir`, `remove`, `rmdir`, and `rename` add and delete entries in the policy oracle database. An add or delete operation takes about $22725\mu\text{s}$; the database is configured to be transactional for consistency purposes, which requires the transaction log to be continuously flushed to disk. The majority of the work is considered to be house-cleaning, and is performed by a different thread and after PDR has replied to the redirector and thus does not affect the latency of operations (see Table 6.1).

To determine the impact of PDR on normal file system usage two additional measurements are done. First, the source for the Linux kernel which is 227MB of source in 17500 files is untarred. The newly created files were scheduled to be replicated in the future, no replication occurred during the untarring process. Second, the PDR system which consists of 530KB of source in 70 files is compiled. Both measurements were done with a cold cache. The tar file of the Linux kernel source is compressed using the *bzip2* compression algorithm and utility. Untarring the Linux Kernel while simultaneously decompressing the archive¹ took 76s. If the archive is decompressed before hand then just the untarring process took 17s. On PDR decompressing and untarring took 104s and just untarring took 41s. Tar is a file system intensive task; especially when creating a large number of small files. The factor of 2.41 slow down is attributed to `creat` being about three times slower, `close` being slower because it creates and inserts a replicate message for the replicator, and the smaller file system throughput. The EXT3 file system, mounted with standard options, does not use synchronous writes, thus the cost of performing the physical write is not felt during the untarring process. Notice that decompression adds a sufficient amount of overhead to mask the majority of the overhead introduced by PDR; a slow down of 1.36 is seen when untarring and decompressing is done simultaneously. Compiling PDR took 40.6s on PDR and 40s on the local file system. Compiling is not a file system intensive task, and thus PDR is on par with the local file system.

¹`bzcat linux-2.6.9.tar.bz | tar -xf -`

6.1.1 Redirector overhead

The overhead introduced by the redirector can be mitigated by using a redirector that performs more of the work in the kernel. Currently, the redirector acts like a detour and PDR performs the bulk of the work for file system calls, and returns the result of the operation to the redirector. If file system operations were entirely performed in the kernel and PDR was simply informed of their occurrence, then a significant amount of the overhead would be eliminated; that is, the redirector should behave like a tee.

Using the Coda redirector is sufficient for the PDR prototype but it is not the ideal solution. The benefit of using the Coda redirector is that it is already written, debugged, and tested. The disadvantage is that it is considerably heavier than needed by PDR. An alternate but similar approach is to use a loop-back NFS server as many other research systems do. In this scenario the client is the standard NFS client that communicates to a userlevel NFS server whose backend is PDR. The problem with this approach is that NFS is stateless and thus has no explicit notion of `open` or `close`; PDR needs to be informed of `close` calls in order to operate efficiently and correctly.

Coda is too heavy because most file system calls, all except `read` and `write`, are intercepted and must be handled at userlevel. Although this is not detrimental it does add additional overhead and latency to all file system operations. Ideally, the redirector should only intercept four file system calls. The `close` call should only be intercepted when a file is created or modified; that is, the `close` call should not be intercepted for a file that is opened for reading. The `remove` and the `rmdir` calls should be intercepted when a file or a directory is deleted. The `rename` call should be intercepted when a file or a directory is renamed.

6.2 Replica node selection

Simulation is used to evaluate the policy creation and replica node selection algorithms presented in Chapter 5. The objective is to evaluate the effectiveness of the algorithm in maintaining the topology as the system evolves over time; i.e., nodes fail, new nodes arrive, and new replication policies are created. The evaluation shows that the algorithm performs better than simple naive approaches, that it is scalable, that the algorithm is insensitive to the rate of policy creation, and that it is insensitive to the size of the system.

6.2.1 Simulation

A simple discrete event simulator is used to perform the simulations. There are four events that are generated in the simulation: node failure, new node insertion, policy creation, and periodic maintenance. There are several different classes of nodes. The simulation assumes that nodes of the same class are independent, have similar failure modes, and that their lifetimes are exponentially distributed; different classes have different failure modes and different expected lifetimes: for certain peer-to-peer systems this assumption has been shown to be valid [9]. Although not perfect, the assumption of exponentially distributed lifetimes represents a valid approximation, especially given that the subject of interest is the evolution of the system rather than specific events.

The probability of a node failure during a period of length τ is $1 - e^{-\mu\tau}$, where $1/\mu$ is the expected lifetime of a node. In the simulation there are two classes of nodes having an expected lifetime of 100 and 300 days; higher expected lifetimes translate to more reliable nodes. A study done by Bolosky et al. [11] showed that in a large organization, such as Microsoft Research, the average uptime of a workstation is approximately 300 days. Periods of 100 and 300 days are chosen because a high frequency of events provides for a better evaluation of the evolution of the system, which is the predominant point of interest. Although the lifetimes may seem short, they roughly correspond to the upgrade pattern in the department of Computer Science at the University of British Columbia where a machine is upgraded a couple times a year and it is wiped before doing the upgrade.

A similar distribution is used for policy creation with $1/\mu$ being varied between one and 120 days. The number of policies created at each policy creation event varied uniformly between one and five. A file system trace is not used to simulate policy creation because the pattern of policy creation does not affect the system and is not a point of interest. Thus, it is sufficient to use a synthetic trace to evaluate the evolution of the system and how the rate of policy creation affects the system.

Each policy specifies that data be stored on three to seven low reliability (100 days) nodes and one to four high reliability (300 days) nodes. Although Starfish [20] showed that three replica nodes are sufficient, it also assumes that at least two of the three replica nodes are tightly coupled to the primary. We do not make this assumption and thus we increase the number of replica nodes to achieve the same or a greater level of protection.

After each event the state of the topology is recorded. Specifically, the total number of nodes, the number of components, and the number of singletons in the system are computed. In addition, the minimum, the average, and the maximum node degrees are computed; the same statistics are computed for the number of cliques a node is a member of. This information is then used to compute the quality of the topology.

6.2.2 Metrics

The evaluation of the system topology is based on three metrics: the average node degree, the average number of cliques a node is a member of, and the number of components in the system. The node degree is equivalent to the vertex degree. The degree d of a node a is the number of nodes it is linked to, and represents the overhead, in terms of the number of nodes affected if a fails and the number of messages, $O(d)$, necessary to perform the recovery. The smaller the degree, the fewer nodes affected by the failure of a . This metric measures the effectiveness of both the policy creation and the replica node selection algorithms.

The number of cliques a nodes is a member of is representative of its virtual load. A node that is a member of a large number of cliques is responsible for a larger amount of data and must interact with a potentially larger number of nodes to keep the data safe. Thus, the goal is to maintain the average number of cliques a node is a member of small, and ensure that the policies are distributed across all the nodes. A node that is responsible for many reused policies must still manage more data, but does not incur more overhead with respect to policy management, since two identical policies are managed as one policy that simply encompasses more data. This metric predominantly measures the effectiveness of the policy creation algorithm since the majority of the load on a node is instantiated when a policy is created.

The last metric is the number of components in the system (a component is a disconnected subgraph of the graph created by the system topology). This metric is an overall measure of how well the policy creation and the replica node selection algorithms are performing together. When the number of components is large then the individual components are relatively small, the node degrees are small, and a failure does not affect a large number of nodes. In addition, the majority of nodes are participating in replicating data. A smaller number of components means that either only a few nodes are participating in policies, and thus some nodes are heavily loaded while others are

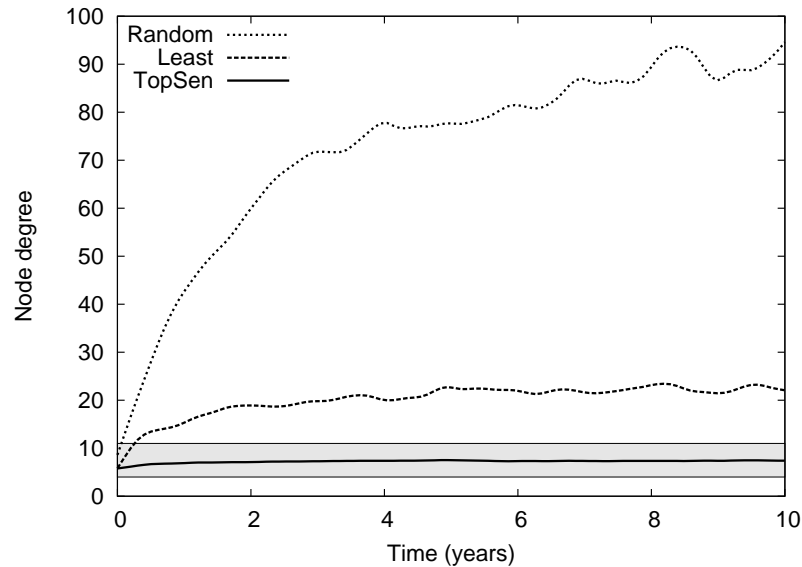


Figure 6.1: Comparison of TopSen, Random, and Least: Node degree verses Time in years.

idle, or the components are large, the number of inter-connections between nodes is large, and thus a node failure may affect a large number of nodes.

In Section 5.2 two algorithms were presented, one for policy creation and one for replica node selection. Unless explicitly mentioned otherwise, they are evaluated as a single entity.

6.2.3 The three algorithms

As part of the evaluation of the TopSen algorithm it was compared to two other algorithms, *Random* and *Least*. The *Random* algorithm selects a random node of the same type when replacing a failed node and selects the nodes at random when creating a policy. The *Least* algorithm selects a node of the same type that has the smallest node degree to replace a failed node, and selects the nodes with the smallest degrees when creating a new policy, and if there is a tie, the node with the smallest number of policies is selected. The simulation consists of 1000 nodes, over a ten year period, and initially stores 100 policies. The initial number of policies may seem low but this is necessary to demonstrate the decline of the number of components in Figure 6.3. If the initial number of policies is increased then the initial starting point for the *Random* and *Least* algorithms becomes a single component, and thus a decrease in the number of components cannot be observed.

Figure 6.1 presents the average node degree for the three algorithms. The light-gray region denotes the optimal range, defined by $[|\mathcal{P}|_{min}, |\mathcal{P}|_{max}]$, for the average node degree which occurs when the number of nodes in a component equals to the number of nodes listed in a policy. Of the three algorithms, the TopSen algorithm maintains the lowest, the most consistent, and uniform node degree throughout the simulation. In both the Random and the Least algorithms the node degree increases over time. After a few years the average node degree is about ten times higher for the Random algorithm and three to four times higher for the Least algorithm. That is, compared to the TopSen algorithm, ten times as many nodes are affected by a node failure with Random selection, and three to four times as many nodes are affected by a node failure with Least selection. Both Random and Least algorithms create topologies with higher average node degrees because they do not attempt to reduce the number of inter-connections made when replacing failed nodes. The reason the Random algorithm has the highest node degree is that when a node is added to a policy, its node degree increases by the size of the policy. Thus, selecting a node at random, and adding it to a policy, increases the node's degree (on average) by the average size of a policy. And this increases the average node degree much more than selecting a low degree node (Least) or a node with the majority of the inter-connections already in place (TopSen) and inserting it into an average sized policy.

Figure 6.2 shows the average number of cliques a node is a member of for the three algorithms. The TopSen algorithm performs best since it tries to overlap and reuse existing policies as much as possible. Thus, the number of completely new policies, with new inter-connections, created is significantly less than that of the Random and the Least algorithms. The Random and the Least algorithms create unique new policies significantly more often because they do not take into consideration the underlying topology. In addition, the TopSen algorithm has the ability to split components thus further reducing the node degree and the node membership in cliques.

Figure 6.3 presents the number of components, on a log-scale, in the system for the three algorithms. The light-gray regions denote the optimal range, defined by $\left[\frac{|\mathcal{I}|}{|\mathcal{P}|_{min}}, \frac{|\mathcal{I}|}{|\mathcal{P}|_{max}}\right]$, for the number of components which occurs when the number of nodes in a component equals to the number of nodes listed in a policy. The TopSen algorithm does significantly better than the Random or the Least algorithm. Whereas with the Random and the Least algorithms the number of components quickly drops down to one or two, the TopSen algorithm maintains the number of components at

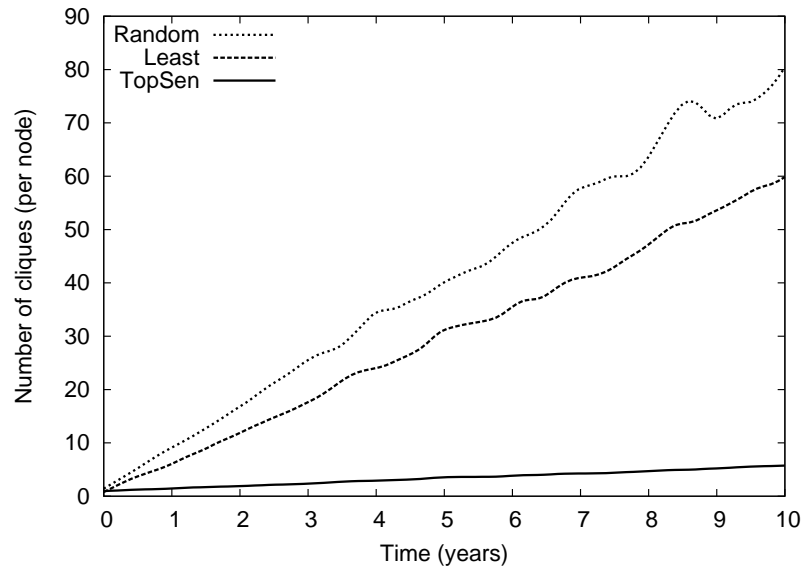


Figure 6.2: Comparison of TopSen, Random, and Least: Node membership in cliques versus Time in years.

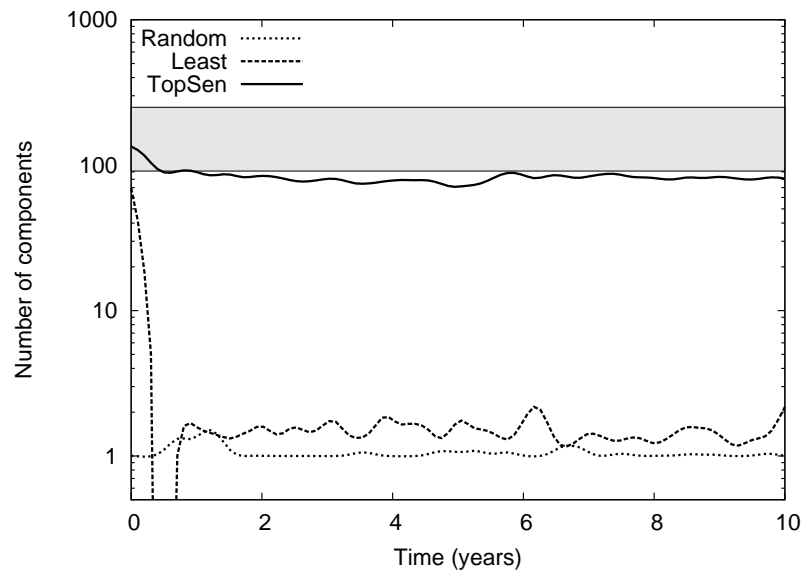


Figure 6.3: Comparison of TopSen, Random, and Least: Number of components versus Time in years.

a relatively constant level; the discontinuity for the Least algorithm is due to the steepness of the curve and the data smoothing function, there is always at least one component. This result is in-

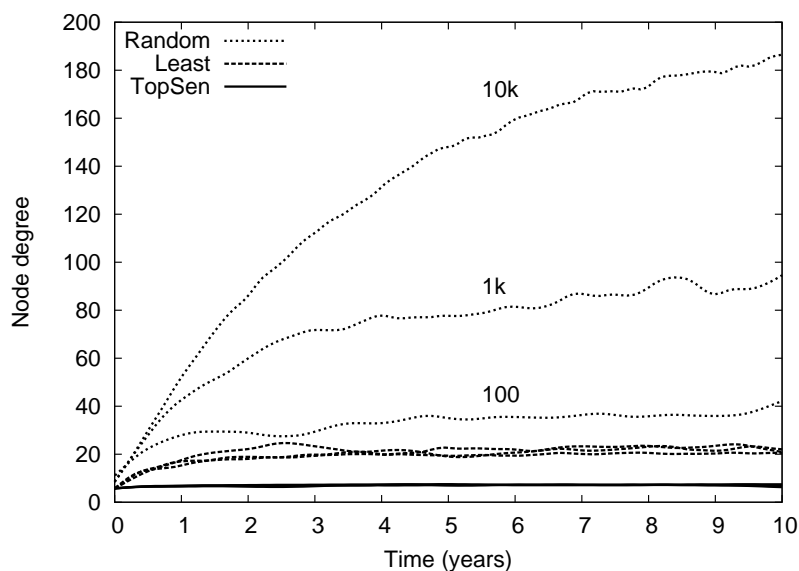


Figure 6.4: Scalability with constant load: Node degree versus Time in years for 100, 1k, and 10k node simulations.

line with the other two metrics and confirms that the TopSen algorithm is maintaining the topology better than the Random or the Least approach, with the result of having many small disconnected components.

6.2.4 Scalability

The scalability of TopSen is compared to the Random and the Least algorithm by performing simulation runs of 100, 1k, and 10k nodes, for a ten year period, with an initial policy load of 10, 100, and 1k policies respectively. Two sets of simulation runs were performed. In the first set the system load, the policy creation frequency, was kept constant for all three system sizes. In the second set the system load was scaled proportionally to the system size. That is, the policy creation frequency for the 10k node system is ten times higher than for the 1k node system. Please note that there is no data for the Least and Random algorithms for the 10k node, proportional load, simulation; this is due to the unreasonably long (~ 30 day) runtime.

Figure 6.4 demonstrates that the TopSen algorithm scales considerably better than the Random algorithm and noticeably better than the Least algorithm with respect to the node degree. For all 100, 1k, and 10k node simulation runs the node degree is low, about six, and constant. The TopSen

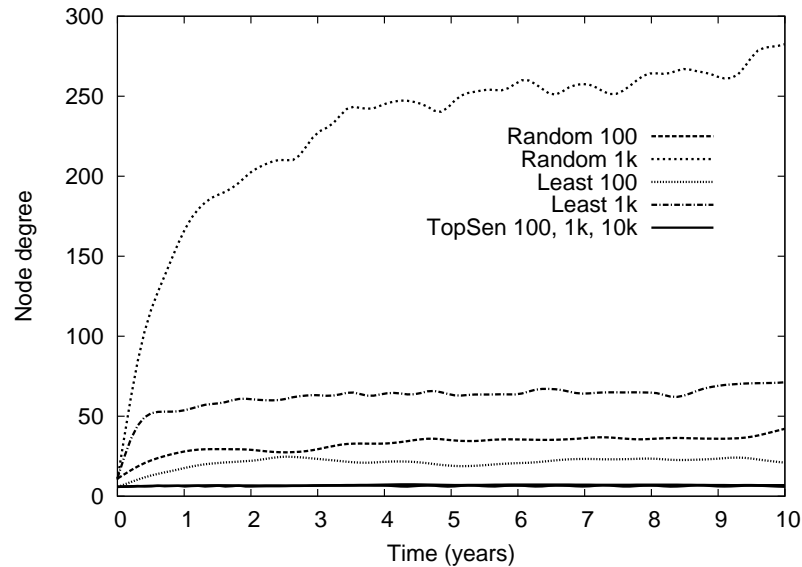


Figure 6.5: Scalability with proportional load: Node degree versus Time in years for 100, 1k, and 10k node simulations.

algorithm is not affected by the number of nodes in the system since its focus is strictly at the topology and component level. No matter how many nodes are available it always selects a node that induces the fewest number of new inter-connections in the system. The Least algorithm is the next best performer. It maintains the node degree relatively constant but it is about three to five times higher than that of the TopSen algorithm. Since the Least algorithm uniformly spreads the policy load by always selecting a node with the smallest node degree, the performance of the algorithm slightly improves as system size increases because there is a larger selection of replica nodes. The Random algorithm performs the worst since it pays no attention to the node degree or the topology.

Scaling the system load, policy creation frequency, proportionally to the system size further demonstrates the scalability of TopSen (see Figure 6.5). TopSen is completely unaffected while the node degree for both the Random and the Least algorithms increases as the system size and load increases. On average, if the system grows by a factor of x then the node degree increases by a factor of $\frac{1}{2}x$.

For the Random and the Least algorithms the node degree and the number of cliques a node is a member of is directly affected by the number of nodes in the system and the rate of policy creation (see Figure 6.6 and Figure 6.7). If the number of nodes in the system is small and the policy creation

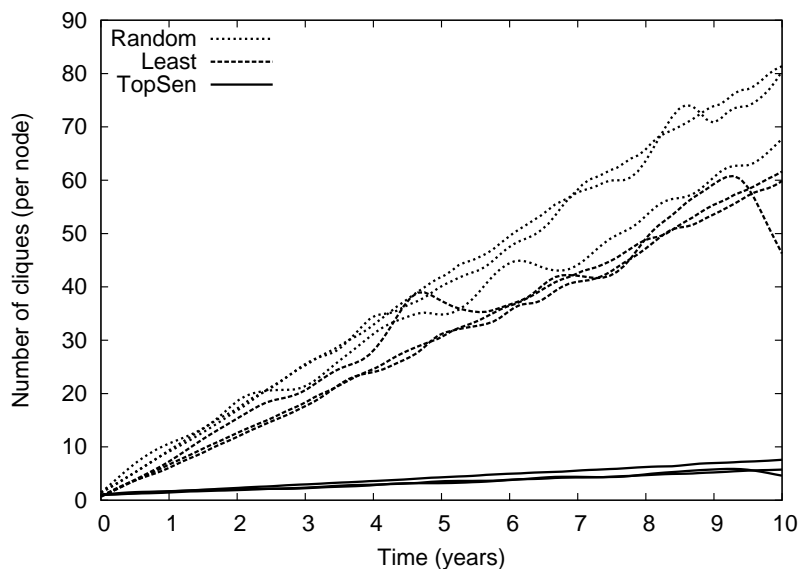


Figure 6.6: Scalability with constant load: Node membership in cliques verses Time in years for 100, 1k, and 10k node simulations.

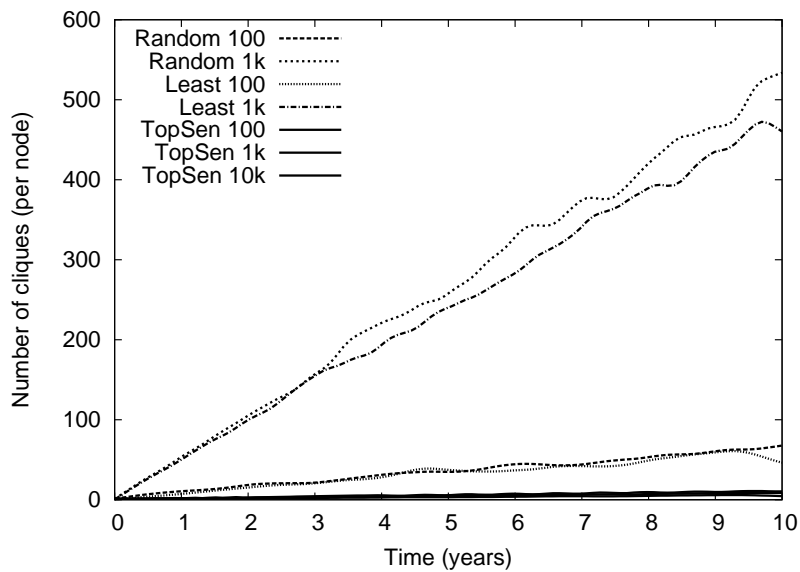


Figure 6.7: Scalability with proportional load: Node membership in cliques verses Time in years for 100, 1k, and 10k node simulations.

rate is high then both the node degree and the number of cliques a node is a member of is also large. This is due to the Random and the Least algorithms being insensitive to the underlying topology.

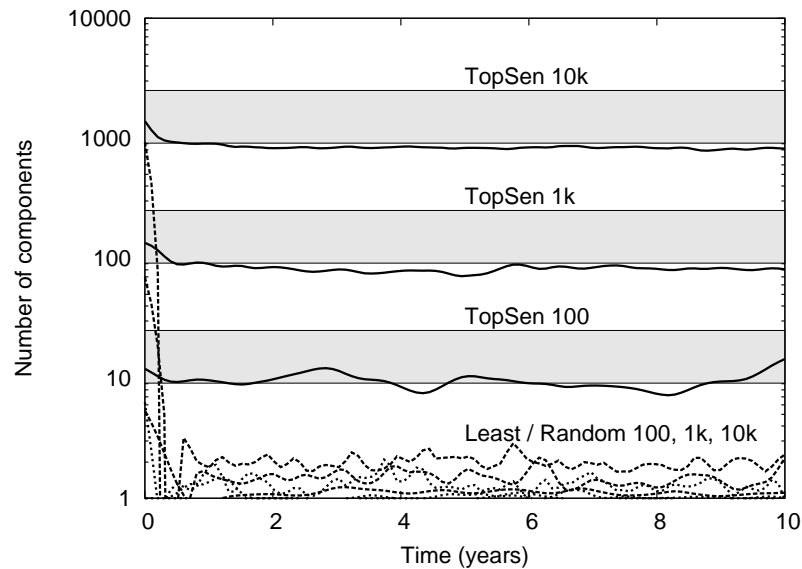


Figure 6.8: Scalability with constant load: Number of components versus Time in years for 100, 1k, and 10k node simulations.

In Section 6.2.5 the affect of policy creation rate on the system topology is discussed. The TopSen algorithm is sensitive to the system topology and attempts to overlay and reuse existing policy definitions as much as possible. Thus, TopSen is practically insensitive to the number of nodes in the system and the policy creation rate.

The scalability advantages of TopSen are most clearly seen when looking at the number of components in the system. TopSen maintains the number of components relatively constant and higher than the Random and the Least algorithms (see Figure 6.8). For the TopSen algorithm the average number of components is about an order of magnitude less than the number of nodes in the system. In the Random and the Least algorithms the number of components drops to just a couple, irrespective of the number of nodes in the system. Again, the TopSen algorithm's superiority comes from being sensitive to the topology and avoiding the creation of new inter-connections between nodes.

6.2.5 Policy creation

Next, how the rate of policy creation affects the policy creation and replica node selection algorithms is examined. The rate of policy creation is varied from 120 days to seven days between policy

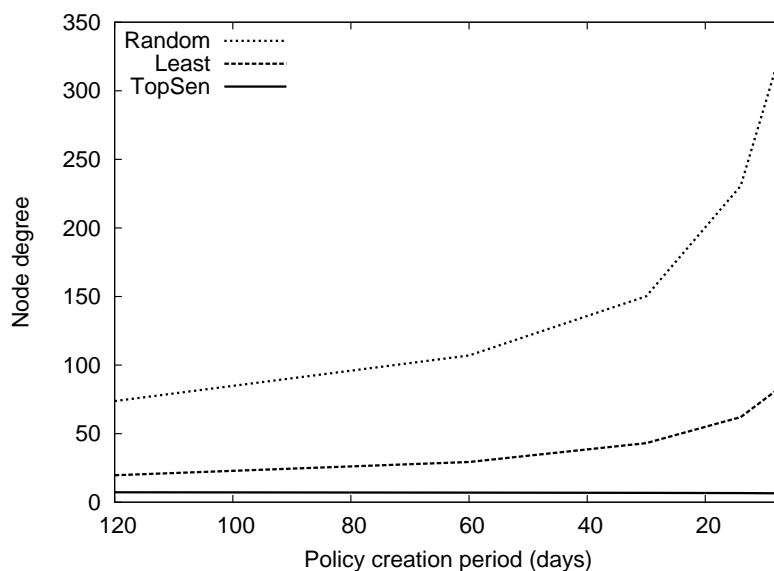


Figure 6.9: Policy creation frequency: Node degree versus policy creation period in days.

creation events. For each frequency a simulation run is performed with 1000 nodes for a period of ten years.

Policy creation frequency does not have a significant affect on the topology when the TopSen algorithm is used for replica node selection compared to the Random or the Least algorithms. Under the TopSen algorithm the average node degree varied slightly (see Figure 6.9). Under Random and less so under Least as the frequency of policy creation increases the node degree starts increasing almost exponentially. With Random and Least there is a high probability that every new policy creates additional inter-connections. Thus as the rate increases so does the node degree. TopSen strives to reuse connections, and succeeds, thus the node degree stays relatively level.

As with the node degree, the average number of cliques a node is a member of is maintained relatively constant with TopSen. Under Random and Least the average number of cliques a node is a member of grows almost exponentially with the increase of the policy creation frequency (see Figure 6.10). Under TopSen the rate at which the clique membership increases on a node approximately doubled, from 0.74 to 1.21 per year, the increase is still small given that the frequency of policy creation increased by two orders of magnitude.

The average number of components in the system were unaffected by the policy creation frequency under TopSen (See Figure 6.11). This is due to TopSen reusing existing policies and mini-

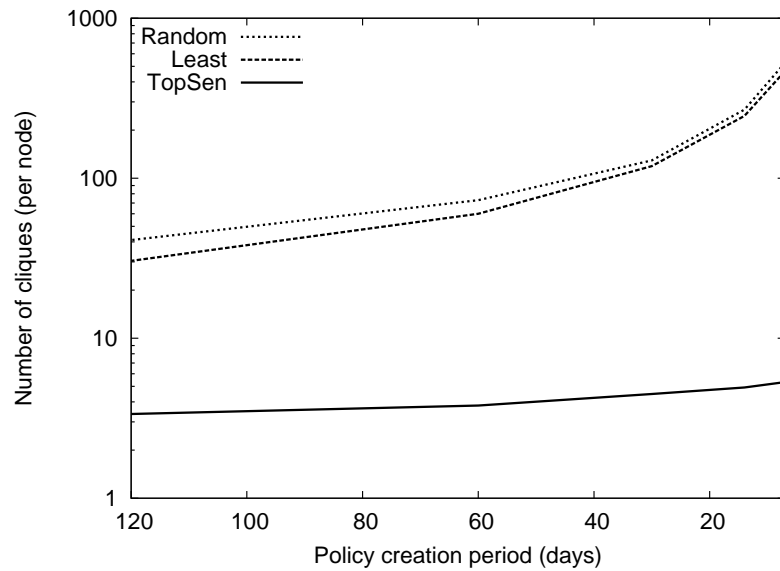


Figure 6.10: Policy creation frequency: Node membership in cliques versus policy creation period in days.

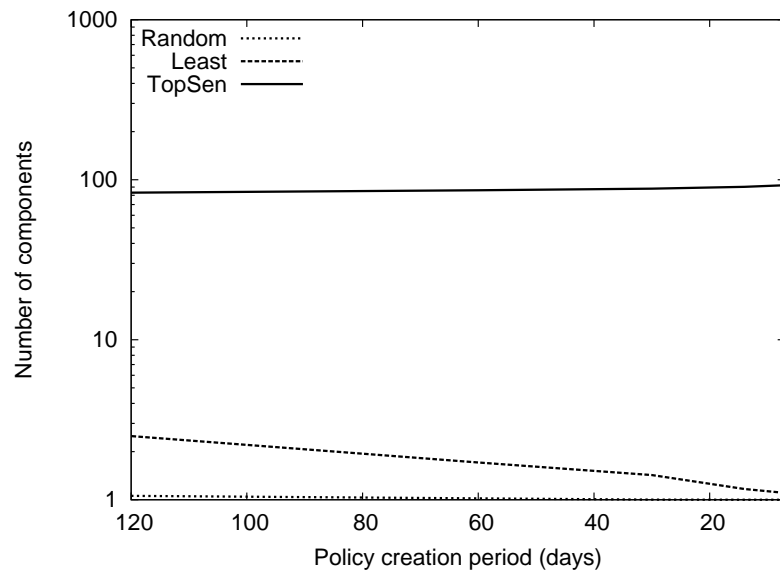


Figure 6.11: Policy creation frequency: Number of components versus policy creation period in days.

mizing creating new inter-connections. The Least algorithm is considerably more affected in terms of average number of components as evident from the graph. The average number of components

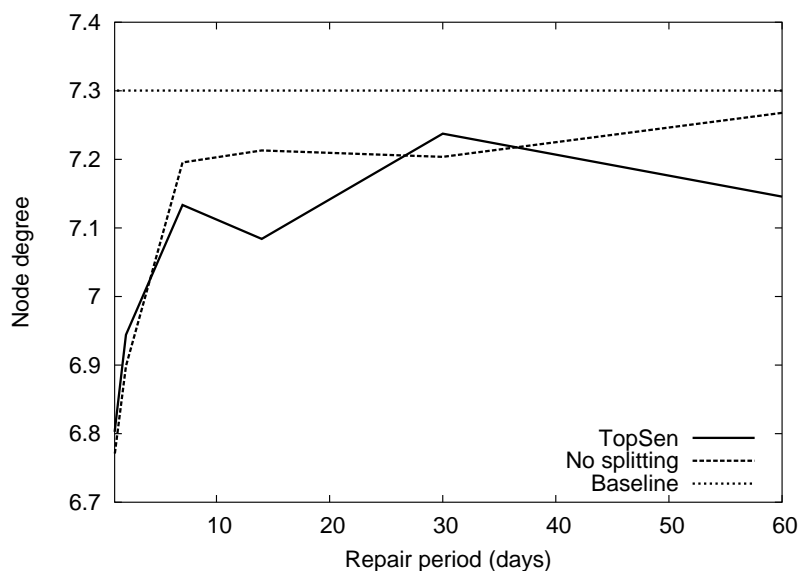


Figure 6.12: Benefits of splitting: Node degree versus repair period in days.

slowly declines as the policy creation frequency increases. Random seems to be unaffected but that is because the average number of components quickly converges to one and stays there.

6.2.6 Splitting and Periodic maintenance

This section evaluates the benefits of splitting components. Splitting is attempted when a node has failed and is a cut vertex and during periodic maintenance when a cut vertex is encountered. To aid the evaluation and comparison a baseline is provided for all of the results. The baseline is the TopSen algorithm that performs no component splitting when a node fails and performs no periodic maintenance. For each metric two versions of the TopSen algorithm are run. The first is the regular TopSen algorithm and the second performs no component splitting when a node fails. The frequency of the periodic maintenance is varied between daily and 60 days. Each simulation run was for 1000 nodes over a ten year period with an initial policy load of 100 policies.

The average node degree (see Figure 6.12) is not significantly affected by either type of splitting unless the periodic maintenance is performed on a daily basis. Performing periodic maintenance on a daily basis is not practical because a significant amount of bandwidth is necessary to re-replicate a node. Performing component splitting when a node fails and performing periodic maintenance on

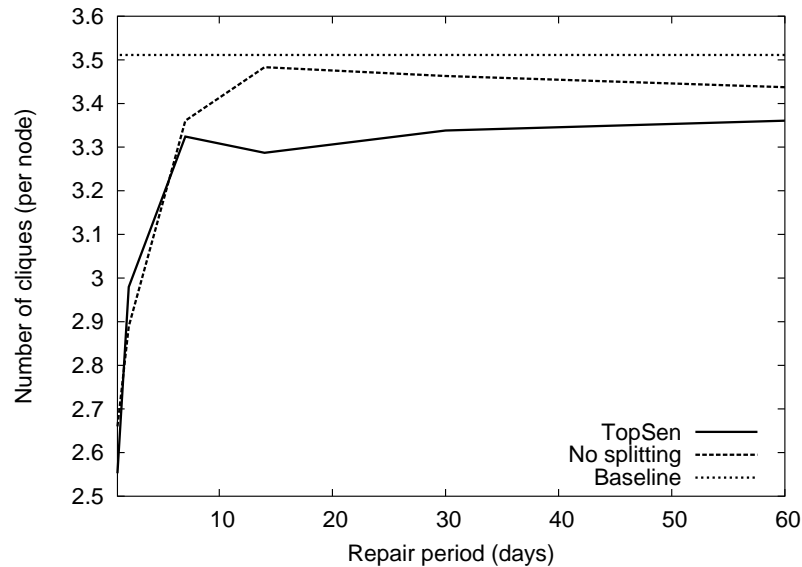


Figure 6.13: Benefits of splitting: Node membership in cliques versus repair period in days.

a weekly or bi-weekly basis does reduce the node degree to some extent but not significantly.

The average number of cliques a node is a member of (see Figure 6.13) behaves in a similar fashion to the average node degree. Employing both types of splitting shows a small but consistent improvement over not using component splitting for a failed node.

The benefits of component splitting and periodic maintenance are much more evident when looking at the average number of components in the system. Performing periodic maintenance on a daily basis provides a 45% improvement in the number of components, which is unfeasible as mentioned above. Performing periodic maintenance on a weekly basis provides a 13% improvement in the number of components, which is still significant. As the frequency of the periodic maintenance decreases, the contribution of the component splitting when a node fails becomes more evident. In fact, without the component splitting the periodic maintenance becomes very quickly ineffective. In addition, the frequency at which benefits are seen from the periodic maintenance is dependent on the expected lifetimes of the nodes. The longer the lifetime the smaller the frequency necessary to see benefits from periodic maintenance.

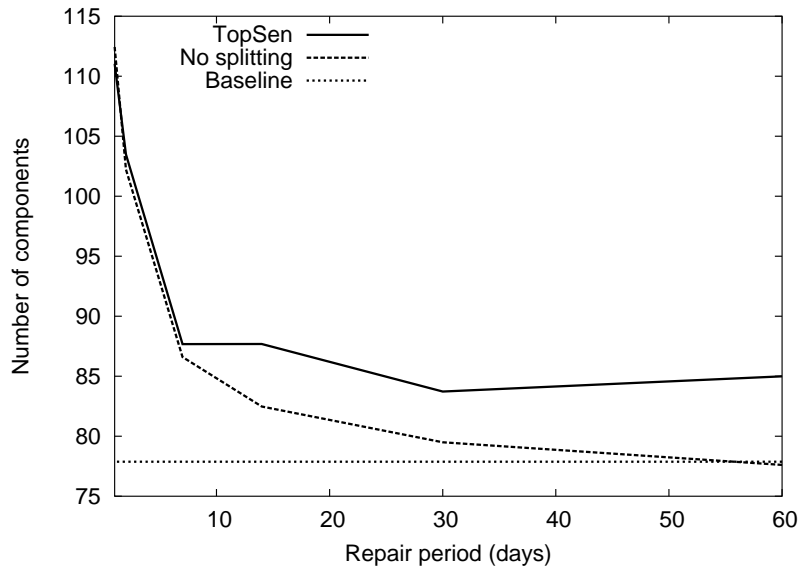


Figure 6.14: Benefits of splitting: Number of components versus repair period in days.

6.2.7 Comparison to DHTs

In peer-to-peer systems that are based on distributed hash tables (DHT), such as Past [69, 68], the topology is such that the node degree is uniform across all the nodes in the system. Replica node selection is solely based on node IDs. Given a node with ID i , the IDs of the corresponding replica nodes are the n closest to i (see Figure 6.15a); the IDs of the replica nodes are also bigger than i . Given this replica selection process each node in the system has a maximum node degree of $2n - 2$, where n is the number of replica nodes (see Figure 6.15b). With TopSen the average node degree is n , where n is the number of replica nodes.

Although the above replica node selection process creates good topologies, where a replica node affects at most $2n - 3$ other replica nodes if it fails, it does not permit the selection of replica nodes based on node attributes. To enable the system to select replica nodes based on node attributes the original selection process is modified to be the n replica nodes with node IDs closest to i and that have the desired node attributes. Given this modification to the selection process the maximum node degree is no longer bounded by $2n - 2$. For example, a system has two types of replica nodes c and r , and all policies must have one node of type r . In this case, all node of type c that are between nodes r_i and r_j rely on node r_j and raise the node degree beyond $2n - 2$ (see Figure 6.15c).

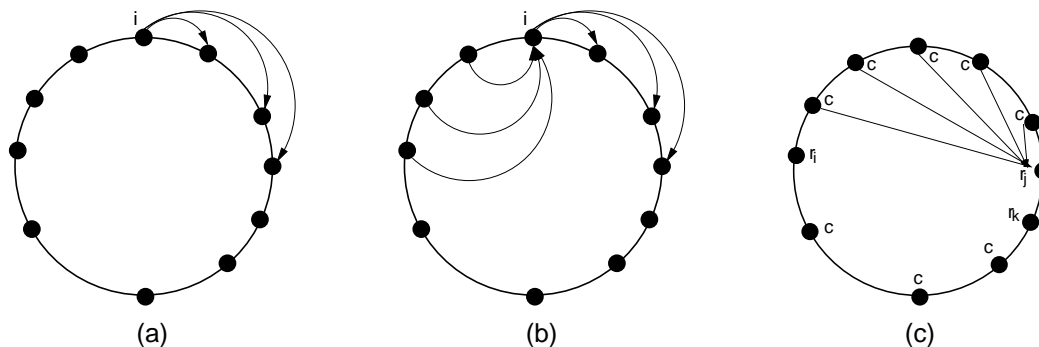


Figure 6.15: (a) The replica nodes for node with ID i . (b) The topology with the maximum node degree for i ; all policies have four nodes. (c) Potential load imbalances in the DHT.

A greater problem with the above modification is that some replica nodes become heavily loaded while others are under-used. This is most likely to happen when two nodes of a rare type are adjacent in the node ID space; i.e., there are no nodes between them. Using the policy just described above, assume that two nodes of type r have adjacent node IDs j and k , i.e., nodes r_j and r_k in Figure 6.15c. In this case, all nodes of type c that are between nodes r_i and r_j will use r_j to satisfy the policy. If the number of nodes is large then the node r_j is going to be heavily loaded and a potential hotspot if it fails. The node r_k is not going to be used and it will only be responsible for a single policy. This type of load imbalance does not occur with TopSen because nodes that are similar and nearby are equally used and selected based on their workload. The probability of an imbalance and its severity depends on the distribution of node types.

To remove the above problem the modified DHT replica node selection process has to be modified again such that the first node of the desired type is not immediately selected. Instead, several nodes should be considered and the node with the least load should be chosen. This selection algorithm is just a special case of the *Least* algorithm used to compare against TopSen. Thus, the expected performance of this algorithm is similar to that of the *Least* algorithm.

6.3 Communication costs

To partly demonstrate the scalability of PDR, the number of inter-node messages required to complete common tasks in the system, such as replicating data, disseminating policy information, and

handling failure is analyzed. For normal PDR operations the four operations that induce a non-trivial number of messages are propagating updates, the heartbeat mechanism, creation of a new replication policy, and the modification of an existing policy. During failure operations, the recovery of the failed node is the main inducer of inter-node messages.

6.3.1 Propagating updates

The most common message that is sent, apart from a heartbeat message, is a store request from a client node to a replica node. The number of store requests made per file modification is proportional to the number of nodes in the file's replication policy. Store requests differ from most of the other messages; while the size of other messages is at most several kilobytes, the size of store requests can range on the order of megabytes because they contain the modified file in its entirety. Thus, under normal operations, the amount of data sent is p times the size of the file, where p is the number of nodes in the policy.

The frequency of an store message for a particular replica node in a policy is governed by how often the associated file is modified and the staleness factor for the node. If the file is constantly being updated then the maximum frequency that an update is sent to that replica node is the staleness factor.

6.3.2 Heartbeat mechanism

The number of heartbeat messages sent is dependent upon the size and topology of the system. Let the topology of the system be represented as a graph G . The vertices are the replica nodes and the edges are the inter-connections between nodes; replication policies create these inter-connections. In particular, a replication policy creates a clique in G .

Let N and M be cliques in G and let $n = |N|$ and $m = |M|$. If N is a disconnected component then the number of heartbeat messages contributed by that component is $\frac{n \cdot (n-1)}{2}$, which is approximately n^2 .

Now, a number of edges between N and M are introduced. This induces additional heartbeat messages which depends on the number of edges introduced. The total number of messages is bound from the bottom when there are no edges, and from the top when N and M form a clique.

Thus:

$$n^2 + m^2 \leq t \leq (n + m)^2, \quad (6.1)$$

where t is the number of messages. Let e be the number of edges between N and M . Then total number of heartbeat messages in a component is:

$$t = n^2 + m^2 + e. \quad (6.2)$$

In general the number of heartbeat messages is equal to the number of edges in the graph. This is an upper bound on the number of heartbeat messages per heartbeat interval assuming that all the nodes in a component have the same interval. Replica nodes that are participating in the same policy have the same heartbeat interval but not necessarily if they are just part of the same component.

Given a set O of connected cliques, $O_0 \dots O_r$, let $o_0 \dots o_r$ be their respective sizes, to compute the total number of heartbeat messages all that is necessary is to sum the number of heartbeat messages generated within each clique and the number of edges between each pair of cliques. Thus t is:

$$t = \sum_i^r o_i^2 + \sum_{\forall i,j=0,i \neq j}^r e_{i,j}. \quad (6.3)$$

The node selection algorithms in PDR strives to maintain small disconnected components. The average size of a component is that of the average number of nodes in a policy. Thus, by Equation 6.3 the average number of messages in a component is going to be about the square of the average number of nodes in a policy; the upper bound is $\frac{p^2}{2}$ messages when the component is a complete graph.

6.3.3 New policy creation

The next most common message is to set a new policy or to change an existing policy. To set a new policy requires at most $3p$ messages, where p is the number of replica nodes in the policy. The policy oracle, which creates the policy, requests updates from each of the replica nodes that has been selected for the policy, resulting in p messages being sent. Each of the nodes responds, resulting in another p messages. Finally, the policy oracle sends out the new policy to each of the participating nodes resulting in a final p messages. If the policy oracle believes that it has up-to-date information about some of the participating nodes, then a fraction of the first $2p$ messages can be avoided.

6.3.4 Policy Update

Let p_o be the number of nodes in the original policy and let p_c be the number of nodes in the changed policy. To change a policy requires at most $3p_c + p_o$. As before, up to $2p_c$ messages are needed to retrieve current replica node information from the new replica nodes and p_c policy updates must be sent. However, the old nodes in the policy must also be notified, of which there are at most p_o nodes.

6.3.5 Failure messages

To restore a failed replica node requires the largest number of messages. When a node detects that a replica node has failed, it notifies the leaders of the policies in which the failed node was participating. If the TopSen algorithm is used for node selection then on average a leader node would receive p messages where p is the size of the average policy. On the other hand, if the Random algorithm is used, then it is possible that a node is connected to every other node and the leader would then receive a message from every node in the system. If a component is a clique of size p , there are p or more policies, and every node is a leader of one of the p policies and participates in all of the p policies then every node sends and receives p messages.

Once the leader selects the replacement it informs the other nodes of the selection. This is another p messages that are sent and received. The upper-bound on the number of messages sent and received to inform nodes of a failed node and of its replacement is $2p^2$.

The selected replacement node then must pull the file data from the other replica nodes. In this step the number of messages equals to the number of files that have to be pulled and their size depends on the size of the files.

Other failure scenarios, such as the failure of a leader node, are similar in cost to the failure of a non-leader replica node. The predominant cost are the notification messages and the re-replication requests. These other failure scenarios add a small and constant number of messages (a dozen messages or so) to the general recovery procedure.

Operation	Number of messages
Store request	p
Heartbeat messages	$\frac{c^2}{2}$
New policy creation	$3p$
Policy update	$3p_n + p_o$
Failure messages	$2c^2$

Table 6.2: Number of inter-node messages needed for a given operation in PDR.

6.3.6 Summary

Table 6.2 summarizes the number of inter-node messages necessary to perform some of the major operations in PDR. The number of messages are presented in terms of either policy or component size; p represents the number of nodes in a policy and c represents the number of nodes in a component. In Section 6.2 it was shown that $p \approx c$. For operations such as the store request, new policy creation, and policy update the number of messages necessary to perform the operation is specified exactly in terms of policy size. For operations such as heartbeat messages and failure messages the number of messages necessary to perform the operation is given as an upper bound in terms of component size. The number depends on the connectivity of the component with the upper bound being attained when the component is a complete graph, every node is a leader in a policy, and every other node is participating in all the other policies.

6.4 Applicability

This section presents two case studies that demonstrate the applicability of PDR. The first is from the Department of Computer Science at the University of British Columbia and the second is from Silicon Chalk [14].

6.4.1 Department of Computer Science, UBC

To determine the potential savings that PDR could provide, file system traces for nine users were gathered and analyzed over a period of a month, September 2003, in the Department of Computer Science at the University of British Columbia. For each participant a script was executed once a day that scoured their home directory searching for files that were modified in the previous 24 hours.

Participants were selected from graduate student, faculty, and technical staff population in the department. A large number of graduate students, faculty, and technical staff were asked to participate and those that agreed were selected as the participants. Administration staff were not asked to participate because they tend not to be heavy users of the department's unified file system. This segment of users was selected because they create and use a wide range of file types and sizes. In addition, their usage patterns are such that they tend to create large amounts of easily recreateable data, e.g. object files, and thus would best demonstrate the effectiveness of PDR.

Two replication policies were created. The first replication policy is equivalent to traditional backup procedures, where data is replicated to the server room once every 24 hours; the replication policy replicates only important files. The files deemed unimportant are the web browser cache directory, object files, and auxiliary files generated by applications such as Latex (i.e., .o, .aux, etc.). The second replication policy is similar to the first, except that in addition, important files, e.g., Latex and Word source files (i.e., .tex, .doc), are replicated within 12 hours of being modified.

On average, the nine users generated approximately 500MB of data for the nightly incremental tape backup. Table 6.3 presents the break down of the file system trace by data type. The amounts are presented as a percentage of the total amount of modified data for a single average user. One participant owned an unusually large mailbox which contributed 30-40% of the total nightly backup; on average a user's mailbox amounts to 10% of their modified data. On average, it was found that the web browser cache and object files made up for 9% of the data being replicated. This is data that is easily re-creatable and does not need to be replicated. Pictures, postscript, and PDF files comprised another 4%. A large portion of this data probably does not need to be replicated because, in an academic environment, postscript and PDF files are usually papers that are obtainable from the web or are generated from source files such as Latex. User's documents and source files contributed about 9% of the total. The first policy would provide a savings of about 10%, and the second would enable users to retrieve a file that is at most 12 hours old verses 24 hours at no extra cost.

The other 68% of the data is in unclassified files. For privacy reasons the traces only had the size, name, and extension of the file. The replication policies were created solely based on file extensions. In addition, no music files were found in the participants home directory. It is believed the reason for this is that home directories are limited to 100MB, and thus users store large data on their research group storage servers that provide general storage space and was not part of the file

Type	Amount
Mail	10%
Object files, browser cache	9%
Pictures, .ps, .pdf files	4%
Documents and source files	9%
Unknown	68%

Table 6.3: Break down of the file system trace by data type for a single average user.

system trace.

PDR would reduce the amount of data that is backed up on a daily basis. Currently, about 1.5 terabytes of data is backed up to tape on a weekly basis at about a cost of a dollar per gigabyte. By using PDR, resources on backing up data such as object files and web browser caches would not be wasted. Other data could be automatically replicated to a variety of local workstations and servers, thus reducing the amount of tape backup even further. The additional benefit of using PDR would be the almost instantaneous availability of lost data due to user error such as deletion. Furthermore, the backups of important data could be done more often and thus be more current.

The first policy, that replicates once a day, provides a savings of about 50 megabytes a day of backup space and bandwidth. The second policy, that also replicates Word and Latex files every 12 hours, provides a savings of 50 megabytes a day of backup space. The same amount of bandwidth is used as with traditional backup techniques except that instead of wasting 50 megabytes of bandwidth on unimportant data, it is focused on providing more current copies of Word and Latex files in the event the files are lost.

In this instance the amount of storage space and bandwidth saved is minimal. The main reason for this is that it is impossible to automatically determine which parts of the 70% of the data that is classified as unknown are important and which are not. With PDR, users can control what is replicated, thus removing the responsibility of classifying the data from the system and the administrator, making the classification more accurate, and further reducing the amount of useless data that is replicated.

6.4.2 Silicon Chalk

Silicon Chalk is a company in Vancouver, British Columbia, Canada that specializes in presentation software for education. Their development environment is primarily Visual Studio from Microsoft. Their code base consists of about three gigabytes of source, resource, and object files. On a nightly basis the entire project is compiled on a build server and an archive (i.e., Zip) is created consisting of source, resource, and object files. On a daily basis developers retrieve this archive as their starting point for their development. Although Silicon Chalk does employ a version control system, the project is so large and takes so long to compile that this is the most efficient route to do development.

A nasty side effect of this practice of distributing a complete build with all the byproduct files is that for a traditional backup and replication system each developer generates three gigabytes of new backup data daily. Thus, not only are large amounts of unnecessary data replicated, both that is easily re-creatable and that has not been modified, but Silicon Chalk would require extremely large amounts of bandwidth if they wanted to perform off-site backups.

With PDR, a small number of policies would be sufficient to replicate only the source files, reducing the amount of data that is replicated. With a small modification to PDR even a greater reduction in the amount of data replicated can be achieved. Currently, newly created files are replicated as per default or set policies. Further replication can be avoided by keeping track of a file's modification time (`mtime`) and replicating the file only if the modification time has changed. Thus the restoration of Silicon Chalk's source archive would result in very few, or zero, files being replicated. This reduction in the amount of replication would make off-site backup feasible since every developer would not automatically be replicating three gigabytes of data on a daily basis. In addition, it would be feasible to replicate important source files more frequently than once a day.

The Linux kernel

To provide additional empirical results the Linux kernel is used as a similar example to Silicon Chalk's code base. The Linux kernel is a 36MB archive that decompresses to a 227MB source tree. Compiling a Linux kernel with a configuration file that is used by the major Linux distributions, such as Fedora [40] and Suse [41], the resulting source tree with object files grows to 1.7GB. The configuration file compiles the kernel with most of the device drivers and file systems as kernel modules. Thus, a total of 1.5GB, or 88%, of the data is easily recreatable and does not need to be

replicated.

Although, this would be feasible to do with a set of custom backup scripts, it does not provide the flexibility of replicating some file more often than others or specifying different protection levels.

Chapter 7

Conclusion and future work

Ahh, this is not the end!
This is not even the beginning of the end!
But it is, perhaps, the end of the beginning!

— Sir Winston Churchill

In this thesis the design and evaluation of a novel replication system was presented. This section summarizes the results, summarizes the contributions, and provides possible future directions for PDR.

7.1 Conclusion

Traditionally, replication systems replicate data in its entirety, without consideration of its importance, or against what failures it must be protected. However, as commodity storage capacity grows exponentially and is filled at roughly the same rate, the strategy of wholesale replication becomes untenable. This is because the growth of other resources, such as network bandwidth, are simply not keeping pace; the cost of putting in another hard drive is dwarfed by the cost of upgrading the backbone or even the local network routers. Thus, to ensure that important data is protected against the appropriate failures the degree of replication must be related to the value of the data; this value can only be determined by the user.

This notion is encapsulated in PDR, a Policy Driven Replication system that allocates replication resources based on per file policies that can be dynamically set and modified by the user. In PDR replication policies are treated as invariants, failed nodes are quickly detected and replacement

nodes selected so that policies are followed. To this end the system constantly monitors the health of nodes on which data is stored in order to quickly detect failure.

The PDR system classifies nodes according to a set of node attributes and uses this classification scheme to decide where to replicate what data, facilitating the protection of data against specific classes of failures. The selection of nodes for new policies and for replacement of failed nodes is sensitive to the topology of the system. This ensures that poor topologies are not created thus avoiding message storms or hot spots in the system. Finally, the system avoids inducing additional overhead on the critical path of file system calls, enabling background replication and minimally affecting system performance.

7.1.1 Summary

PDR is a peer-to-peer system that has no centralized services and all nodes (physical machines) run the same software. A node is a client, a replica, or both. Client nodes have no responsibility but to push data to replica nodes. Replica nodes store data for clients and direct recovery of failed replica nodes.

The PDR software is composed of two parts: the *replicator* and the *policy oracle*. The *replicator* is responsible for replicating data, storing data for other clients, and translating, executing, and enforcing replication policies. PDR hooks into the local file system using a VFS redirector and receives upcalls whenever a file is created or modified. In this way the number of modifications to the operating system is minimized and the task of replication is removed from the critical path of regular file system operations. Although micro-benchmarks show that PDR is on average an order of magnitude slower than the local file system, the evaluation also demonstrated that for normal file system usage the performance of PDR is reasonable. For tasks such as document processing and software development PDR performs on par with local file systems.

A significant portion of user's files are unimportant or easily re-creatable; for example, object files. With a small set of replication policies a user can eliminate the replication of unimportant data, and increase the availability and reliability of important data.

There does not exist a single uniform way to express replication requirements. Users have widely varying views as to the level of protection afforded by a particular backup and replication approach, the cost for providing a specified level of protection, and what data is worth. Given this

non-uniformity, PDR provides a plug-in interface that enables administrators of the system to create a set of tools appropriate for its users to create, set, and modify replication policies. When a policy is created it is translated by the plug-in into a lowlevel representation that PDR understands.

The *policy oracle* is responsible for storing the lowlevel replication policies, selecting replica nodes for new replication policies and for replacing nodes that have failed, communicating with other policy oracles to maintain up-to-date topology information, and for discovering new replica nodes. The most important function of the policy oracle is to maintain a good topology because doing so is necessary for system scalability. If replica node selection is not done wisely then the number of nodes involved in the recovery from a single node failure can be very large. In a system of a million nodes, where the average uptime is 100 days, there are on average about 10k failures per day. Given that many distributed consensus and election algorithms require $O(n^2)$ messages, where n is the number of nodes, it could require up to 10^{16} messages simply to determine who should lead the recovery process and what needs to be re-replicated; to handle a failed node requires $O(n^2)$ messages. For a system of 10k nodes approximately 10^{10} messages would be needed and this does not include the replication of the data itself. Thus, these systems would spend all their time performing recovery.

PDR performs replica node selection in such a way that a large number of small disconnected components are maintained. When a policy is created the nodes listed in the policy become dependent on each other with respect to monitoring and failure handling. If a replica node has many links then its failure affects a large number of other nodes. Thus, one goal of selecting replica nodes is to minimize the number of links any replica node has, thus minimizing the affect of its failure on other nodes. Another goal is to select replica nodes that provide the desired level of protection, which is accomplished through a set of node attributes.

PDR uses a set of algorithms, called *TopSen*, to perform replica node selection that is sensitive to the topology of the system. Through simulation it is shown that TopSen maintains a topology such that the number of messages necessary to perform recovery is constant. That is, the cost of recovery is independent of the size of the system.

7.2 Contributions

The research presented in this thesis makes three contributions. First, a method for describing nodes and a generic plug-in infrastructure was designed and implemented that enables the translation of replication requirements into a set of replica nodes. This node description method is able to describe all physical aspects of a replica node and the plug-in interface enables administrators to create interfaces that are tailored to users' views on data value and protection level. These highlevel policies are then translated, by the plug-in, into lowlevel policies the replication mechanism understands and uses to select the replica nodes. In addition, this plug-in interface enables the exploration of interfaces for managing policies.

Second, an analysis of the network topology created by the PDR system was performed and a replica node selection algorithm was developed that maintains good topologies. This algorithm is sensitive to the topology of the system, selecting replica nodes such that a failure of a replica node affects the minimum number of other replica nodes. The replica node selection algorithm does not require complete knowledge of the entire system topology to operate, although the more of the topology the algorithm knows about the better its selections become. Not only does the algorithm strive to maintain a good topology, but it actively rearranges the location of the data to further improve the topology.

Third, a prototype of the PDR system was designed and implemented to evaluate the ideas present in this thesis. The system is mostly portable, it is integrated with the local file system, requires little or no modifications to the operating system, and no kernel level modifications. A small amount of latency is introduced into the standard file system operations, but for real-world tasks this overhead imperceivable.

7.3 Future work

During the design and implementation of PDR a number of interesting issues and questions arose. There are a number of improvements and enhancements that could be made to PDR and a number of outstanding questions that deserve further investigation.

7.3.1 Further improvements

There are a number of improvements to PDR that would improve its functionality and performance. Currently, PDR employs the Coda [36, 77] redirector to hook into the local file system. The primary drawback to using the Coda redirector is that it intercepts the majority of the system calls. Thus, everything except `read` and `write` is handled at userlevel, which adds unnecessary overhead.

A redirector should be developed that acts as a tee rather than redirecting the call up to the user-level. That is, the local file system should handle all the file system operations and only messages specifying that an operation was executed should be propagated up; similar to the *netlink* interface in Linux for network devices such as firewalls and routers. In this way, the user would obtain the full performance of the file system and the replication aspect would be completely removed from the critical path of file system operations.

Currently all inter-node communication in PDR uses TCP. Since the majority of the messages are small using a stream based protocol is unnecessary. Instead a reliable UDP protocol could be used which would further reduce some of the communication overhead.

7.3.2 Future enhancements

Recently Bell Canada introduced a service called Bell Business Backup which enables users to selectively replicate their data to Bell's data centre. Bell Canada provides an interface that is similar to Windows Explorer where users specify how often a file or a directory should be replicated. There are several drawbacks to this approach. First, it is relatively expensive given the cost of storage; ranging from \$4.95/65MB/month to \$39.95/10GB/month. In addition, the network connection has bandwidth, upload, and download caps. Thus, to use the full 10GB of storage the user may need to pay for a higher bandwidth connection. Second, data that is older than 15 days is automatically deleted, thus this service is only good for files that are constantly being used; one cannot use it as an archive repository.

The above service brings out an interesting time dimension that PDR currently does not support but can easily be modified to do so. It would be useful to specify a time constraint for data and then potentially perform an action on the data; for example, store file x for only t days and then delete it. To support this functionality it would be necessary to create a thread that would process events from a priority queue that is sorted based on time. The policy specifier would need to be slightly

modified to include a time and action attribute. When a file is replicated an event for the desired action is created and inserted into the priority queue to be executed at the desired time.

The functionality to execute an action brings forth two questions. First, what actions are useful and second, where would the actions execute. For example, actions such as *delete file* or *keep n versions of the file* could be implemented as part of a suite of actions that PDR provides by default. Thus, such actions would automatically be run on the replica nodes and the client or the leader node would not need to worry about their execution. Other functions could be provided by a piece of code that is shipped to the replica node during the initial replication of a file.

Finally, in a replication system such as PDR the notion of *file groups* or *checkpointing* [30, 63] would be useful. That is, either implicitly or explicitly by a user, a file would be marked as being part of a set of files that form a consistent checkpoint. This functionality would be beneficial for users such as developers who want to restore a set of files from a specific point in time. Currently such a mechanism is not implemented in PDR and it is not immediately clear as to the best way to implement such functionality. The difficulty lies in defining appropriate metadata and maintaining its consistency when a file belongs to multiple file groups.

7.3.3 Open questions

There are a number of questions that have been left unanswered and require further investigation. They primarily deal with policy setting interfaces, useful policies, and inter-operability between domains with respect to node attributes.

The first question is very much a human-computer interaction issue. As more information goes digital and as more backup services become available I believe that users will become more aware of the potential dangers to the safety of their data and thus they will start thinking about protecting their data. Currently, each backup service provider and backup application has their own interface to specify and manage backup. It is unclear whether there would be a convergence with respect to replication or backup, or whether users would simply adapt to whatever service they are using.

The second question is also very user centric. If there is a convergence of interfaces then it may be possible to develop a standard set of useful policies, but first it is necessary to define what is useful and to determine what this set is. This is an area that is only slightly addressed by the policy plug-in mechanism. For example, consider two users, one that lives in an area of high earthquake

activity and the other in an area of heavy flooding. Both users want to protect their data against natural disasters. Both probably need to replicate to a node that is not in the region or that is built to withstand the specific types of natural disaster. The goals of both users are quite similar but their needs are significantly different. Is it possible to encompass the users needs with a single policy? How would it be specified?

The last several questions deal with node attributes, specifically what node attributes are useful and how to map between them. This relates to the first two questions since the node attributes are the system level representations of a user's goal to protect their data. For example, nodes in the earthquake zone may have a property that rates the soundness of the building they are in, while the nodes in the flooding area may have a property that specifies the elevation of the nodes or the amount of precipitation the region receives. In the earthquake zone the higher the soundness the more reliable the node, in the flooding zone the higher the elevation or the less precipitation the region receives the more reliable the node. How can these node attributes map to each other so that nodes in both domains correctly interpret the implied reliability level?

I implemented, I measured, I'm done!

— Norm Hutchinson

Bibliography

- [1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, December 2002.
- [2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 627–644, October 1976.
- [3] R. Anderson. The eternity service, 1996.
- [4] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.
- [5] Network Appliance. <http://www.netapp.com>.
- [6] Computer Associates ArcServe. <http://www3.ca.com/>.
- [7] B. Baker and R. Shostak. Gossips and telephones. *Discrete Mathematics*, 2(3):191–193, 1972.
- [8] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.
- [9] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of IPTPS'03*, 2003.
- [10] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total recall: System support for automated availability management. In *Proceedings of the First ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.
- [11] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proceedings of the international conference on Measurements and modeling of computer systems*, pages 34–43, 2000.

- [12] D. Brodsky, A. Brodsky, M. Feeley, and N. Hutchinson. Policy driven replication. Technical Report TR-2003-15, University of British Columbia, 2003.
- [13] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of the ACM SIGCOMM*, pages 56–67, 1998.
- [14] Silicon Chalk. <http://www.siliconchalk.com>.
- [15] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.
- [16] Bram Cohen. Incentives build robustness in bittorrent, May 2003.
- [17] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, December 2002.
- [18] Frank Dabek, M. Frans Kasshoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, October 2001.
- [19] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *In Proceedings of The International Conference on Dependable Systems and Networks (DSN 2001)*, July 2001.
- [20] Eran Gabber, Jeff Fellin, Michael Flaster, Fengrui Gu, Bruce Hillyer, Wee Teck Ng, Banu Ozden, and Elizabeth Shriver. Starfish: highly-available block storage. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*, pages 151–163, June 2003.
- [21] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, March 1988.
- [22] Gnutella. <http://www.gnutella.com>.
- [23] Ashvin Goel, Calton Pu, and Gerald J. Popek. View consistency for optimistic replication. In *Symposium on Reliable Distributed Systems*, pages 36–42, 1998.
- [24] Jim Gray, Pat Helland, Patrick O’Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, June 1996.
- [25] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

- [26] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the SIGCOMM Internet Measurement Workshop (IMW 2002)*, August 2002.
- [27] Vassos Hadzilacos and Sam Toueg. *Fault-Tolerant Broadcasts and Related Problems*. Distributed Systems. Addison-Wesley, 1993.
- [28] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 155–162, October 1987.
- [29] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, August 1995.
- [30] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Conference: January 17–21, 1994, San Francisco, California, USA*, pages 235–246, Winter 1994.
- [31] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [32] John A. Howard. An overview of the andrew file system. In *Proceedings of the Winter 1988 USENIX Conference*, pages 23–26, February 1988.
- [33] M. Ji, E. W. Felten, R. Wang, and J. Pal Singh. Archipelago: An island-based file system for highly available and scalable internet services. In *Proceedings of 4th USENIX Windows Systems Symposium*, August 2000.
- [34] Michael K. Johnson. Red hat’s new journaling file system: ext3. Technical Report <http://www.redhat.com/support/wpapers/redhat/ext3/index.html>, Red Hat Inc., 2001.
- [35] Kazaa. <http://www.kazaa.com>.
- [36] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25, October 1991.
- [37] John Kubiawicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, November 2000.
- [38] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

- [39] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, *Computer Architecture News*, pages 84–93, October 1996.
- [40] Fedora Linux. <http://fedora.redhat.com>.
- [41] Suse Linux. <http://www.suse.com>.
- [42] Mallik Mahalingam, Chunqiang Tang, and Zhichen Xu. Towards a semantic, deep archival file system. In *The 9th International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003)*, 2003.
- [43] Keith Marzullo. Theory and practice for fault-tolerant protocols on the internet, October 2002.
- [44] EMC² Corporation. <http://www.emc.com>.
- [45] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.
- [46] Sun Microsystems, Inc. NFS version 4 technical brief, October 1999.
- [47] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.
- [48] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 15–28, October 2001.
- [49] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, December 2002.
- [50] Napster. <http://www.napster.com>.
- [51] NetBackup. <http://www.veritas.com/>.
- [52] Legato Networker. <http://www.legato.com/>.
- [53] HP Omniback. <http://www.hp.com/>.
- [54] John Ousterhout. A trace-driven analysis of the unix 4.2 bsd file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 79–86, December 1985.
- [55] Overnet. <http://www.overnet.com>.
- [56] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 28(2):155–180, February 1998.

- [57] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data: 1988 Annual Conference, Chicago, Illinois, June 1–3*, pages 109–116, 1988.
- [58] David A. Patterson. A conversation with Jim Gray. *ACM Queue*, 1(4):53–56, June 2003.
- [59] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Thurlow. The nfs version 4 protocol. In *In Proceedings of the Second International SANE (System Administration and Networking) Conference*, May 2000.
- [60] Fernando Pedone, Matthias Wiesmann, Andre Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *International Conference on Distributed Computing Systems*, pages 464–474, 2000.
- [61] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Sixteenth ACM Symposium on Operating Systems Principles*, 1997.
- [62] Peter Braam Philip. Removing bottlenecks in distributed filesystems: Coda intermezzo as examples.
- [63] D. Presotto. Plan 9. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 31–38, April 1992.
- [64] Calton Pu and Avraham Leff. Replica control in distributed systems: An asynchronous approach. In *SIGMOD Conference*, pages 377–386, 1991.
- [65] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Conference on File and Storage Technologies (FAST '02)*, March 2002.
- [66] Peter L. Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the ficus file system. In *Proceedings of the Summer 1994 USENIX Conference*, pages 183–195, 1994.
- [67] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the oceanstore prototype. In *Conference on File and Storage Technologies (FAST '03)*, March 2003.
- [68] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms, Middleware 2001*, November 2001.
- [69] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, October 2001.

- [70] Yasushi Saito. Optimistic replication algorithms. Technical report, University of Washington, 2000.
- [71] Yasushi Saito. Consistency management in optimistic replication algorithms. Technical report, University of Washington, 2001.
- [72] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 1–15, December 1999.
- [73] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, December 2002.
- [74] Yasushi Saito and Henry M. Levy. Optimistic replication for internet data services. In *International Symposium on Distributed Computing*, pages 297–314, 2000.
- [75] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, December 1999.
- [76] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the eighth ACM symposium on Operating systems principles*, pages 96–108, 1981.
- [77] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment. In *Second IEEE Workshop on Workstation Operating Systems*, September 1989.
- [78] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
- [79] Tracy F. Sienknecht, Richard J. Friedrich, Joseph J. Martinka, and Peter M. Friedenbach. The implications of distributed data in a commercial environment on the design of hierarchical storage management. In *Proceedings of the 16th IFIP Working Group 7.3 international symposium on Computer performance modeling measurement and evaluation*, pages 3–25. Elsevier Science Publishers B. V., 1994.
- [80] D. Simpson. Does HSM pay? be skeptical! *Datamation*, 41(14):322–337, August 1995.
- [81] Sleepycat Software. The berkeley database (berkeley db).
- [82] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1993.
- [83] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report TR-819, Massachusetts Institute of Technology, March 2001.

- [84] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings Third International Conference on Parallel and Distributed Information Systems*, pages 140–149, September 1994.
- [85] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 172–183, December 1995.
- [86] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.
- [87] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of Middleware*, 1998.
- [88] R. Y. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 71–78, October 1993.
- [89] Hakim Weatherspoon, Tal Moscovitz, and John Kubiawicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proceedings of International Workshop on Reliable Peer-to-Peer Distributed Systems*, pages 362–367, October 2002.
- [90] Hakim Weatherspoon, Chris Wells, Partrick E. Eaton, Ben Y. Zhao, and John D. Kubiawicz. Silverback: A global-scale archival system. Technical Report CSD-01-1139, University of California Berkeley, March 2001.
- [91] Matt Welch, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 230–243, October 2001.
- [92] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.
- [93] B. Y. Zhao, J. D. Kubiawicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report CSD-01-1141, University of California Berkeley, April 2001.