Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2010-047 September 22, 2010

A File Location, Replication, and Distribution System for Network Information to Aid Network Management

Tiffany Cheng

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

A File Location, Replication, and Distribution System for Network Information to Aid Network Management

by

Tiffany Cheng

S. B., E.E.C.S. M.I.T., 2009

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
August 2010

© 2010 Tiffany Cheng. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now and known or hereafter created.

Author...
Department of Electrical Engineering and Computer Science
August 27, 2010

Certified by..
Dr. Karen Sollins
Principal Research Scientist
Thesis Supervisor

Accepted by..
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

A File Location, Replication, and Distribution System for Network Information to Aid Network Management

by

Tiffany Cheng

Submitted to the Department of Electrical Engineering and Computer Science

on August 27th, 2010, in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis demonstrates and evaluates the design, architecture, and implementation of a file location, replication, and distribution system built with the objective of managing information in an Internet network. The system's goal is to enable the availability of information by providing alternative locations for files in case of situations where the original piece of information cannot be found in the network due to failures or other problems. The system provides the mechanism for duplicating files and executes the act of placing them in multiple locations according to predefined rules for distribution. The resulting system is a working model for a file management system that can exist over the Internet and will aid in overall network management by organizing and overseeing the information found within a network.

Thesis Supervisor: Dr. Karen Sollins

Title: Principal Research Scientist

Acknowledgements

After five years at MIT, I can finally admit that I am grateful for having come here for college. There were many moments during my years here where I just wanted to quit and leave, but now having survived, I am thankful for my experience here and am especially thankful for those people who have always been there for me throughout it all.

To my family: mom, dad, Michael, Margaret – you will always mean the world to me. I love you.

Kat, you are my first and greatest friend at MIT. I am glad I got to share my college experience with you, and I look forward to many more years of friendship.

Jizi, I am glad you are always there for me, in happy times and in sad. Your memorable quotes are encouraging, witty, kind, and hilarious. You remind me to be optimistic, and you are one of the best things I got out of being at MIT.

Joyce, dearest friend, you empathize with and understand me so well. I would never have finished college if it were not for you. My gratitude is ineffable. Thank you for everything.

Garrett, your caring and trust mean more than you will ever know. Because of you, I have learned a lot about myself, and it has been a tremendously growing experience. I truly cherish our friendship.

To all my friends, I feel very lucky to have all of you in my life. From my oldest childhood friends to all the people I have met along the way, I hope our friendships are long lasting and ever growing.

Last, but not least, to Karen, my adviser but also a role model and friend. Thank you for being academically supportive and for patiently teaching me about the world of research and academia. I sincerely admire your achievements in your field and am grateful and proud to have had the chance to learn from you, whether it was research related matters or life wisdoms. You inspire me to put my intelligence and education to good use in the future.

This work was supported by NSF Grant 0915629, 'NeTS: Small: KPBase: Core of the Knowledge Plane for Network Management.

Contents

1 Introduction 13

1.1 Background 13

1.2 Overview of the System 17

2 Related Work 22

2.1 The Need for a Better-Informed Network 23

2.2 What Types of Information are Required or Important 26

2.3 How to Find Desired Information in a Network 29

2.4 Related Work as a Foundation 34

3 System Components and Architecture 35

3.1 Description of File Location and Distribution System's Components 35

3.2 Network Model Components 36

3.2.1 Machine 36

3.2.2 AutonomousSystemGenerator, AutonomousSystem 36

3.2.3 AutonomousSystemPath 39

3.2.4 ASPathDatabase 40

3.3 File Management Architectural Components 42

3.3.1 Open AFS 43

3.3.2 FileUniqueID 45

3.3.3 FileLocation 46

3.3.4 DistanceAwayRule, RuleOperator 47

3.3.5 FileUIDToLocationsAndRule 51

3.3.6 LocationsToFileUID 55

3.4 Summary 58

4 Implementation of the System at Work 59
4.1 Set-up 59
4.1.1 Topology 59
4.1.2 File Space Structure of AFS 62
4.2 Capabilities of the System Through Examples 66
4.2.1 Case 1: Copying files to an AS less than some distance away 67
4.2.2 Case 2: Copying files to an AS greater than some distance away .. 76
4.2.3 Case 3: Removing the rule after copies are distributed 80
4.2.4 Case 4: Removing the original file when it has distributed copies 82
4.2.5 Case 5: Copy a file, remove the original, and set a new rule on the remaining copies 85
4.2.6 Case 6: Copy a file to a specific AS 86
4.3 Addendum to the Rule 88
4.4 Recapitulation 89
5 Conclusion 91
5.1 Overview 91
5.2 Going Forward 93
6 Appendix A: File Management Objects and their Methods 97
6.1 ASPathDatabase 97
6.2 AutonomousSystem 97
6.3 AutonomousSystemGenerator 97
6.4 AutonomousSystemPath 97
6.5 BinaryTree 98
6.6 DistanceAwayRule 98

6.7 FileLocation 98

6.8 FileUIDToLocationsAndRule 99

6.9 FileUniqueID 99

6.10 LocationsToFileUID 99

6.11 Machine 99

6.12 Node 100

6.13 RuleOperator 100

7 Appendix B: Directory Watching Objects and their Methods 101

7.1 AbstractResourceWatcher 101

7.2 BaseListener 101

7.3 DirectorySnapshot 101

7.4 DirectoryWatcher 101

7.5 FileListener 101

7.6 IFileListener 101

7.7 IntervalThread 101

7.8 IResourceListener 101

7.9 IResourceWatcher 101

7.10 MyFileListener 101

8 Appendix C: File I/O Objects and their Methods 102

8.1 FileSystemManipulator 102

8.2 LongToByte 102

8.3 MD5Calculator 102

9 Works Cited 103

List of Figures

Figure 1 Network topology of sixteen autonomous systems, each composed of three machines 61
Figure 2 Example AFS directory tree 63
Figure 3 Structure of AFS file space for the experimental network 64
Figure 4 Example of copying files across autonomous systems 66
Figure 5 Creation of FileUniqueID for a new file in the network 69
Figure 6 FileUIDToLocationsAndRule and LocationsToFileUID after notifying the system of the new pingData0.txt file, before the copying of the file has happened.. 70
Figure 7 ASPathDatabase entry for AS6 71
Figure 8 File space after pingData0.txt is copied from as6 to as15 72
Figure 9 LocationsToFileUID after pingData0.txt is copied to AS15 from AS6 72
Figure 10 FileUIDToLocationsAndRule after pingData0.txt is copied to AS15 from AS6 73
Figure 11 Schematic of pingData0.txt copied from AS6 to AS15 74
Figure 12 File space after pingData1.txt is copied to AS10 from AS6 75
Figure 13 The tables after pingData1.txt is distributed with a < 4 DistanceAwayRule 76
Figure 14 File space after pingData0.txt has been copied using the > 3 AS hops away rule 78
Figure 15 Tables after pingData0.txt has been copied using the > 3 AS hops away rule 79
Figure 16 FileUIDToLocationsAndRule object after copying pingData1.txt to ASs > 3 hops away from AS6 80
Figure 17 LocationsToFileUID object after copying pingData1.txt to ASs > 3 hops away from AS6 80
Figure 18 File space after removing the > 3 rule for the pingData0.txt file. Copies that existed in AS2, AS3, AS4, AS12, AS13, and AS14 are now gone. Original file in AS6 still exists. 82

Figure 19 Tables after removal of > 3 rule for the pingData0.txt file 82
Figure 20 File space after deletion of original pingData0.txt file from AS6 83
Figure 21 Tables after original pingData0.txt file in AS6 is deleted 84
Figure 22 File space after new rule is imposed on the new original file in AS13. Copies placed in AS15 and AS11. 86
Figure 23 Tables after new rule is set on the new original file pingData0.txt in AS13. 86
Figure 24 File space after spot duplication of data2010Aug08.txt from AS2 to AS4 87
Figure 25 Tables after spot duplication of data2010Aug08.txt from AS2 to AS4..... 88

1 Introduction

1.1 Background

The Internet in its present form is a great achievement. Its great strengths include its heterogeneity – illustrated by the Internet's truly global reach, spanning continents and allowing exchange of information across political and socioeconomic boundaries; its rich end-system functionality over a transparent network, evidenced by how a naive user can utilize applications such as Facebook to connect to people anywhere in the world without needing to know the underlying details; and its multi-administrative structure, manifested as the ability to have multiple Internet Service Providers (ISPs) procure and maintain Internet connections to millions of people without needing one overriding entity. Built from multiple layers of abstraction, the Internet is impressively robust in being able to support all the information that flows through it, and it is this robustness that allows for the Internet's openness to new applications, adaptability of its protocols, and most importantly, its ability to evolve over time.

However, despite the many benefits of the Internet, the Internet is not without flaws and grave weaknesses. It has become increasingly obvious that today's data networks are surprisingly fragile and increasingly difficult to manage. Small failures in the Internet network are capable of cascading and amplifying into problems that may end up being orders of magnitude greater than the originating error. This magnification of errors is one of the more glaring problems of the current incarnation of the Internet, especially when contrasted with the desirable contained resolution of complications found in stable systems.

Presently, the Internet is maintained by network managers, each of whom is in charge of their own domain of the network. As an illustration, a network manager for an ISP will provide support and maintenance for a portion of the network owned by the ISP. These network managers each use their own tools or methods to diagnose and solve problems within their own domain, but as soon as a

problem appears to be out of their scope of knowledge or responsibility, they pass the problem on to the next individual, who may or may not know how to resolve the problem. To make the situation even more complicated, network managers are not able to collaborate fully since they are generally employed by competing ISPs, who may not necessarily want to help other providers or may be unable to do so due to security issues. Thus, if the network manager encounters an issue that falls outside of his ISP's network domain and inside another ISP's jurisdiction, the network manager sometimes must resign the problem as unsolvable. But it is important not to undervalue the effort put forth my network managers. They do try as much in their power as possible to resolve issues. They do coordinate informally and collaborate pair-wise when they notice problems with one particular neighboring network, but the communication is often ad hoc and with little or no organized approach or protocol. The end result is mostly a system where each ISP maintains as well as possible the health of its own domain but ignores the larger good.

This ineffectiveness of managing network problems only on a localized scale on a case-by-case basis is a well-recognized problem, and many researchers have come to the general conclusion that a single architecture or framework for network management would improve the overall management of the widely distributed network environment. The goal is to promote network-level objectives and network-wide views. By promoting these higher level goals, the network no longer has to shuffle problems around without actually solving them, and will function better as a whole by achieving the purposes of its end users. The network should be cognizant of a high-level view of what its purpose is, whether it is the goals of the applications running on it or that of the end users. When the network knows what it is being asked to do, it will be able to take care of itself rather than depending on people - in this case network managers - to attend to it.

In order for a network architecture to accomplish the network's overall goals, the architecture needs to be aware of how its components are functioning. Network failures can occur anywhere, from DNS server failure to path failures, and there are various likelihoods associated with each failure. To work around these failures, it

would be helpful if the architecture could gather pertinent information, such as data on link latencies, bandwidth, or loss rate. Traditionally, such information on the functioning of a certain link is stored at a particular router. For example, a router will store information about its outgoing links. This brings us back to our most basic conundrum. When links fail, the network wants to collect attributes about those links, and logically, it will try to go to the nodes connected to those links to query for information about the links. However, if the link and its general vicinity have failed, trying to go to the source of the failure is futile since the general area is nonfunctional and the information it holds is now inaccessible. Another example besides link failure would be congestion in the network. If a particular router is experiencing problems with congestion, trying to query that router for information in order to fix the congestion problem only worsens the issue since the queries overload the router with additional work, adding to the congestion.

In order to provide a unified approach and global perspective to networks, the Internet as a whole must support the creation, storage, propagation, and discovery of a variety of information, including observations and current conditions. To promote these goals, one will need to figure out how to store information about the network effectively so that the aforementioned problems do not arise. By collecting information about the network such that users can access that information readily, the higher goals of the network are obtained more effectively with fewer failures. Therefore, it is necessary to find a method of storing the information strategically around the network in such a way so that if there are failures or difficulties in portions of the network, the critical information stored in those failed locations will still be accessible to the rest of the network. The straightforward approach is to store multiple copies of the information in various places distributed across the network in locations that will hopefully be accessible at all times. The goal is for the network to make better high-level decisions with these pieces of information that should always be readily available.

To look at the scenario from one extreme, one can hypothetically store information everywhere in the network, at every node. If any part of the network

were to fail, the information there will always be available elsewhere. However, this over-redundancy is obviously inefficient, and in addition, policies may exist that dictate storing information in certain locations is illegal or not allowed. At the other extreme, we can store the information about a node at that exact one location and name it so that the network will know how to find and go to that particular location for the information. Since we do not want to store information at every node, the problem becomes a question of which locations we should use to store the information and how much redundancy is necessary, or how many locations we should use overall. We can argue that the probability of multiple locations failing decreases with each additional location, and can therefore set threshold probabilities to determine the total number of locations.

In order to diagnose a failure in the network at a certain area, logically there should be some model of the failure's location and the information associated with it. The types of information that are stored in the network are very broad and can encompass anything from low-level information including link latency, average congestion of a link, bandwidth, or a router's packet queue history, to higher-level information such as AS states. The hope is that all these pieces of information will help to diagnose failures. Our goal is to build a system that models the locations of these failures and provides an architecture of how to find the necessary associated information, and also to provide a protocol on how to organize and distribute the information overall. The goal, then, is to find a method of storing the information strategically around the network in such a way so that if there are failures or difficulties in portions of the network, the critical information stored in those failed locations will still be accessible to the rest of the network. We do this by storing multiple copies of the information in various places distributed across the network, with the idea that a sufficient number of well-chosen locations will minimize the probability that any piece of information is not available at any point in time, regardless of the state of localized portions of the network. The key here is that the locations for these copies are not arbitrary, but instead purposefully chosen for some quality that indicates that they would be a reasonable location. By storing copies in

strategically chosen locations, the network will be able to achieve the overall objective of making better high-level decisions, since critical information about the health of any portion of the network will always be readily available.

1.2 Overview of the System

Our project works within this problem space of trying to solve the problem of finding and distributing information to intelligent places, which is a subspace of the even larger problem space of effective network management and the endeavor to create a smarter, more self-sufficient network management world, . The objective of our project is to manage the files of information and to replicate them to multiple locations in such a way that it is easy to provide this information to tools that will use the information to aid network management

To achieve the objective, our work provides the outline of a system whose purpose is to give structure to the amorphous problem of managing information that exists within the network. By managing the locations of the information in the network, this system and its associated information will help to support and promote the goals of network management. The information in our system consists of files, found all across the network, that contain data or content that could potentially be helpful or important in achieving the tasks and intentions of the network. Our system provides the network with an overarching framework for discovering these relevant or critical files, cataloging the files in an organized fashion so that they are locatable, distributing copies of essential files so that more than one copy exists in the network according to concrete rules, and maintaining a database of all the files in the network. The main purpose of the system is to ensure that important files used in network management are available and accessible at all times. The system supports the ability to replicate important files so that if their original location in the network is not reachable, there are other possible locales for the file. The system achieves the replication by following a set of basic rules for how a file should be copied and distributed. Finally, by maintaining knowledge of the files and all their copies in

the network, the system enables the facile finding of desired files for network management. As a result, our file location, replication, and distribution system achieves the initial goal of finding a way to distribute information so that it resides in desirable locations in the network.

There are a set of core problems that we needed to answer in order to build the system's functions as just described. The first problem was to define more concretely the definition of what copying a file to an intelligent location exactly means. Our system wants to copy files to intelligent locations so that if the original file is unavailable, the copy will still be accessible, but there is no clear meaning for what intelligent means. We argue that intelligent copying means distributing files such that they are physically distant; to achieve that, we distribute files across Autonomous Systems. As an example, we imagined a scenario where a file exists on a specific computer in a particular office building in some country. We want this file to be available in case its home machine fails, so now the question arises as to where to place a copy of the file. It would seem that placing the file at a neighboring computer in the same office building would be ineffectual. If the office building were to lose power, the neighbor as well as the home machine would be inaccessible and our copying will not have achieved the goal of ensuring that the file is available. The conclusion is that the more physically distant the files are, the better the probability is that one failure in one file will not affect the other. We believe that this independent failure is achievable at the Autonomous System (AS) level. ASs usually control a physical region, so different ASs will have domains in different regions (with some overlap). Because ASs are different entities, they are not likely to fail simultaneously, and since they usually reside in disparate physical regions, we do not need to account for environmental or natural disaster effects. In addition, the purpose of the existing ASs is that they are persistent, long-standing entities. For these reasons, we demonstrate the abilities of our system at the AS level. The end result is that our system uses rules for distributing copies of a file according to AS distance. The system views a network as interconnected ASs, and

AS distance is measured in the number of hops between ASs. The rules of our system take on the syntax of some number "of AS hops away from the origin," and the system uses this rule to copy files to other locations, where locations and AS are synonymous.

We can imagine the usefulness of having additional rules to help with ensuring that files are available. For instance, having rules that know how to reduce the load on an AS would support the availability of files by preventing congestion and denial of service errors. Replicating a file so that it exists widely enough so that no single location needs to handle traffic for that file would likely significantly boost network performance as well as aid the goal of making sure a file is available as well as accessible. These additional rules are interesting future problems to solve. As for now, we look simply at distance rules to help files maintain availability.

The second problem our system dealt with was detailing the process of identifying and discovering new information in the network. When information for network management is created in the network, our system must be able to find this new information and incorporate it. As a demonstration of how such a system might work, our system first builds an Andrew File System into the network at the Autonomous System (AS) level of the network as a storage substrate. We chose to implement our system using AFS, but other possible storage substrates are possible. We happen to use AFS for its helpful qualities, which we now proceed to describe.

If we imagine that an AS is represented by a single computer, having AFS run over a system of these AS computers means each computer sees an identical file system space. Using a single file system space means that different ASs see the same tree of directories in this file system space and can therefore see files that exist in other ASs. Having the AFS system built into the network is important because the uniform file space provides the foundation for being able to copy files across the network and across different ASs. AFS abstracts the copying of files in a network to be identical to copying files within a file system.

The difference lies in that different parts of the file system reside within different ASs. Using AFS solves the problem of identifying and discovering new information since our system can discover new information in ASs by watching the part of the AFS file space that belongs to each AS. The system identifies which information in an AS is actually crucial to network management by enforcing the assumption that important files will be created in specific parts of the file space and not in others.

A third problem that the system resolves is the precise copying rules it would support. Our system has the capability to copy a file of information to "less than" some AS distance away, "greater than" some AS distance away, "less than or equal to" a distance, "greater than or equal to" a distance, and "equal to" a distance. The system also supports copying files to an absolute AS destination. The last significant problem the system addresses is the question of how to organize files in the system. There is ambiguity over whether a file and its copy should be considered a single file or two independent ones. Our system treats a file and its copy as the same file in that our system implements identifiers that it gives only to unique files. A file and its copy would therefore share the same identifier.

There were other trade-offs made and problems evaluated in the construction of our network information management system, and the details are described in the following chapters. Chapter 2 discusses the previous research done and related work in our problem space of locating and finding information in the network. Chapter 3 details the individual components of our file location, duplication, and distribution system by reviewing how each component's capabilities, how it functions, and how they all relate to each other. Chapter 4 demonstrates the capabilities of our system on a real network by studying case studies of different copying scenarios for a file. Chapter 5 discusses future work and other possible uses for the system and concludes with an analysis and review of the system and its impact on network management.

Ultimately, our system is a functional model and proof of concept for how network management information within a computer network can be coordinated in a regulated and organized fashion. Our system demonstrates that information can be kept accessible even in the face of failures at specific locations. Our system illustrates that files can be successfully distributed and the new locations tabulated so that alternative copies to a desired file can still be found. Overall, this system achieves the objective of providing information for network management even when the information's origin is unavailable, and it accomplishes this by replicating and distributing files to disparate locations based on predefined rules.

2 Related Work

The internet as it currently exists is a system that requires a persistent amount of upkeep and support. At any given moment, there is maintenance performed for a multitude of problems and failures that arise in the network. As we move forward, it is becoming evident that there needs to be a systematic and more automated way of directing, governing, and supervising the network so that it successfully survives and adapts to future demands. Currently, the internet is maintained by individuals at the individual autonomous network level as opposed to supervised as an integrated whole. This status quo is becoming increasingly problematic as it is becoming more and more apparent that management at the microcosmic level is not enough to solve the macrocosmic problems that arise on the greater Internet networks. The current method of managing of the internet by fragmenting the larger Internet into small divisions of control, fails to address many of the flaws in the network simply because most flaws cannot be contained to one small domain or principality and instead tend to affect multiple areas. It is not uncommon for network managers to ignore issues that affect them if they find that the origin of the issue seems to be outside of their realm of control.

This ineffectiveness of only managing network problems on a small scale is a well-recognized problem, and many researchers have come to the general conclusion that a single architecture or framework for network management would improve the overall management of the widely distributed network environment. By having a single architecture for network management that governs the entire Internet as a whole, we are able to solve a myriad of problems that treatment of the Internet as smaller pieces of a whole could not, because viewing the network as a whole makes the architecture better informed. As opposed to multiple individual network managers trying to run the network piecemeal, this single architecture will hold responsibility for maintaining the

network and achieving the network's overarching goals since it knows what these goals are. The single framework will include abilities such as diagnosing failures in the network as well as making the desired automated, educated decisions of how best to fix and maintain the successful functioning of the network as a whole.

In considering related work for the background to these problems, we will separate our review into three topics. There are three different aspects to the problem, and we will talk about the relevant work in these separate contexts. First, we review work that examines the need for information in managing networks. The first section discusses how this need for information dictates the general requirements for a new system of network management. We then review work that examines the problem of assessing what information is required. This second section answers the question of how to determine what kind of information is necessary to perform network management. The crucial problem here is finding out what one needs to know when there is a failure in the network. We conclude with a set of related work on placing and finding this information, as well as how to move this information around the network. This final section examines how to store information and, consequently, how to locate it.

2.1 The Need for a Better-Informed Network

In this section, we examine the arguments made for why we need information to manage networks.

There has been much research done in the plausibility, necessity, achievability, and functionality of employing a single framework for network management. The clean slate 4D Approach proposed by Greenberg[1] argues that there is a need for a higher level decision plane built over networks that will be responsible for making decisions in order to achieve network-level objectives. Greenberg et al. are in clear support of having a single framework for network management since the "decision plane" outlined in *A Clean Slate 4D Approach to Network Control and Management*[1] is simply just another name for the single

framework. The 4D project proposes leveraging the packet-handling plane below this higher 4D plane in order to accomplish its network-level goals, and its design principles include a desire to promote network-level objectives and network-wide views. The 4D approach recognizes the necessity of promoting higher level, network-wide goals and why it is crucial to move away from the current Internet management model of small domains. It explains that with this new decision plane, the network no longer has to shuffle problems around from domain to domain without actually solving them, and that as a result, the Internet will function better as a whole in achieving the purposes of its end users.

Similarly, the Knowledge Plane[2] described by Clark, Partridge, Ramming, and Wroclawski is another name for the desired single framework for network management. The Knowledge Plane calls for a network designed with a cognitive system where the network has a high-level view of what its purpose is. These goals are anything from the goals of the applications that run over the network to the goals of the end users. The Knowledge Plane asserts that if the network has the ability to know that it is being asked to do, such as achieving an application's purpose, it naturally follows that it will be able to take care of its own functioning rather than depending on people such as network managers to attend to it. Again, the goal is to not require human supervision of the network. The knowledge plane is self-aware, has a high level view of the network, and is able to make informed decisions without relying on maintenance at the local level, which is exactly what the goal of the single framework is.

In order for a single network architecture to accomplish the network's overall goals, the architecture needs to be aware of how its components are functioning. In order to maintain the network properly, the single framework needs information about the network itself before it can make informed decisions. As elaborated in *An Architecture for Network Management* by Sollins[3], gathering of information to inform decisions then becomes crucial for the functioning of the single framework. With this gathering of information come affiliated issues such

as storage and discovery of the information, as well as how to find and use the information to when determining what network-level decision to make. As an example, failure in the network is a crucial type of information that the single network framework will need to have in order to better achieve the network's goals. Failure information is helpful in diagnosing problem sites as well as averting future issues. Network failures can occur anywhere; whether it is a DNS server failure or a path failure, there are multiple sources for failure, and there are various likelihoods associated with each failure. To work around these failures, it would be helpful if the architecture could gather information about each failure, such as data on link latencies, bandwidth, or loss rate around the time of the failure occurs.

Traditionally, information such as the attributes of a certain link is stored at a particular router. For example, a router will store information about its outgoing links. This brings us to our most basic conundrum: when components such as a group of links fails, the network architecture wants to collect information about those links. It will try to go to the nodes connected to those links to query for the information about those failed links. However, if the link and its general vicinity have failed, trying to go to the source of the failure is futile since the general area is down and unreachable. The information about these nodes which is stored in their routers is then also inaccessible. Ironically, the location of the most helpful information concerning the failure is inaccessible because of the failure itself, and without this information, the network architecture is not able to make the best possible decision.

Another example or a complication besides link failure would be congestion in the network. If a particular router is experiencing troubles with congestion, trying to query that router for information in order to fix the congestion problem only worsens the issue since the queries overload the router with additional work, which in turn adds to the congestion.

Hearkening back to the Knowledge Plane, in order to provide a unified approach and global perspective to networks, the Knowledge Plane must support the creation, storage, propagation, and discovery of a variety of information, including observations and current conditions. To promote the goals of the Knowledge Plane and its proposal of a single framework, there is a need to decide how to store information about the network effectively so that the aforementioned problems do not arise. The high-level goal of our work is to address these problems by ensuring that information in the network is readily available and reachable by users of the network, whether these users are human consumers, applications that run over the network, or other services. Our goal is to create a system for the storage and propagation of the variety of information about the network necessary for network management, and in creating this system, to pay particular attention to the importance of the location of information. We believe that in order to optimize the management of the network, we must be attentive to how information used in the management process is stored and distributed, and in particular, we attest to the value of associating information about the network with its location in the network. Because we store and propagate information to specific locations, it is worth looking at work that has been done in the field that relates to systems describing how one finds desired information in the network.

2.2 What Types of Information are Required or Important

In this section, we discuss the types of information that are important to network management and how they have been dealt with in other research.

An Architecture for Inter-Domain Troubleshooting by David G. Thaler and Chinya V. Ravishankar[4] discusses the goal of their Global Distributed Troubleshooting system (GDT) to match up information from a requester to the source of the information. In GDT, the type of information passed around is troubleshooting information for the network, and the system is meant to allow

users to report problems and receive timely feedback. In order to match up a request for troubleshooting information to the source where the troubleshooting information can be found, GDT uses expert location servers. The expert location server functions so that when a user queries an expert location server, the server is either able to return the necessary information, or it passes the request on up a hierarchy of expert location servers. The expert location servers in GDT have a knowledge of their own location and connect to each other into a hierarchy based on which other server is closest. Each region of the hierarchy is an expert on what information is held in that region. Users attach to the nearest available expert location server and the requests are passed from region to region until the desired information is found. Although the expert location servers use location to form regions of information, they do not use it in direct relation to the information. We argue that information and its location are a valuable association, and in GDT, this association is not made. Any region of the hierarchy of servers in GDT could contain any type of information, whereas we believe that it is possible to place information in the network at intelligent locations. In GDT, that would mean that the servers are experts on their information because that location is the appropriate place for that type of information.

The Splice system created by Moore et al. at HP Laboratories[5] is an example of a system that does take into account location information in a network, albeit not a computer network. The Splice system was created in order to manage a huge number of sensor measurements in data centers with the goal being to utilize the sensors' data sets to better manage resource consumption of huge data centers more effectively. It does this by associating a sensor's set of data with its exact location in the warehouses that contain these data centers. There are essentially two parts to the problem space of information and its location in regards to networks. One is the association of information with its location, and the second is how to use that location data to create policies that help manage the network better as a whole. The Splice system explores the first

part of this problem space in that it outlines a framework for how to associate data with its location, and expands a little into the second by giving examples for how the framework is able to handle certain problems that require location information. Instead of computer networks, Splice functions over sensor networks, but a sensor network's similarity to real computer networks makes Splice a good example that can illustrate several useful lessons for computer network management, particularly since the efforts of managing a data center using measurements and data collection are analogous to the efforts of managing a computer network. Like the internet, the drive to improve operating efficiency of data centers is motivating the development of knowledge planes that take responsibility for the coordination, monitoring, and control of large data computing infrastructures. Moore et al. found that there were benefits to extending a data center's knowledge plane to include data from environmental sensors. What this means is that the having knowledge of the physical location and the spatial and topological relationships of a data center's components was beneficial in helping to build a system that successfully monitored the data centers. This is very similar to our goal for the knowledge plane overlaying computer networks; we also want to embed physical location into the knowledge plane of the internet in order to improve management of the overall network.

Another piece of work that is related to our goal of making network management information reachable and available is CAPRI: A Common Architecture for Distributed Probabilistic Internet Fault Diagnosis by Lee[6]. CAPRI is meant to be a framework within which different diagnostic agents can interact and share information. Similar to our goal of providing a system to store and distribute information so that users can access it, the goal of CAPRI is to assess root cause localization and fault diagnosis. It does this by providing a system where heterogeneous diagnostic agents can communicate their own acquired observations and knowledge to each other, which allows CAPRI to probabilistically infer the cause of network failures. CAPRI describes a system

for how to enable various diagnostic agents to find the information they need. The CAPRI method of finding information is to create a more precise structure where the different agents use a specific service description language, message exchange protocol, and processing procedure to describe the information they own or the information they want. Lee demonstrates the capabilities of CAPRI by focusing primarily on the diagnosis of reachability failures in which a user cannot access a particular resource or service on the Internet, which is also the problem we want to solve. However, Lee's work is not concerned with where the source of information is located or where it currently resides, which is the focus of this work. Instead, CAPRI's focus centers on how to communicate this information to all the other diagnostic agents within the network without paying attention to the location of the information's origin.

2.3 How to Find Desired Information in a Network

This last section discusses the salient point to our problem space of how to find information in a network.

There has been much work done in the area of storing and finding information in a network. In the past, much research has been devoted to the process of finding general information, meaning how to find any information that any application on the network requests, and not specifically network-related information and attributes. In the past, few of the theories proposed have emphasized the value of location to information and its potential effect on the finding process. The use of location metadata in order to help determine and establish the source of information and services in the internet network has substantial consequences on the theories and models behind finding desired information in the network.

A popular architectural paradigm is the Publish-Subscribe model for getting the applicable information from the source to the asker. *Fast and Flexible Forwarding for Internet Subscription Systems* by Joanna Kulik[7] groups the

various forms of publish-subscribe under the name Internet Subscription Systems, indicating that each is a distributed mechanism for notifying subscribers as quickly as possible to the arrival of relevant information on the Internet. Kulik proposes a match-structure forwarding system that is a hybrid between single-identifier multicast systems and content-based multicast systems, which improves both performance and scalability. Ultimately, though, like all Internet Subscription Systems, the focus of the match-structure subscription system centers on the mechanism of getting the appropriate information to the requester. No location information is leveraged in the mechanism because similar to all other publish-subscribe models, the match-structure model makes the assumption that a requester's information is possibly widely disseminated across the network and could potentially be located anywhere. Thus none of the various publish-subscribe systems make judgments or assumptions of where information could be located.

By proposing that information is a key attribute to be considered, the major publish-subscribe archetypes of topic-based, content-based, and type-based can be vastly improved since there would no longer be a need to blindly forward messages or multicast throughout the network in order to match information from the publisher to the subscriber. Protocols such as LIPSIN: Line Speed Publish/Subscribe Inter-Networking[8] would no longer be required to optimize every small detail of the multicasting forwarding fabric, which is currently in publish-subscribe systems so that the network is not overwhelmed while trying to route data. The publish-subscribe methods that include knowledge of the location of certain types of information would now be able to execute their mechanisms of matching publisher to subscriber much more methodically and effectively, which demonstrates the value in associating information in the network to its location. An example would be a subscriber now knowing exactly what part of the network to look for published information, as opposed to simply broadcasting the desired subscriptions to the entire network, followed by the entire network's publishers

then trying to communicate with the individual subscriber. Associating location with the information itself means that there is no longer a crucial need for painstaking, stringent optimizations. Publish-subscribe is highly supported as an architectural model for an improved internet, and a progressive step would be to incorporate location as an integral part of the publish-subscribe paradigm.

The design and implementation of an intentional naming system by Adjie-Winoto, Schwartz, Balakrishnan, and Lilley[9] outlines the Intentional Naming System (INS), which is a resource discovery and service location system for dynamic and mobile networks of devices and computers. Their goal is also to coordinate locating information (which they call resource discovery) and getting it to users of the network. In the INS, applications use a language to describe what the information they are looking for, rather than specifying the location or hostname where the information they want is located.. The idea behind the INS is that users of the network – in this case applications on the network -never need to know precisely where information can be found. Users simply broadcast what they want to look for, and a system of INS resolvers fulfills the requests for information. INS resolvers receive a request and either return to the requester a list of the locations where information can be found, or forward the request on to another INS resolver (INR), which will again make the decision of resolution or forwarding. The resolvers automatically distribute request resolution load among the resolvers, and the INS uses a decentralized network of resolvers to fulfill user requests. Users can attach to any of the resolvers in order to resolve their requests. These INS resolvers essentially contain the crucial information about how to realize user requests, and as such must always be made available to users of the network. In order to load balance, INS resolvers can spawn more resolvers if there appears to be a particularly heavy load of requests. Because the INS resolvers are a distributed network, there is no location implication for the resolvers. They can be located anywhere in the network at any time and so can their spawn. Users simply send requests to whichever INS resolver is most

convenient and the network of resolvers figures out how to achieve the request. We propose that location does play a key role in the network and that the network's performance can be improved with some knowledge of location. With the INS, it is possible that if the resolvers were not absolutely distributed and instead had some knowledge of location information tied to the resolvers, users could better identify which resolvers it would be most useful to send their requests to. Also, since the INS resolver network requires sending updates to other resolvers, it is possible that using location information would be helpful in determining which resolvers need to be updated with which information. Again, we reinforce the idea that location of information is valuable to a network. In INS, the important information is located in INS resolvers and so they should not be completely, absolutely widely distributed.

Systems such as INS and GDT, discussed in section 2.2, choose to keep the necessary information in a set of servers widely distributed across the network. There is also the possibility of storing information everywhere and not just in designated servers. Information-based networks and content-centric networks as described by Jacobson et al.[10][11] as well as the related PSIRP project[12] choose to distribute the information across the entire network with no control over location and opt to copy the information multiple times across the network to increase availability. The approach completely decouples content from location since it decides that the content simply needs to be copied everywhere and there is no guideline for where "everywhere" entails. The content-centric network outlined by Van Jacobson and others assume that users do not need to be aware of where the desired content resides. In the content-centric network, information is widely distributed and copies cached everywhere. Users retrieve one of these copies, and it does not matter which copy is chosen. This shields users from needing to know where to look for the content. While shielding the users makes getting information easier since the system is free to choose which copy of the content to deliver, this approach still has the problem of not being able to fulfill

user requests. There is a possibility that the copy of the needed information is somewhere else in the network, but there is no guarantee.

On the polar opposite end of content-centric networking's approach to storing information in the network is the iPlane project[13][14]. Whereas the idea behind content-centric networks is to place information everywhere and to replicate as much as possible, iPlane and its related project iPlane nano actually stores all its crucial information at a central location. From this central location, copies of the information are distributed out to appointed servers. Users of the information collected in the iPlane project then can retrieve the information from the servers. By taking the opposite stance of most theories by centralizing information all in one location, the iPlane project has the obvious vulnerability that if the headquarters go down, then users can only work with stale information found in the appointed servers. More importantly, iPlane also decouples information from its location properties. By accumulating all data at one centralized point, iPlane ignores the possibility that information might be better utilized if it made use of its location properties instead of just forcing all information to one place, which may or may not be an intelligent choice for location. The iPlane project is still worth noting because in addition to providing one possible model of how to find information in the network, it is also a preliminary step towards the goal of providing a more high-level view of the Internet, showing how it is possible to provide accurate predictions of Internet path performance so that overlay services function more reliably. Overlay service examples include voice-over-IP or peer-to-peer file sharing. This is a significant step forward towards the goals of creating a Knowledge Plane[2] because the project demonstrates that by collecting information about the network such that users of the network can access that information readily, the higher goals of the network- for example the goals of applications running on the network - are achieved more effectively with fewer failures.

After reviewing previous work in the field of network architecture on the question of how to find and distribute information, a common theme is that these previous efforts all decouple information from its location properties. However, the network ultimately is built on devices and entities that are physical objects with physical properties which are eventually affected by locations and environments. The reasonable approach is to incorporate this location data and use it when considering how to distribute information in a network.

2.4 Related Work as a Foundation

Unlike much of the previous work in the field, we work to create a system that does associate information with its location, since we believe that such a link would be valuable in creating a more efficient and effective architecture. We create a system where information is distributed intelligently by making decisions of how to propagate the information based on its location information. Crucial information then has a much higher probability of being available and accessible when needed because the information was distributed to rationally chosen locations.

3 System Components and Architecture

3.1 Description of File Location and Distribution System's Components

This chapter describes the architecture of our system and how it functions. We are creating a system that is meant to function within today's Internet network. Our system takes our entire network's multitude pieces of information, which are files, and performs a systematic, governed copying and distribution of these files across the internet at desired locations. The system takes any pieces of information (files), whether new or existing, in the network and replicates and places the files in certain locations in the network according to pre-specified directions and rules. The system then proceeds to keep track of the multitudes of files by maintaining working knowledge of the origins of files, the locations of the copies, how to find the files, along with a whole array of other attributes such as which autonomous system files originated from. Files may be associated with a rule governing how this file should be treated. Because we care particularly about the replication of information across the network, we study rules and policies that determine how to distribute copies of some file from its original location.

We created all of the components of the overall file location and distribution system using Oracle and Sun's Java language. This chapter will first explain each of the different functional units, their purpose, and their own distinct specialized operations. It will explain each component's architecture and implementation and its specific functions and what purpose they serve. This chapter will explain the building blocks of the system and how the various units integrate with each other to achieve the overall functionality of the system from a bottom-up approach. Continuing chapters will then further explain how the different components integrate to accomplish the system's actions of maintaining a database of the files in the network. In addition, there will be a detailed look at

the procedures of distributing these files and copies of these files around the network as necessary. How the system functions will then be explained top-down.

3.2 Network Model Components

This section describes the abstractions that describe a real world computer network.

3.2.1 Machine

A computer network is fundamentally a set of connections among some group of computers as well as other devices such as routers or firewall boxes. In our system, we begin by modeling the computer network by starting with the basic unit of a single machine. If we view a network as consisting of devices with links between the different devices, machines are these devices – any non-link element in our network. These non-link elements could be the familiar stand-alone consumer computer or other network devices such as routers. Our abstraction for these assorted, distinctive devices is the Machine component. The Machine object has one defining attribute which is a unique name.

The conventional means of naming a Machine object is to pull the name from a descriptor file that exists on the machine at a known location. For our implementation, we use the name of the machine found in the file /etc/hostname, but we do provide users of the system the freedom and flexibility to name machines as they wish. In that sense, virtual names are a possibility since users are not restricted to using names that strictly come from files that physically exist on the machine, although for practical purposes, it is most likely pragmatic to have the name of the machine noted at some location. In addition, to provide yet even more flexibility, machine names are not static and can be modified using the setName method. The machine name we use is a globally unique identifier.

3.2.2 AutonomousSystemGenerator, AutonomousSystem

Although it is true that a computer network can be relegated to the description of an agglomeration of interconnected machines, for our system, we abstract this

view of the network to the higher level of Autonomous Systems, as is done in the Internet today. As opposed to describing the network as a collection of connected computers, we describe the network as a collection of connected Autonomous Systems (AS), basing it off of the real world Autonomous Systems that exist presently such as Internet Service Providers or other sovereign entities. An AS is generally composed of a subset of computers within the entire network, and this subset acts as one entity, which is the AS. By taking this higher level approach to the network, our system performs logic and acts upon the network with an Autonomous System as the basic unit. It is important that our system exists at this level because our system is designed for locating and distributing files within and outside of an AS, *not* a machine. Our system's base logic does not function along the machine level. For example, when we are attempting to distribute files, it is not a question of in which particular machines other than the original machine should the file of information exist. The actual logic would be a question of within which AS should the file exist in relationship to its own originating AS. It is important to function at the AS level and not at the machine level because the idea behind an AS is that is it a longer-lived entity. Since they are more static, they are better placeholders for information than machines. Any given machine could be down or up at any given time and, in further complication, could be mobile. However, by distributing files at the much more static AS level, we can benefit from the increased organization and stability. Also, by abstracting the details of what particular machines belong to a certain Autonomous System, the AS now has the liberty to store a file within itself in its own desired way. For instance, if a file's policy is to have a copy in Autonomous System X and Autonomous System Y, AS X and AS Y only need to promise that they will indeed own a copy and can then privately decide for themselves how they will hold that copy and in which machine. From our perspective, the machines within an AS are considered equivalent, and it does not matter which machine in the AS stores a file; we want to make the distinction between the ASs, not their machines.

In the real world, there is a defined and limited number of Autonomous Systems, many of them Internet Service Providers (ISPs). It is not a chaotic world but instead an organized world with rules that govern how different ASs connect with each other.

The AutonomousSystem object represents Autonomous Systems in the real world. An AutonomousSystem object is identified by a unique integer ID number and contains a list of all the Machine objects that make up that AutonomousSystem object, much like how a real world Autonomous System is a group of machines. We can add or remove Machines from the AutonomousSystem using its addMachine and removeMachine functions as well as query it for its name and its ID number. The name of an AutonomousSystem always follows the format "as*xxx*" where *xxx* is some integer number. The names of all ASs are unique so that we can identify any of them by name. This simplifies the logic of the file distribution system since we can create logic that dictates "distribute file *i* in as*xxx* and as*yyy*" knowing that as*xxx* and as*yyy* will not refer to the same Autonomous Systems.

The AutonomousSystemGenerator is a listing of all the Autonomous Systems in our artificially created network. There can be anywhere from only one AS to thousands, depending on how many ASs we choose to generate for our network. Having the AutonomousSystemGenerator keep track of all the ASs in the network ensures that we maintain a finite number of ASs in our system and that they are not created or destroyed. To reflect the finite and distinct number of ASs in the real world, our system uses an AutonomousSystemGenerator object to create as well as keep track of AutonomousSystem objects for our system. AutonomousSystems cannot be created independently and must use the createAS method. AutonomousSystemGenerator's createAS method creates an AS, ensuring that there are no ASs created arbitrarily. The AutonomousSystemGenerator then maintains a list of all Autonomous Systems

in the system so that the system will always know the total number of ASs in the network and what the ASs are.

3.2.3 AutonomousSystemPath

Thus far we have defined Machines and the AutonomousSystems that group those machines. However, these are not individual, stand-alone objects, and to explicitly show the connections, we have the AutonomousSystemPath object, which is a representation of the connections between Autonomous Systems in the real world. Because we do not concern ourselves at the machine level, we do not model the connections amongst machines within an Autonomous System. We believe that how machines are connected within an Autonomous System is under the domain of control of that Autonomous System, and an AS can connect its machines in whatever way it pleases without us trying to put a model on these connections. To model the connections amongst AutonomousSystem objects within our system, we have the AutonomousSystemPath component. The AutonomousSystemPath represents a path through the Autonomous Systems of a network. It is defined as a list of the Autonomous Systems a packet would unidirectionally traverse through when trying to get from some start AS to some end AS. For instance, if a packet wants to go from as4 to as7, the order of Autonomous Systems it goes through when traveling through the network might be as4, as1, as6, as7. An AutonomousSystemPath would store this list of ASs in order, from beginning to end. There can be AutonomousSystemPaths within other AutonomousSystemPaths. Going back to the example just illustrated of the path "as4, as1, as6, as7"; "as1, as6, as7" is another different AutonomousSystemPath since it has a different starting Autonomous System (as1 as opposed to as4). An AutonomousSystem by itself is an AutonomousSystemPath with a length of one, where the start and end AS are the same AS.

An AutonomousSystemPath's endpoint ASs can be discovered using the getStart and getEnd methods. The getLength method returns the length of the

entire Autonomous System Path. Each AS in a path is some number of hops from the starting AS of a path. We use the getAS*x*HopsFromStart method, where *x* is some integer number of hops from the start AS, to figure out which AS is that distance away. For the path "as4, as1, as6, as7", as1 is one hop away from as4, as6 is two hops away, and as7 is three hops away. Which ASs are a specific number of hops from a particular AS is important information to know because when it comes to distributing information across a network, a common and significant measure is distance. When it comes to the Internet network, physical distance such as kilometers is not necessarily the meaningful unit. Instead, the AS hop distance is more meaningful because each different AS is a different realm of control and can act independently, something a distance measure such as kilometers cannot capture. By being able to measure AS distance relative to other ASs, we are able to employ the logic in our system of distributing files at some minimum or maximum AS distance from the originating AS, e.g. distribute file *i* within four hops of the originating of its current location AS.

3.2.4 ASPathDatabase

ASPathDatabase, which stands for an Autonomous System Path Database, is the object that represents the entire Internet network. For our system, we view the network as a collection of paths through the Autonomous Systems, and it is the ASPathDatabase that keeps track of all these many paths through the network by keeping a list of all the AutonomousSystemPaths in a network.

When ASPathDatabase is first instantiated, it is an empty database, signifying a new and empty network. In order to populate the network with paths, we use the addPath method to add AutonomousSystemPath objects into the ASPathDatabase. Adding any single AutonomousSystemPath into ASPathDatabase also adds all the valid permutations of paths found within the initially supplied AutonomousSystemPath. For example, the adding the AutonomousSystemPath "as4, as1, as6, as7" also adds the subpaths "as4, as1"; "as4, as1, as6"; "as1, as6"; "as1, as6, as7"; and "as6, as7." Recall that an AS is the

entity across which we route in our system, and if a packet can get to as7 from as4 by going through as1 and as6, then it is self-explanatory that the packet should be able to get to as1 and as6 from as4. ASPathDatabase also adds the unit length paths of the AutonomousSystems themselves, e.g. just "as4." The single unit paths are useful because they allow for standardization of the idea of hop length. An autonomous system by itself would be 0 hops away from itself. By allowing a single autonomous system path, we do not have to include extra logic just to deal with the zero case. This allows for more robustness.

In reality, we would not enumerate all possible paths in a real-world implementation since that is an enormous scaling problem that has $O(n^2)$ complexity. The central concept is that given a path, ASPathDatabase implicitly knows the subpaths within the path and when queried, will be able to give the appropriate responses for which ASs are connected to which other ASs. Our implementation enumerates all the possible paths for didactic purposes and is a simplification for demonstration purposes.

In order to organize the numerous AutonomousSystemPaths in the ASPathDatabase, we organize all the various AutonomousSystemPaths by the start AS. All the AutonomousSystemPaths with the same start AutonomousSystem are grouped together into one set. The ASPathDatabase has a table that maps from a string that is the name of an AutonomousSystem to the set of AutonomousSystemPaths that all begin with that AutonomousSystem. Simply put, the table maps String → Set, where String is an AS name and Set is a set of AutonomousSystemPaths.

In order to figure out if the Internet network has a path from one AS to another, we can use the containsPath method of the ASPathDatabase that takes in a start AS and an end AS. To check if a valid path exists between the two, ASPathDatabase goes to the table and retrieves the set of paths for the indicated start AS. It then checks the end AS of all the paths to see if the desired end AS

exists. It is possible that there are multiple paths between two different ASs, in which case the ASPathDatabase returns the shortest path by length of hops.

3.3 File Management Architectural Components

The previous section describes the different system components that make up the model of a real world network. Built on top of this set of network objects is the system for file management. This section describes the different components that have the responsibility of organizing, copying, and managing the files in the network.

We just outlined the various system components and objects that model a real life computer network. Now we turn to the file storage and distribution portion of the system that functions over the network model. We describe the various components and how they integrate to accomplish file distribution across the different AutonomousSystems and their neighbors in the network. We explain how we keep track of which files are located where and how the distribution is accomplished.

We illustrate how the different components of how the file copying and distribution system works by using a simple example. A computer network abounds with a copious amount of information on every machine in every AS. Our illustrative example takes a look at a single file, which represents a single piece of information that sits on some Machine that resides in AutonomousSystem as4. As4 is connected to other ASs in the network. There is the AutonomousSystemPath “as4, as1, as6, as7”; “as4, as5, as2, as3, as8, as9”; and “as4, as15, as14, as13, as12.” These paths are all in the ASPathDatabase, and the various different ASs are all listed in the AutonomousSystemGenerator. The ultimate goal of our system is to make information available in the case of failures by copying it and disseminating it to locations as dictated by some rule or logic. In this example, we want to make the single file available outside of as4 in the case that “as4” is down and cannot provide service by using the example

logical rule that we want the file replicated to all ASs more than three hops away from AS4. This means that the file should be found in "as8" and "as12." Our system provides a mechanism for the replication and distribution of this single file outside of "as4" according to this logical rule by using its various components to accomplish the intended goal of placing the file into "as8" and "as12."

3.3.1 Open AFS

In our DETER experiments, we define both a network topology and an overlaid AS topology. In this work we run Ubuntu's Hardy Heron OS on each experimental node. The nodes are connected in a way so that certain nodes are a part of designated ASs, and the whole is a model of a real Internet network.

As mentioned earlier, we choose AFS, and in particular OpenAFS running on Ubuntu's Hardy Heron, as our file storage substrate. Our set-up is as follows. We make each machine an OpenAFS client and server. OpenAFS is the open source implementation of the Andrew File System (AFS). It is a distributed networked file system that presents a location-transparent uniform file name space to all the clients of AFS. The idea behind using AFS is that we want to use its client server model. We are trying to place information at a particular location so that other machines in the network can access the information at that particular location. The particular location that stores the piece of information or the file can be viewed as the server for the piece of information with every other machine in the Internet a client when it accesses the information at that particular server. By using OpenAFS, we automatically get this server-client model implemented.

The benefit of using OpenAFS also includes presenting a uniform file space to all the machines in the network. This means "addressing" our pieces of information becomes very straightforward. In associating a file with a location, the implication is that the file must have an "address" or a way to find the file. In a computer network, this can become really complicated. One possibility would be to address the file by an IP address followed by some system of directory names

down to where the file ultimately resides. OpenAFS provides a means of systemizing the addressing procedure because it is a file system. It makes use of the familiar systems of paths in a file system as an analogy to a file's address in a computer network. Therefore, our file addresses simply become a familiar file system path which embed an AS name in the file system hierarchy, often along with a machine "name." To users, this becomes very straightforward as to how to find a file. Simply follow the path of directories to where the file resides. For an administrator behind the scenes of the system, this makes it easy to distribute file locations to different physical localities without end users realizing since the file's address remains the same. For example, a file in as4 titled "routetrace.txt" could be found in an address like "/internet/as4/nAmerica/region7/pc10/measuredData/routetrace.txt". The users who use that file only need to know the address. The details of which particular machine in as4 the file resides in is nicely abstracted away, which means the administrator can move the file into whichever desired machine without always needing to change the address of the file. Although the path names do include a machine name if desirable, the abstraction comes in that the name of the machine is not tied down to a specific, physical machine forever. If that tangible machine were decommissioned, the machine's name could be ported to a different machine and the file placed on the replacement machine.

For our system, we use the standard that the file location must include which AS it belongs to in its address. Following this criterion means we are able to figure out any piece of information's AS location. We utilize the knowledge of the AS to distribute the information away from the original AS.

Deciding to use OpenAFS for our system meant making some tradeoffs. Though AFS in general is scalable to thousands of clients, this is a minuscule number compared to the number of machines in all of the Internet network or even in any given autonomous system. AFS can be run efficiently on a decently large network of computers, like the dozens of machines used in our miniature

model of the network, but the Internet is on the scale of millions of machines, which means our usage of AFS cannot completely transparently carry over to the real world cleanly. However, the transferring may still be possible. One potential solution in trying to accomplish the transferring of AFS from an experimental group of computers to the wider internet is possibly the AFS architectural unit of a cell. A cell is a cluster of computers in the same network that are clients of the same set of AFS servers. AFS allows for different cells to communicate with each other, but for the most part, cells exist independently and run their own system of AFS. This means their file space is different from another cell's files space. The addressing of files using paths is still possible even with these separate cells since path names can easily include the name of the cell as part of the pathname. Current AFS systems do just that by having the cell name early on in the path name. For example, /afs/isi.deterlab.net/users/tiffanyc/tmp is the path for the directory tmp for a user named tiffanyc in the isi.deterlab.net cell

3.3.2 FileUniqueID

As a brief reminder of the long-term goals of the Internet, there exists a desire for the Internet to evolve a single architecture for management. As mentioned before, this architecture needs to take in information about the Internet in order to make its management decisions. There is a plethora of information in the Internet found in files all across the network, and in order to be able to use any of these files, we start by solving the elementary issue of how to label and identify any of these files. For our system, we give all unique files in the network their own unique id, which in our system is represented by the object FileUniqueID. Different files receive different FileUniqueID's; copies of the same file do not have different FileUniqueID's but instead all share the same FileUniqueID. We do not give copies of a file their own FileUniqueID because the goal of a FileUniqueID is to identify unique files that hold their own distinct content. Replication of the same file does not introduce fresh information into the network, and so copies do not receive a different FileUniqueID.

In order to generate a unique identification for a file, we use the MD5 hash algorithm with the contents of the file and its modification date as input to the hash function. We use the MD5 algorithm because it is a well known algorithm that gives us a reasonably high chance of producing a unique ID number for different files. The MD5Calculator component of our system is an object that, given any generic file, reads in the contents of the file along with the file's date and returns the 128-bit result of the MD5 hash as a 32-digit hexadecimal string. This string form of the 128-bit MD5 hash becomes the FileUniqueID of files. Having the global identifier FileUniqueID aids in systematizing our file management system since it remains the same for a file across the universe of the Internet.

3.3.3 FileLocation

We label a file using a FileUniqueID. To represent the file's specific location, we use the FileLocation system object. The substrate we build our system on is AFS, and in AFS, files are located by a path name. In AFS, this path name describes how to navigate a tree layout of directories to the desired file. Our system also utilizes the path name addressing system of AFS to locate files, which means all files in our system have their own path within the system, and it is this path that locates the precise location of the file. A FileLocation object encapsulates the file's path, which are strings.

In AFS, a file's path can be any listing of directories in the tree, starting with topmost directory. In our system, we recognize that files in a network are found in machines in Autonomous Systems, and as such, the organizational standard we impose is that one directory in the path of a file must be the name of the Autonomous System to which the file belongs. This means file "sample.txt" in Autonomous System "as4" must have "as4" as a part of the path name. The Autonomous System can be named anywhere in the path, but it must be named once.

As of now, FileLocation is unrigid in that its only requirement for a file is that its path includes the Autonomous System once. In the future, extending FileLocation will likely involve more shaping of the structure of paths into a more standardized form as opposed to the current malleable form of paths. Preserving this growing room now means we can mature the FileLocation object as the precise addressing framework is more standardized

3.3.4 DistanceAwayRule, RuleOperator

In order to be able to distribute files, there needs to be some definition or directions of how to distribute the files. We need a criterion or requirement for the distribution action. One model could be distributing files by magnitude. For example, the rule could be that four copies of a file must exist throughout the network at any given time, and the files can exist anywhere in the network. Another model could be that of specific, designated replication. An example would be something along the lines of dictating that file must exist specifically in Autonomous Systems “as7” and “as10.”

There are many possible ways to distribute files, and our system has implemented a rule for distribution based on distance. In our case, distance is a measure in number of AS hops. Our distance rule dictates that files are distributed according to some magnitude of AS hops away from any file’s originating AS. If a file exists in as5, an example of the distance rule would be declaring that the file must be placed in ASs that are four hops away from as5. Our system would then figure out which ASs are four hops away from as5 and would then copy the file into some number of ASs at the specified distance away.

Our system chose AS hop distance as a helpful rule for distributing files because we believe it best promotes the end goal of having files available and reachable in the case of failures. In the current Internet, we argue that it is difficult to diagnose problems because there are too many limited domains of network management that are unable to collaborate when a problem spans more

than one domain. Often these domains are real world Autonomous Systems who self-manage themselves. Currently, ASs will manage themselves based on what limited information they themselves happen to own, and if they do not have enough information or if helpful information may happen to exist in another AS, it is simply unfortunate and the problem is left unresolved. In the future Internet architecture, the goal is to make information a shared resource and to share it intelligently by placing the information in places that will be available even when another location is inaccessible. ASs would be able to use the needed information to figure out failures because the information will be at reachable locales. By creating a distance rule that distributes files across ASs, we ensure that more than one realm owns a file and can provide access. This is helpful because in the real world, problems in an AS such as an Internet Service Provider will sometimes result in that ISP temporarily unable to provide services. In those cases, it is favorable and advantageous if another AS, which could be another ISP, is able to provide the service of access to the file of needed information.

A distance rule based on AS hops entails many variations, which allows for more flexibility, adjustability, and robustness. The first degree of freedom comes in being able to specify the magnitude of the number of hops one desires. The second degree of freedom comes from specifying the precision of distribution. To be clearer, the system allows one to specify the placement of files at some *exact* number of hops away from an AS, where 'exact' is the equivalent of the '=' operator. The degree of freedom comes in when we realize that we can vary the operator. Perhaps we do not want files placed at ASs *equal to* some x number of hops away, where x is some non-negative integer. The intent might instead be to place a file at ASs *less than* some x number of hops away, which would be the mathematical '<' operator. We have extended our system to account for all these variations, allowing for the $<$ (less than), $\leq$ (less than or equal to), $>$ (greater than), $\geq$ (greater than or equal to) operators in addition to the '=' (equal to) operator when it comes to measuring AS hops.

Introducing additional operators also introduces technical details of how those operators should function. When the rule says that file placement should be at an AS fewer than *x* hops, there exists ambiguity because there can be multiple ASs that are fewer than *x* hops from the original AS. The rule might mean that the file should go in every single AS fewer than *x* hops. Or the rule might accept one random AS fewer than *x* hops. Or perhaps it will tolerate some other measurement of which AS to choose. If we look our system's model of the network, the network is composed of a set of linear Autonomous System paths (the AutonomousSystemPath object). Treating the network as a set of paths, our system implements the < operator as picking one AS along each path that sits at a position fewer than the specified number of hops away from the original AS. To illustrate with an example, we have the original file exists in as5. The paths starting at as5 are AutonomousSystemPath_1 (as5, as2, as4, as6, as7), AutonomousSystemPath_2 (as5, as10, as12, as1), and AutonomousSystemPath_3 (as5, as9, as8). The rule says that we must copy the file in as5 to all ASs fewer than three hops from as5. All qualifying ASs in AutonomousSystemPath_1 not including as5 itself would be as2 and as4. For AutonomousSystemPath_2, as10 and as12 would quality, and for AutonomousSystemPath_3, as9 and as8. Our system does not replicate the file into every single AS along a path that fulfills the rule's requirement. Instead, for each path, the system picks one AS that conforms to the rule legally and replicates the file there. In our example, this means picking one AS each from AutonomousSystemPaths 1 through 3. This implementation for our system may change in the future to copy a file to every single qualifying AS, but our system currently only picks one AS because we do not see the need for relentless duplication all along a path. We argue that duplicating all along a path is, while still correct, overly redundant and results in unnecessarily large numbers of copies of files to fulfill a simple rule. This same method of picking ASs is utilized when picking ASs for the greater than, less than or equal to, and greater than equal to operators. For the 'greater than' and the 'greater than or equal to' operators, we pick ASs that are one hop greater

than the minimum number of hops away, and we pick a single AS along each path originating from the file's AS. The 'less than or equal to' operator is fulfilled by executing it the same way as the 'less than' operator.

To implement the AS hop distance rule in our system, we have the DistanceAwayRule component. It contains two attributes: the magnitude of the number of hops, an integer, and the operator, which is an instance of the RuleOperator enumeration. The RuleOperator component enumerates the five different operators of $<$, $\leq$, $>$, $\geq$, and $=$. The DistanceAwayRule objects can be associated with different files in our system, and the file's rule tells the system how that particular file should be distributed in the network. As of now, our system currently associates a single file with a single DistanceAwayRule. In the future, as different types of rules enter the system, the system will extend to accommodate these new rules as well as expand to let files follow multiple rules at the same time. Also, in the near future, we plan on including a qualifier to the DistanceAwayRule that declares how many total copies of a file should exist within the network. The semantics would be along the lines of "wanting between 5 to 8 copies of a file in a network" or having "3 copies." More complicated extension would include wanting m copies at ASs that are n hops away from each other, but we leave that to further exploration in the future.

A DistanceAwayRule never exists on its own unless it is associated with a file. Files do not necessarily have to follow a rule. If there is no rule for a file, the file can be replicated ad hoc if desired. A rule is only associated with the original file. If the original file is deleted, the rule does not carry over to the copies of the file and is instead removed. The copies of the file continue to exist, and one of the copies is selected randomly to become the original.

The system is responsible for enforcing a file's DistanceAwayRule, whether the rule is associated with the file upon the file's creation or if the rule is added to the file later on. A file's DistanceAwayRule can be changed to a new rule, and the

system will take steps to enforce the new rule. Rule enforcement is executed as soon as a rule is associated with a file. We assume that an entity outside of our system such as a human user is the creator of a rule for a particular file. This entity is knowledgeable about the nature of the file and the desired distribution for the file. It is not the system's role to perform artificial intelligence analysis on the contents and types of files and to decide which rules would best govern how the files should be distributed. This decision-making process for what rules to create and which rule a file should have is external to our system.

We conscientiously point out that a rule may become untrue in its lifetime. There are two possible causes for this. One, the topology of the network may change such that the rule no longer holds true. Two, a copy may go away permanently due to failure in the network, leaving the rule enforced incorrectly. This raises the question of when to check for if a rule has become unenforced, how often to check, and when to execute the proper procedures to reinforce the rule. For example, if an important file must have five copies within the network at all times, we must provide some mechanism to ensure that the file follows the rule. In this particular case, a modification of our system to be able to transfer rules from an original to a copy file will also be needed. This problem of rule re-enforcement after creation is an important problem space that will require further study and expansion on our existing system.

3.3.5 FileUIDToLocationsAndRule

In our system, a file has a FileUniqueID to identify the file. The file also has a starting FileLocation, which is just a path, and once it is replicated, it will have multiple FileLocations, which are more paths to different locations. The duplicate sites for the original file are dictated by a DistanceAwayRule. In order to organize all this information, we have the FileUIDToLocationsAndRule system component.

FileUIDToLocationsAndRule is the object that represents the abstraction of a table where each table entry holds information for a different file. Abstractly, the columns of the table are FileUniqueID, Primary FileLocation, Copy FileLocations, and DistanceAwayRule. Each entry in the table stores the relevant information for a unique file, starting with its FileUniqueID. Primary FileLocation field stores the file's original path location. The Copy FileLocations field stores a list of the paths of the duplicates of the original file. The DistanceAwayRule field lists what rule is associated with the file. The list of paths in the Copy FileLocations field should satisfy the rule for distribution as specified by the file's DistanceAwayRule.

All unique files in the network have an entry in the table, which means the size of the table tells us how many unique files exist in the network. The number of unique files is not the same as the total number of files in the system. Counting unique files means we are not counting duplicates of files, which means the number of unique files in the network should always be less than or equal to the total number of files in the network.

The table must hold an entry for all the files in the network, and we assume that the network is constantly creating new pieces of information and new files, with little of it deleted. We believe this tendency of the network to skew more toward creating information more often than it deletes information is an accurate depiction of the real Internet since, as everyone well knows, once anything arrives onto the Internet, we accept that it will exist forever somewhere in the network, impossible to ever fully eradicate or remove. As such, FileUIDToLocationsAndRule is a constantly growing table where an entry in the table is created for every new file created in the network and very few entries are ever deleted. In fact, if there is a deletion, it is more for the reason that the file is no longer necessary than it is because the file is truly gone from the network. Our system regulates and informs the system's users where authoritative copies of the file reside in the network, but there will always be users who download the

file privately to store it and use it somewhere in their own private network or machine. Since new files are added to the system constantly with few deletions, the data structure supporting the table must be able to handle insertions efficiently while deletions are not as much of a concern.

The primary usage for the FileUIDToLocationsAndRule object is to help users of the system find the list of possible locations where a file may reside. The table lists its entries by FileUniqueID, so if a user wants to know the possible locations of a specific file, they take the file's FileUniqueID, look it up in the table, and can get back a list of locations where the file resides from the Copy FileLocations field of the table. We believe that in addition to inserting entries into the table, a table lookup by FileUniqueID will be the most common function performed on the table. The main goal of our system is to provide alternative locations for a file when an entity trying to find the file cannot find it at a particular location. It is the FileUIDToLocationsAndRule object which performs the service of figuring out what these alternative locations are for a file and to provide this list to the entity seeking the file. To accomplish its function of figuring out the list of possible locations for a file, the FileUIDToLocationsAndRule must look up the file by its FileUniqueID. Since lookups are core to the functioning of FileUIDToLocationsAndRule table, lookups in the table need to be fast and efficient, just like insertions into the table.

Because the table needs fast lookups and insertions, we implement it using a binary tree, which is desirable for being a lightweight data structure that has a good running time of log(n) on average for operations such as a lookup or an insertion. It is important that these two functions have good running times because we use them so often, and we do not want our system bogged down by slow running times. The use of a binary search tree was a design decision. There are other trees that give even better running times. The binary search tree yields an average log(n) running time for operations, but in the worst-case scenario, these operations take up to O(n) running time to complete. This happens when

the data in the nodes on which the binary search tree is sorted is not random enough. We key the nodes of the binary search tree underlying FileUIDToLocationsAndRule with a FileUniqueID and argue that the probability of the worse-case scenario is almost zero due to the fact that a FileUniqueID is created using the MD5 hash, which means its generation is as random as MD5 and will result in good binary search tree performance. The binary search tree is an advantageous design decision because we can rely on it to perform efficiently. And it is still a lightweight data structure so we are able to avoid the cumbersomeness of using more complicated data structures in order to achieve the negligible gains in performance.

By using a binary search tree as the underlying implementation of the abstract table that is the FileUIDToLocationsAndRule object, each node of the binary search tree becomes an entry in the table. Each node stores all the information for finding a file and its copies along with the file's rules. The node is keyed on a file's FileUniqueID, and the node itself stores a list of all the paths that are locations for where to find the file. It also stores information about which was the original AS the file belongs to as well as which ASs the file was copied into. Finally, the node stores the DistanceAwayRule that the file follows for spreading its copies. A node keeps track of the original AS along with the ASs where copies reside separately from the list of all paths because this enables the system to be able to perform a quick check of whether or not a file's DistanceAwayRule is being followed by checking the list of the copy ASs without having to go through the paths and checking each final destination.

When the FileUIDToLocationsAndRule object is first initialized, it is empty and it is up to the system to populate it with the files from the network. The object has the method addNewUIDLocRuleEntry, which takes in a file's FileUniqueID, its original FileLocation, and a DistanceAwayRule for the file. This adds a new node into the tree for the file. The system then automatically distributes the file according to the DistanceAwayRule, and updates the file's

node appropriately to add in the list of copy locations. It is possible to add a file and its original location without a rule by using the method addUIDLocEntry. The system also allows for designated copying of a file into a specific AS, not according to any rule, by using the method addLocationForFile, which will place a copy of the file specified in the argument into the location, given as another argument to the method. Rules for a file can be added at a later time using the addRuleForFile method and removed at a later time using the removeRuleForFile method. A file's locations can also be deleted from precise spots using removeLocationForFile. Finally, the FileUIDToLocationsAndRule object has methods that allow users to query its contents, including a containsFile method to check if a file exists in the system. Methods for retrieving a file's list of locations, its original AS, the list of ASs where copies exist, and the rule for a file are getLocationsForFile, getMainASForFile, getCopyASForFile, and getRuleForFile respectively.

FileUIDToLocationsAndRule will continuously update itself as the files in the network change or if new files are created. The system lies over AFS, and since the system sees a uniform file name space that looks like a tree of directories, in order to discover new files, our system uses an observer called DirectoryWatcher to watch for additions of new files at the different portions of the file name space that belong to different ASs. Thus when a new file in an AS is added, the system will be notified of it and can add it to the FileUIDToLocationsAndRule object appropriately

3.3.6 LocationsToFileUID

The FileUIDToLocationsAndRule system component is only useful for querying the information for a file once we know the file's FileUniqueID since the FileUIDToLocationsAndRule object is sorted based on FileUniqueID's. However, a file itself in the network is not tangibly tagged with this FileUniqueID. Users of our system navigate the network by navigating the file space of AFS and can find files anywhere in that file space. For example, a user of our system might try to

navigate to the path /afs/network/as5/public/routerdata/router515data.txt because they want to find the router515.txt file. Our system was built to solve the problem of the case where the user cannot reach the file /afs/network/as5/public/routerdata/router515data.txt either because of some routing error or that part of the network suffering from some problem. The user knows what file they want because they know the path, but for the FileUIDToLocationsAndRule object, a user must know the file's FileUniqueID in order to query for the list of other possible locations for a file. The missing link is how a user can figure out the FileUniqueID for a file just from its path.

LocationsToFileUID is the object that represents the table that provides the missing connection. A user can look up a path in the LocationsToFileUID table to find its corresponding FileUniqueID. The user can then take the FileUniqueID and query the FileUIDToLocationsAndRule table to retrieve the list of all locations for the file. Then the user can use that list to search for the file at a different location since the location the user originally tried to get to is down. In our example, the user originally tried looking for the file at /afs/network/as5/public/routerdata/router515data.txt. Since the user could not access the file there, the user goes to the LocationsToFileUID table, which says that for path /afs/network/as5/public/routerdata/router515data.txt, the FileUniqueID is "243cd5eab." The user then takes this FileUniqueID and looks through the FileUIDToLocationsAndRule table. For 243cd5eab, FileUIDToLocationsAndRule returns a list of paths that includes /afs/network/as9/others/as5/public/routerdata/router515data.txt, /afs/network/as10/others/as5/public/routerdata/router515data.txt, and /afs/network/as11/others/as5/public/routerdata/router515data.txt. The user can then try to find the file at these other locations.

The nature of the LocationsToFileUID table is that it undergoes many insertions and many deletions as well as many lookups. If we think about our system, when a file is newly created in the network, it creates an entry in the

FileUIDToLocationsAndRule object. The new file also creates an entry in the LocationsToFileUID object that maps the new file's location path to its FileUniqueID. If the new file has an associated rule, this file is copied into appropriate locations conforming to the rule. The LocationsToFileUID table then has multiple more insertions that add all these new copied locations that all map to the same FileUniqueID. When a file and all its copies are removed from the system, the FileUIDToLocationsAndRule object simply removes that one table entry with that particular FileUniqueID. The LocationsToFileUID table must delete multiple entries to remove all the paths that have the same FileUniqueID. And because users of the system will be interacting with files by using their path names, the LocationsToFileUID table is used often for lookups for a file in order to find the file's FileUniqueID. This increased frequency of insertions, deletions, and lookups means the LocationsToFileUID table must have an implementation that performs well for all these actions. Whereas the FileUIDToLocationsAndRule object only had to make sure insertions and lookups were optimized since deletions were rare, the LocationsToFileUID has to make sure all three run quickly and efficiently or risk slowing the entire system down.

We implement the LocationsToFileUID table using a red-black tree because it promises worst-case performance of O(log *n*) for the search, insert, and delete functions and often performs better than log n. The LocationsToFileUID object is modified much more often than the FileUIDToLocationsAndRule object, by using a red-black tree, we ensure that its worse running time of log n will always be commensurate to FileUIDToLocationsAndRule's average running time of log n so that our whole system runs on log n.

We can find out the total number of files, including copies, in the network by getting the size of the LocationsToFileUID table since copies get different entries. The object provides the method for looking up a FileUniqueID for a given path using getUIDForPath. The object adds and removes entries using addLocationFileUID and removeLocation.

3.4 **Summary**

We have described all the different important components of our file location and distribution system along with their purposes and functions. We now look at how all the pieces integrate and function together by looking in detail at our working implementation of the system and its distribution of files across a real network of computers in the DETER Network Security Testbed.

4 Implementation of the System at Work

4.1 Set-up

In this chapter, we describe the steps the file location and distribution system goes through as it takes a file in a network, adds it to the system, and then distributes it. The chapter goes into detail with respect to the workings of the system using a real implementation of the system on a network of machines hosted by DETER Network Security Testbed. The chapter first describes the set-up of the machines, including what operating systems the machines are using and how they are connected, and what the topology of the experimental network looks like. The chapter goes on to describe how our system runs over this topology of machines and the processing of files found in the network using several examples. These files are generated by various machines in the network. The overall simulation illustrates the proper functioning of our system. It demonstrates that files can be distributed correctly according to predefined rules such that access to a file can be reached by accessing its copies in the network.

4.1.1 Topology

To demonstrate our system in action, we needed to first create a small computer network that would serve as an example of a real world network. The topology of the network we created using machines in the DETER Network Security Testbed is diagrammed in Figure 1. The triangles in the figure are the individual Autonomous Systems, and they are numbered. There are sixteen total autonomous systems in our experimental network. Each autonomous system consists of three machines, and each machine has its own unique name within the autonomous system. The machines are the vertices of the triangles, and the edges of the triangles represent the links between the machines. The lines connecting the different triangles represent links between different autonomous systems. In our experimental set-up, links between ASs are bidirectional. If AS6 can send packets to AS7, then AS7 can send packets to AS6. This may not always

be true. For our experimental network, we make the simplification that there is only one link between two connected Autonomous Systems. This differs from the real world where there are thousands of links between two Autonomous Systems.

We set up the ASPathDatabase object so that it stores all the possible AS paths in our network. The AutonomousSystem objects are created by the AutonomousSystemGenerator. They're combined into AutonomousSystemPath objects which are then added to the ASPathDatabase object.

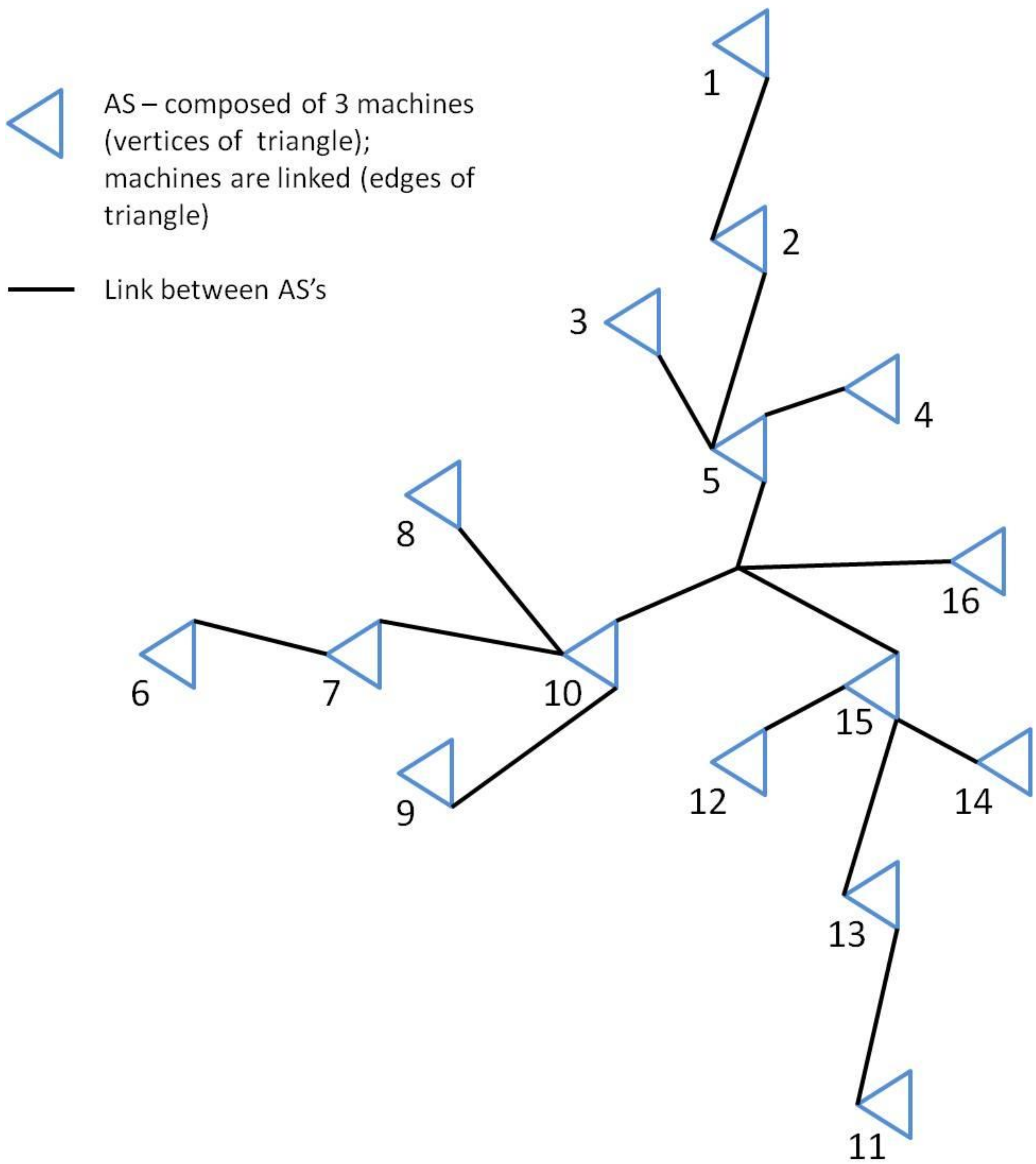

Figure 1 Network topology of sixteen autonomous systems, each composed of three machines

4.1.2 File Space Structure of AFS

All the machines in our experimental topology run Ubuntu Hardy Heron as their operating system and all run the Ubuntu version of OpenAFS, the open-source Andrew File System. Each machine is both a file server and client of AFS.

AFS is a distributed file system that presents a uniform-looking file space to users in the form of a single tree of directories with the root at /afs. Having a uniform file space means that all clients of the AFS system will have a directory in their own file system called /afs, and this directory and all of its contents will appear identical to every single client machine. AFS is a distributed file system, which means files and directories reside physically in memory on only select file servers within the AFS system. The machines that are clients can see the entire /afs directory and all its children, but these directories and files do not necessarily exist in the clients' memory unless the client is also a server for those directories and files. It is important to remember that machines have their own built-in hard drive and also their own file system outside of the AFS file system. Only the directories and files within the /afs directory are part of AFS. Machines that are clients of the AFS file system can see those files that sit on another machine's hard drive as long as that file is somewhere in the /afs directory tree. Other directories on the machine such as /bin or /tmp all reside within the machine's own file system and memory. A machine that is a server only acts as a server for the files that exist in its memory within the /afs directory tree.

AFS is a distributed system because the different directories in its file space sit on different machines. For example, imagine an AFS setup where there are a total of three file servers, and the structure of the file system looks like Figure 2. Recall that the afs directory is the root of the AFS file system.

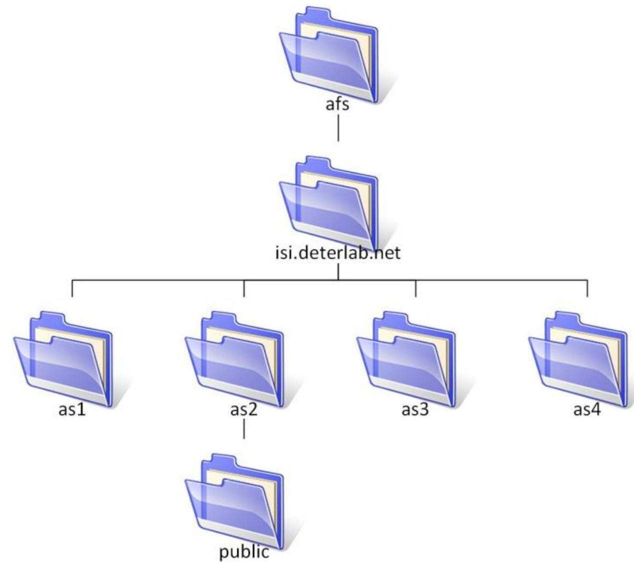

Figure 2 Example AFS directory tree

A possible set-up for the three file servers would be to have the first file server have in its memory the afs root directory along, its child directory isi.deterlab.net, and the as1 directory. The second file server has directories as2 and public in memory. The third server has directories as3 and as4 in memory. AFS clients are able to access a directory without needing to know which exact server a directory sits on.

In our implemented experimental network of 48 machines divided into 16 autonomous systems, every machine is both an AFS file server and client. We organize the AFS file space so that different subtrees within the entire file tree reside in different Autonomous Systems. Each subtree that resides within an autonomous system has a root directory whose name is that AS's name. Our organization of the file space looks like Figure 3.

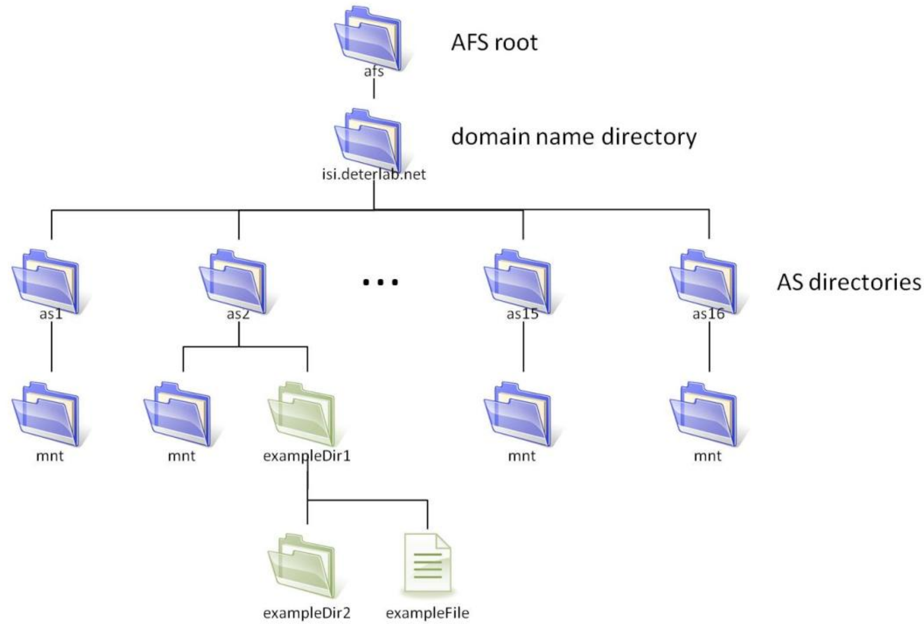

Figure 3 Structure of AFS file space for the experimental network

The root of the AFS file system is the afs directory. Within the afs directory is a single child directory, the domain name directory. Our domain name directory is isi.deterlab.net since that is the domain name for the DETER testbed. The afs directory and the domain name directory is the standardized set-up for all AFS networks that exist in the world. Within the domain name directory is where we have a directory for each AS in our experimental network. Since we have 16 total ASs, there are 16 directories, as1 to as16. The names of the directories correspond to the names of the ASs, and we call these directories AS directories. They are the only directories in the domain name directory.

The AS directories and all their contents, including more directories and files, reside physically within that AS. This means that the AS directory and all of its children directory and files are hosted by some machine within that AS. It can be any machine, as long as that machine belongs to the AS. In Figure 3, this would mean that all of the directories and files within directory as2 along with the as2 directory itself must be physically in memory somewhere in Autonomous System 2.

Within each AS directory is a mnt directory. This directory is where only copies of files from other Autonomous Systems reside. Files that are native to an AS may not reside in that AS's mnt directory. The mnt directory is strictly for foreign files. This mnt directory helps to clear up confusion and removes the problem of file aliasing when copying files across Autonomous Systems. For instance, imagine a file created in Autonomous System 4 that has the name data2010Aug08.txt. Imagine that there exists another, different file in Autonomous System 2 that also has the name data2010Aug08.txt. Confusion may arise when we try to distribute AS 2's file into AS 4. Trying to rename files so that every file has a unique name would be an impractical and unreasonable approach. Instead, we copy AS 2's file into AS 4's mnt directory such that the path for AS 2's file becomes a subpath in AS 4's mnt directory. To elucidate, the path for data2010Aug08.txt in AS 2 (excluding the default /afs/isi.deterlab.net parent directories) might be /as2/public/router4147/august/data2010Aug08.txt . We copy this file and its entire path starting at the AS directory (in this case as2 is the AS directory) into the mnt directory of AS 4. The path for the file in AS 4 becomes /as4/mnt/as2/public/router4147/august/data2010Aug08.txt . Note that we know this file resides in AS 4 and not AS 2 because the AS directory is as4. We keep the same path for files from other autonomous systems because it provides a way to have unique paths for files without worrying about conflicts. AS 4's file data2010Aug08.txt still resides somewhere in AS 4 *not* in the mnt directory, and we are therefore satisfied that there are no collisions. Figure 4 shows the file copied from AS 2 to AS 4.

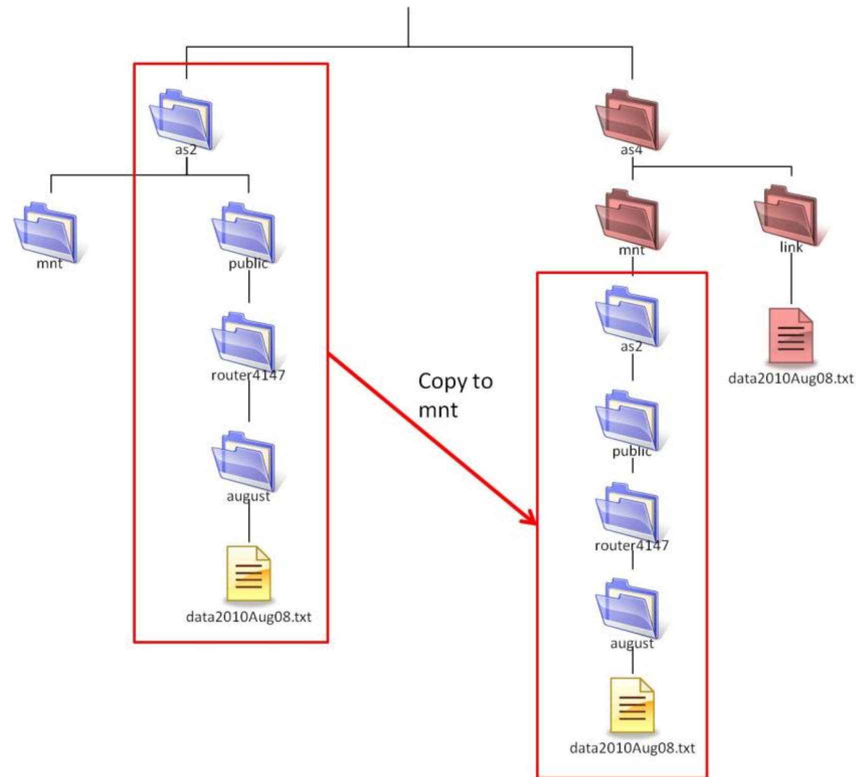

Figure 4 Example of copying files across autonomous systems

4.2 Capabilities of the System Through Examples

Starting with the above network topology for our machines with the file space set up as described, we now walk through examples of the system taking files from one Autonomous System and copying it to other Autonomous Systems. We will cover five cases as follows:

1. Copying a file from one autonomous system to an autonomous system less than a certain distance away.
2. Copying a file from one autonomous system to an autonomous system greater than a certain distance away.
3. Creating the file and its copies. Then removing the rule for distribution, which results in deletion of the copies.
4. Copying a file from its original autonomous system to other autonomous systems and deleting the original along with its rule.

5. Copying a file from its original autonomous system to other locations, deleting the original, and imposing a new rule.
6. Copy a file to a designated AS without using a rule.

For each case, we detail the interaction between the system components that were described in the previous chapter. We show how they interact in order to accomplish each case.

4.2.1 Case 1: Copying files to an AS less than some distance away

To demonstrate our system's functionality, we start with the example of Autonomous System 6, AS6, wanting to distribute one of its files to ASs that are fewer than four hops away. If we look back at Figure 1, the qualifying ASs within the topology would be AS7, AS8, AS9, AS10, AS5, AS15, and AS16. Our currently implemented system chooses not to propagate the file to every single AS that qualifies and instead picks one qualifying AS at random. There is an option to set copying to all the ASs that qualify, but for the sake of showing the sequence of steps taken to copy a file from one AS to another, we show the copying process to just one other AS. One can then infer that the same exact steps are taken to copy the file to the other qualifying ASs.

Although each AS has three machines, for simplification, we choose a delegate machine for each AS. This means copying files from one AS to another is really copying the file from the delegate machine of the first AS to the delegate machine to the second AS. We believe this is a reasonable simplification that does not limit our system in practice in the real world because the delegate machine can be any machine in the AS. It does not matter that we choose only the delegate machine to copy to and from because an AS can change which machine is the delegate machine. Because the AS can use any machine as the delegate machine, our system is still copying files at the AS level, not at the machine level.

In our experiment, AS6 runs a shell script that collects ping data every 10 minutes and writes the data to the file pingData*X*.txt where *X* is a counter that

increments each time the script runs. Thus the first time the script starts up, the first file it writes is pingData0.txt . The next time the script runs and creates a file, it is pingData1.txt, then pingData2.txt and so on. The ping data files all reside in the /afs/isi.deterlab.net/as6/networkInfo/ping directory of the file space. Because /afs/isi.deterlab.net is the prefix for every single path in our system, from here on out, when we list paths, we remove the "/afs/isi.deterlab.net" portion. Paths will start with the AS directories instead, as in /as6, but it is important to remember that all paths are still prefixed with /afs/isi.deterlab.net .

Our system takes the newly generated pingData*X*.txt files and distributes them to a chosen AS that fulfills the rule of being fewer than four hops away. As of now, the AS where the copy resides is chosen at random from the pool of qualifying ASs. This means that the various pingData*X*.txt files do not necessarily copy to the same AS.

In order to detect the creation of the new ping data files in AS6, we have a DirectoryWatcher object that watches the /as6/networkInfo/ping directory for new files. We create a DirectoryWatcher for any of the directories within AS6 where network management information is generated. When the DirectoryWatcher detects the creation or deletion of a file, it notifies our system. For this case, we are looking at the creation of ping data files.

When the first ping data file is created, pingData0.txt, the system's MD5Calculator component takes the file, runs an MD5 hash, and returns the string of the 128 bit hash number. In this example, the string is "de893e1e." This string is used to create a FileUniqueID for the pingData0.txt file. Figure 5 diagrams the creation of a FileUniqueID for a file pictorially.

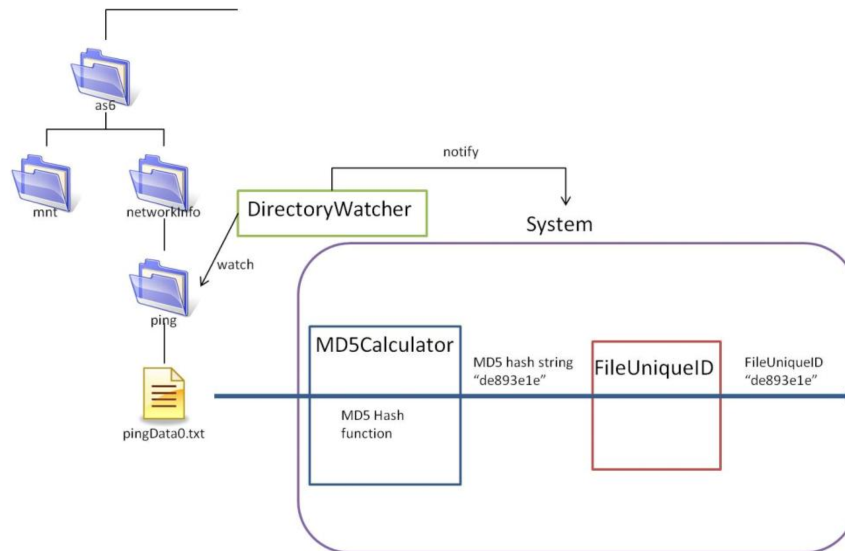

Figure 5 Creation of FileUniqueID for a new file in the network

The system knows beforehand that the file needs to be copied to Autonomous Systems fewer than four hops away, so we create the DistanceAwayRule with a magnitude of four and the RuleOperator object "LESS_THAN" (See Appendix A for the DistanceAwayRule constructor). Abstractly, this represents the rule " < 4." The FileUniqueID "de893e1e," the path "/as6/networkInfo/ping/pingData0.txt," and the DistanceAwayRule "< 4" are inserted into the FileUIDToLocationsAndRule object as shown in Figure 6. The entry under Copy ASs is a list of the Autonomous Systems where the copies of the file reside. Since the file has not been copied yet, the list is empty. The Paths column has the list of all paths where a file and all its copies reside. As of now there is only the path of the original file since the file has not been copied yet. Also shown in Figure 6 is the LocationsToFileUID object. We add the file's path and its FileUniqueID to this object. There is only one entry in the LocationsToFileUID object for the file, and it is for the path /as6/networkInfo/ping/pingData0.txt and the FileUniqueID "de893e1e."

FileUIDToLocationsAndRule

FileUniqueID	Paths	Main AS	Copy AS's	DistanceAwayRule
de893e1e	/as6/networkInfo/ping/pingData0.txt	as6		< 4

LocationsToFileUID

Path	FileUniqueID
/as6/networkInfo/ping/pingData0.txt	de893e1e

Figure 6 FileUIDToLocationsAndRule and LocationsToFileUID after notifying the system of the new pingData0.txt file, before the copying of the file has happened

Recall that the FileUIDToLocationsAndRule object is really a binary search tree, so each entry in the table is actually implemented as a binary tree node. Also recall that the LocationsToFileUID is implemented as a red-black tree so every entry in that table is actually a red-black tree node.

After initially populating the FileUIDToLocationsAndRule and the LocationsToFileUID objects, the system now has to implement the rule for the file. The system knows from the DistanceAwayRule's RuleOperator field that it is looking at a "less than" scenario. Because it is a less than scenario, the system only needs to find one Autonomous System that qualifies.

The system goes to the ASPathDatabase, and retrieves the set of all paths that start with AS6 in our topology. The ASPathDatabase looks like Figure 7 for AS6. The set of paths for AS6 are in no particular order.

Autonomous System	Autonomous System Paths	Chosen AS for <4 rule:
as6	(as6)	n/a
	(as6, as7)	as7
	(as6, as7, as10)	as10
	(as6, as7, as10, as8)	as8
	(as6, as7, as10, as9)	as9
	(as6, as7, as10, as16)	as16
	(as6, as7, as10, as5)	as5
	(as6, as7, as10, as5, as3)	as5
	(as6, as7, as10, as5, as4)	as5
	(as6, as7, as10, as5, as2)	as5
	(as6, as7, as10, as5, as2, as1)	as5
	(as6, as7, as10, as15)	as15
	(as6, as7, as10, as15, as12)	as15
	(as6, as7, as10, as15, as14)	as15
	(as6, as7, as10, as15, as13)	as15
	(as6, as7, as10, as15, as13, as11)	as15
...	...	

Figure 7 ASPathDatabase entry for AS6

The system randomly chooses one path in the set of paths that begin with AS6, and picks an AS along that path that is fewer than 4 hops away. The system will not pick an AS if it is the origin AS6 itself. The system picks an AS fewer than four hops away by first checking to see if the path has an AS three hops away from AS6. If the path is too short, e.g. its total length is three ASs long, the system picks the last AS in the path. This means the resulting chosen AS to duplicate the file to could be one hop, two hops, or three hops from AS6, depending on which path the system inspects first. Figure 7 lists the AS that would have been chosen to duplicate the file to depending on which path the system randomly chose. For this example, our system chose the AS to be AS15.

The system then uses the FileSystemManipulator object to create a copy of AS6's pingData0.txt in AS15's mnt directory following the procedure as described in the earlier section "**File Space Structure of AFS.**" Now AS15 has a copy of AS6's file, and the file space now looks like Figure 8.

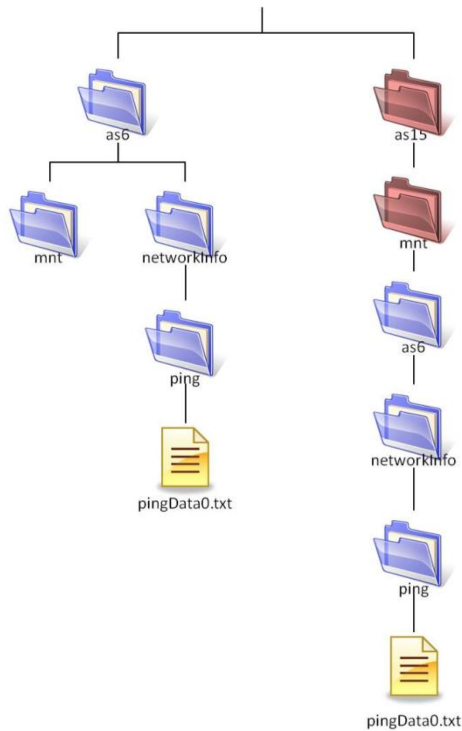

Figure 8 File space after pingData0.txt is copied from as6 to as15

When the FileSystemManipulator finishes copying the file, the system updates the LocationsToFileUID object so that it now has the new location listed for the file and its FileUniqueID. The LocationsToFileUID object now looks like Figure 9.

LocationsToFileUID

Path	FileUniqueID
/as6/networkInfo/ping/pingData0.txt	de893e1e
/as15/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e

Figure 9 LocationsToFileUID after pingData0.txt is copied to AS15 from AS6

After updating the LocationsToFileUID object, the FileUIDToLocationsAndRule object is updated to reflect the new path and the new ASs. The object now looks like Figure 10. This is always the order of updates.

The LocationsToFileUID is always updated before the FileUIDToLocationsAndRule object. We do this because we consider the FileUIDToLocationsAndRule object to be more authoritative. Thus, if somehow the system fails in the middle of changing the LocationsToFileUID object, we just propagate the information in the FileUIDToLocationsAndRule object back into the LocationsToFileUID object.

FileUIDToLocationsAndRule

FileUniqueID	Paths	Main AS	Copy AS's	DistanceAwayRule
de893e1e	/as6/networkInfo/ping/pingData0.txt; /as15/mnt/as6/networkInfo/ping/pingData0.txt	as6	as15	< 4

Figure 10 FileUIDToLocationsAndRule after pingData0.txt is copied to AS15 from AS6

Figure 11 graphically depicts the copying of pingData0.txt, showing the starting file space on the left, and the resulting file space on the right.

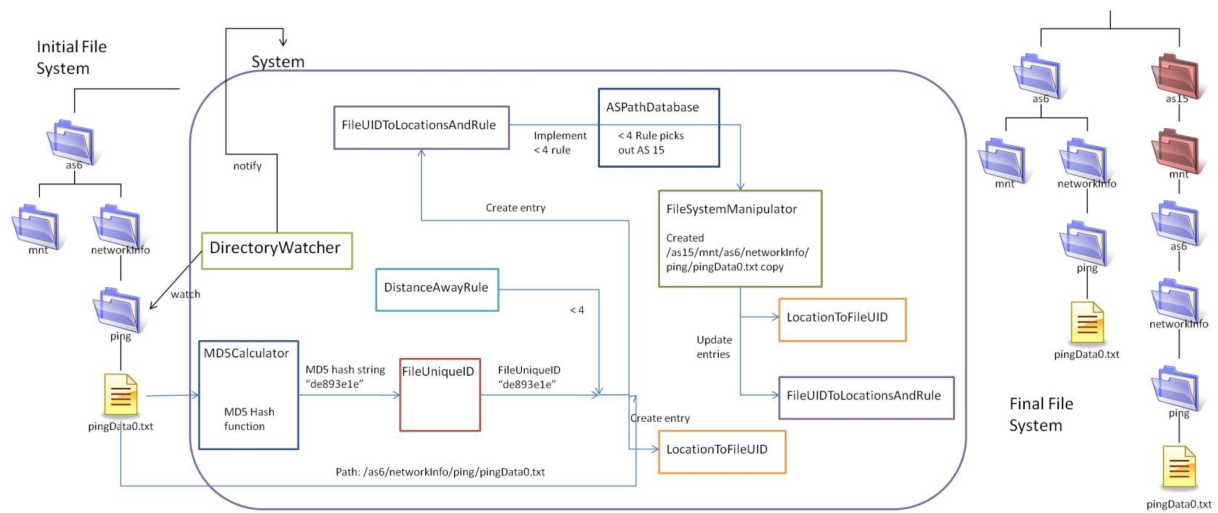

Figure 11 Schematic of pingData0.txt copied from AS6 to AS15

The next file that is created is pingData1.txt in the /as6/networkInfo/ping directory. Again, the DirectoryWatcher notifies our system of the new file, create a FileUniqueID “0dba5210c,” and the same steps are taken to duplicate it at a distance fewer than four AS hops away. This time, the AS chosen might be AS10 instead of AS15. The resulting file space would look like Figure 12.

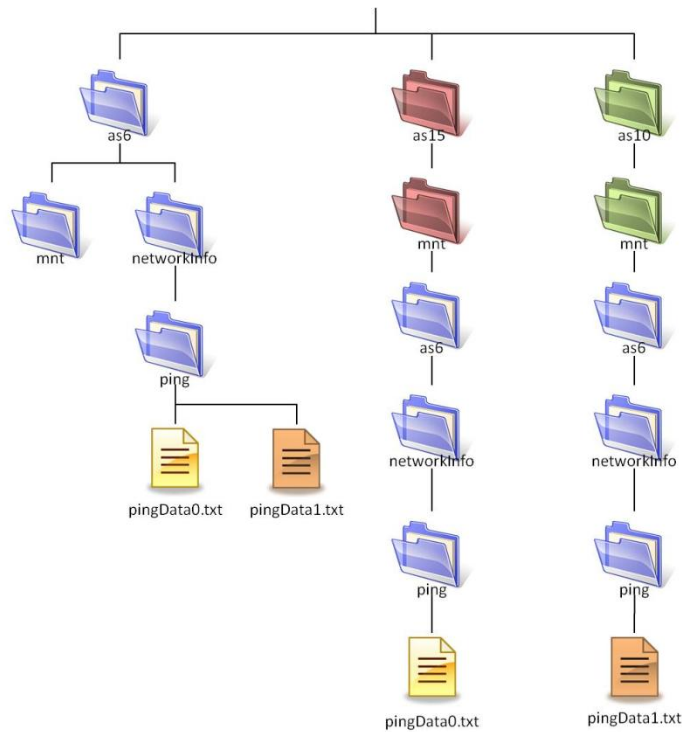

Figure 12 File space after pingData1.txt is copied to AS10 from AS6

The FileUIDToLocationsAndRule object and LocationsToFileUID object would look like Figure 13.

FileUIDToLocationsAndRule

FileUniqueID	Paths	Main AS	Copy AS's	DistanceAwayRule
de893e1e	/as6/networkInfo/ping/pingData0.txt; /as15/mnt/as6/networkInfo/ping/pingData0.txt;	as6	as15	< 4
0dba5210c	/as6/networkInfo/ping/pingData1.txt; /as10/mnt/as6/networkInfo/ping/pingData1.txt	as6	as10	< 4

LocationsToFileUID

Path	FileUniqueID
/as6/networkInfo/ping/pingData0.txt	de893e1e
/as15/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as6/networkInfo/ping/pingData1.txt	0dba5210c
/as10/mnt/as6/networkInfo/ping/pingData1.txt	0dba5210c

Figure 13 The tables after pingData1.txt is distributed with a < 4 DistanceAwayRule

4.2.2 Case 2: Copying files to an AS greater than some distance away

Our next example is to illustrate the file location and distribution system for the same /as6/networkInfo/ping/pingData0.txt file in Case 1 with a greater than rule as opposed to Case 1's less than rule. The greater than rule requires that a file must exist in a location greater than a certain number of AS hops away as opposed to less than. These two examples of using a less than DistanceAwayRule and a greater than DistanceAwayRule rule are umbrella rules as the fulfilling of these two rules also enables us to fulfill the "less than or equal to" rule, the "greater than or equal to" rule, and the "equal to" rule.

The copying process for the "greater than" DistanceAwayRule is the same as the mechanism for the "less than" DistanceAwayRule. The resulting file system will look like how we expect: the copied files will exist in the copy AS's mnt folders. The primary difference between the less than and greater than cases is the number of ASs the file is copied into. Whereas for the less than case, we copy the file into just one qualifying AS, for the greater than case, we copy the files into all ASs that are one hop greater than the indicated distance in the DistanceAwayRule. We copied into only one AS for the "less than" rule for simplification purposes. In reality, the copy of the file would most likely be copied

multiple times. The quandary that is not quite resolved and needs additional investigation is whether or not copying into every qualifying AS for a rule is excessive. We choose not to copy many times in the “less than” case to show its workability to achieve a rule. At the same time, here in the “greater than” case, we copy the file into more ASs to show that having many more copies is also a feasible option. As an example of our implementation of the “greater than” DistanceAwayRule, for a file says that the file needs to be copied at AS locations greater than six hops away, our system copies the file to all ASs that are seven hops away. We do not copy the files into ASs that are any farther away; we simply create enough copies to meet the basic requirement.

For this example, we use the same pingData0.txt file in AS6, and the rule is now that we want to copy the file into all ASs greater than three hops away. Looking at our AS topology in Figure 1, the ASs that are greater than three hops away from AS6 are AS1, AS2, AS3, AS4, AS11, AS12, AS13, and AS14. Our system only picks out the ASs that are four hops away from AS6 to copy pingData0.txt to. The ASs that are exactly four hops away are AS2, AS3, AS4, AS12, AS13, and AS14. We do not copy the file to ASs any farther away so AS1 and AS11 are not chosen.

Other than how many ASs pingData0.txt gets copied to, the rest of the process of the system is very close to Case 1. As a quick summary, similarly to Case 1, when the DirectoryWatcher detects the creation of pingData0.txt, it notifies the system, which creates a FileUniqueID for the file and then creates entries in the FileUIDToLocationsAndRule and LocationsToFileUID objects using this FileUniqueID, the path of the file, and the DistanceAwayRule “> 3.” The resulting objects look identical to Figure 6, except the DistanceAwayRule entry is “> 3” instead of “< 4.” The main difference comes in implementing the rule. When the system retrieves the list of paths that start with AS6 in order to figure out the qualifying ASs, as opposed to just picking one AS as it would have with a “less than” rule, the system picks out all the ASs that are exactly four

hops away from AS6. These ASs are AS2, AS3, AS4, AS12, AS13, and AS14. Again, the FileSystemManipulator object executes the physical copying process from AS6 to each of those ASs. The LocationsToFileUID object is updated each time a copy is made, resulting in six new entries. After all the copies are made, then the FileUIDToLocationsAndRule object is updated with all six new locations at once. The resulting file space along with the final LocationsToFileUID and FileUIDToLocationsAndRule objects after copying the file are shown in Figure 14 and Figure 15.

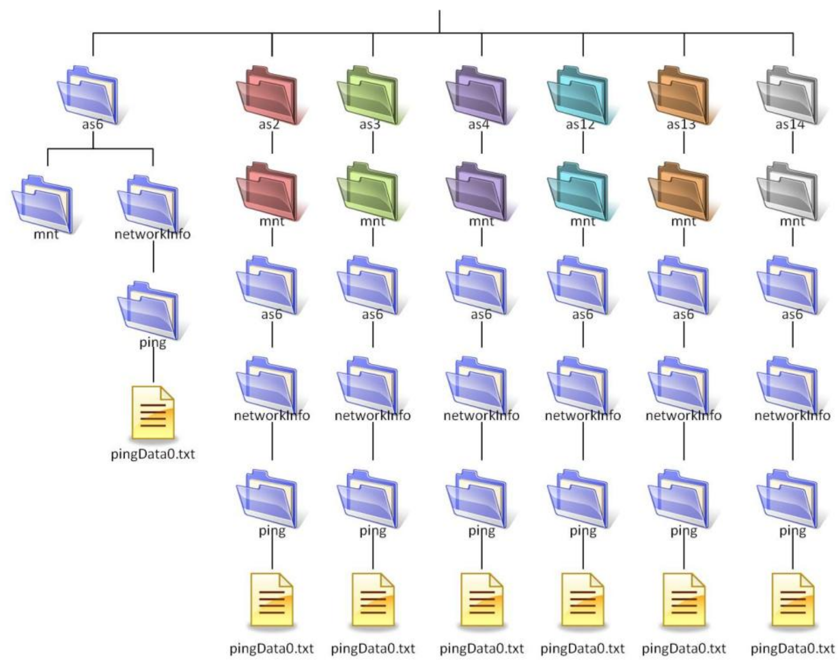

Figure 14 File space after pingData0.txt has been copied using the > 3 AS hops away rule

FileUIDToLocationsAndRule

FileUniqueID	Paths	Main AS	Copy AS's	DistanceAwayRule
de893e1e	/as6/networkInfo/ping/pingData0.txt; /as2/mnt/as6/networkInfo/ping/pingData0.txt; /as3/mnt/as6/networkInfo/ping/pingData0.txt; /as4/mnt/as6/networkInfo/ping/pingData0.txt; /as12/mnt/as6/networkInfo/ping/pingData0.txt; /as13/mnt/as6/networkInfo/ping/pingData0.txt; /as14/mnt/as6/networkInfo/ping/pingData0.txt	as6	as2, as3, as4, as12, as13, as14	> 3

LocationsToFileUID

Path	FileUniqueID
/as6/networkInfo/ping/pingData0.txt	de893e1e
/as2/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as3/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as4/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as12/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as13/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as14/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e

Figure 15 Tables after pingData0.txt has been copied using the > 3 AS hops away rule

For the file pingData1.txt with the same "> 3" rule, pingData1.txt would exist in all the same ASs as pingData0.txt, in the same directory. This is because the system no longer picks one AS that follows the rule at random as in the less than case. The system picks out all the same ASs for the same rule because it always picks out all the ASs that are exactly four hops away from AS6. The resulting file space looks identical to Figure 14 but with pingData1.txt as a sibling file to all instances of pingData0.txt. The resulting FileUIDToLocationsAndRule and LocationsToFileUID objects look like Figure 16 and Figure 17 respectively. Notice that the pingData1.txt file is copied to all the same ASs as the pingData0.txt file.

FileUIDToLocationsAndRule

FileUniqueID	Paths	Main AS	Copy AS's	DistanceAwayRule
de893e1e	/as6/networkInfo/ping/pingData0.txt; /as2/mnt/as6/networkInfo/ping/pingData0.txt; /as3/mnt/as6/networkInfo/ping/pingData0.txt; /as4/mnt/as6/networkInfo/ping/pingData0.txt; /as12/mnt/as6/networkInfo/ping/pingData0.txt; /as13/mnt/as6/networkInfo/ping/pingData0.txt; /as14/mnt/as6/networkInfo/ping/pingData0.txt	as6	as2, as3, as4, as12, as13, as14	> 3
0dba5210c	/as6/networkInfo/ping/pingData1.txt; /as2/mnt/as6/networkInfo/ping/pingData1.txt; /as3/mnt/as6/networkInfo/ping/pingData1.txt; /as4/mnt/as6/networkInfo/ping/pingData1.txt; /as12/mnt/as6/networkInfo/ping/pingData1.txt; /as13/mnt/as6/networkInfo/ping/pingData1.txt; /as14/mnt/as6/networkInfo/ping/pingData1.txt	as6	as2, as3, as4, as12, as13, as14	> 3

Figure 16 FileUIDToLocationsAndRule object after copying pingData1.txt to ASs > 3 hops away from AS6

LocationsToFileUID

Path	FileUniqueID
/as6/networkInfo/ping/pingData0.txt	de893e1e
/as2/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as3/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as4/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as12/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as13/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as14/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as6/networkInfo/ping/pingData1.txt	0dba5210c
/as2/mnt/as6/networkingInfo/ping/pingData1.txt	0dba5210c
/as3/mnt/as6/networkingInfo/ping/pingData1.txt	0dba5210c
/as4/mnt/as6/networkingInfo/ping/pingData1.txt	0dba5210c
/as12/mnt/as6/networkingInfo/ping/pingData1.txt	0dba5210c
/as13/mnt/as6/networkingInfo/ping/pingData1.txt	0dba5210c
/as14/mnt/as6/networkingInfo/ping/pingData1.txt	0dba5210c

Figure 17 LocationsToFileUID object after copying pingData1.txt to ASs > 3 hops away from AS6

4.2.3 Case 3: Removing the rule after copies are distributed

Cases 1 and 2 cover the copying of files according to their predetermined rule. The steps for copying and distributing are the same, regardless of whether or not the rule is added to a file immediately or at a later time.

Sometimes a user of our system will decide that they no longer want the file to be distributed according to the originally assigned rule and will want to

eliminate the copies. This is done by using the removeRuleForFile method of the FileUIDToLocationsAndRule object. When a rule is removed from a file this way, the system takes this as a desire to reset the file back to its original state with no copies. The system does this by first retrieving the list of all possible paths for the file from the FileUIDToLocationsAndRule object, which would be the Paths column in Figure 16. It also retrieves the set of ASs that hold copies of the file, which is the Copy ASs column in Figure 16. The paths are checked one at a time to see if they are paths within ASs that are in the CopyAS's set. If the path is a path within one of the Copy ASs, the FileSystemManipulator object executes the physical deletion of the file at that path from memory. We then remove the path from the LocationsToFileUID object, and remove the path from the list of possible paths for the file in the FileUIDToLocationsAndRule object. We also remove that AS from the list of CopyASs. We repeat the checking and deletion for the entire list of possible paths. After all the deletion from memory completes, we finally delete the rule for the file from the FileUIDToLocationsAndRule object. There is now only the original file left in the file space. All copies are gone.

Using pingData0.txt from Case 2 as an example to show removal of a rule from a file, we start with assuming that pingData0.txt has already been copied according to the > 3 rule and the state of the file space and objects are as depicted in Figure 14 and Figure 15. To remove the > 3 rule, we take the list of paths and compare it to the Copy AS set to figure out which paths are actually copies of the original file. For pingData0.txt, the Copy ASs are AS2, AS3, AS4, AS12, AS13, and AS14, which means the files that need to be deleted are those whose paths start with /as2, /as3, /as4, /as12, /as13, and /as14. These are also the paths that must be removed from the LocationsToFileUID object. After all the files are deleted, the paths removed, and the Copy AS set cleared, the resulting file space and objects will look like Figure 18 and Figure 19.

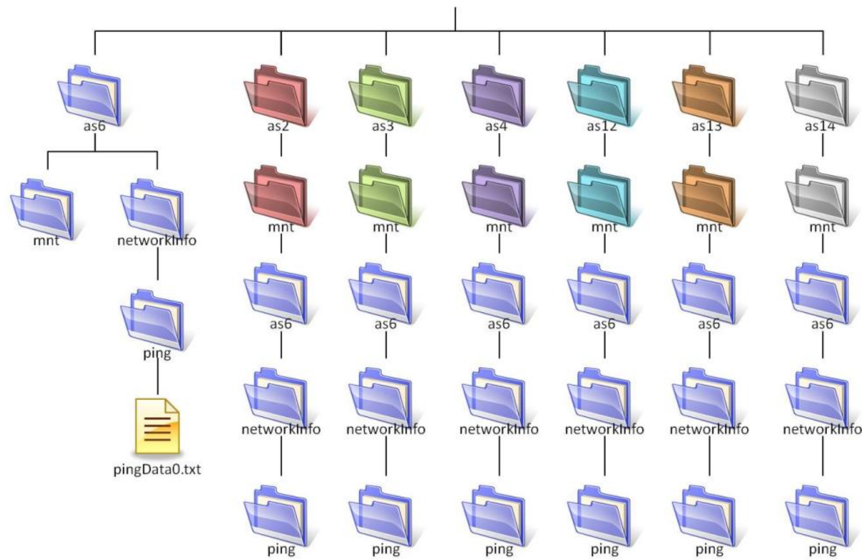

Figure 18 File space after removing the > 3 rule for the pingData0.txt file. Copies that existed in AS2, AS3, AS4, AS12, AS13, and AS14 are now gone. Original file in AS6 still exists.

FileUIDToLocationsAndRule

FileUniqueID	Paths	Main AS	Copy AS's	DistanceAwayRule
de893e1e	/as6/networkInfo/ping/pingData0.txt	as6		

LocationsToFileUID

Path	FileUniqueID
/as6/networkInfo/ping/pingData0.txt	de893e1e

Figure 19 Tables after removal of > 3 rule for the pingData0.txt file

4.2.4 Case 4: Removing the original file when it has distributed copies

All the cases discussed thus far have dealt with copying files from the original based on some DistanceAwayRule and then deleting the copies if the rule no longer applies to the file. In response to worries that the system is too restrictive, the system does allow for deletion of the original file.

In the case where the file is not duplicated, deleting the original starts with deleting the file as executed by the FileSystemManipulator object followed. After

deleting the file from memory, we remove the entry for the file from the FileUIDToLocationsAndRule and LocationsToFileUID objects.

In the case where the file has been duplicated, when we delete the original, there is a need for a new physical file to take up the title of "original file." Currently, our system selects one of the remaining copies at random to be the original file. Future implementations of the system can consider including the feature of being able to choose which file inherits the label of being the original file. The original rule for the file is voided out.

We use the example outlined in Case 2 to demonstrate what happens when the original file is deleted. We use the example where the pingData0.txt file has an original in AS6 and has copies in AS2, AS3, AS4, AS12, AS13, and AS14. The system's state is portrayed in Figure 14 and Figure 15. We now envision a scenario where the owner of the file in AS6 decides he no longer needs pingData0.txt and deletes the file. The file space now looks like Figure 20.

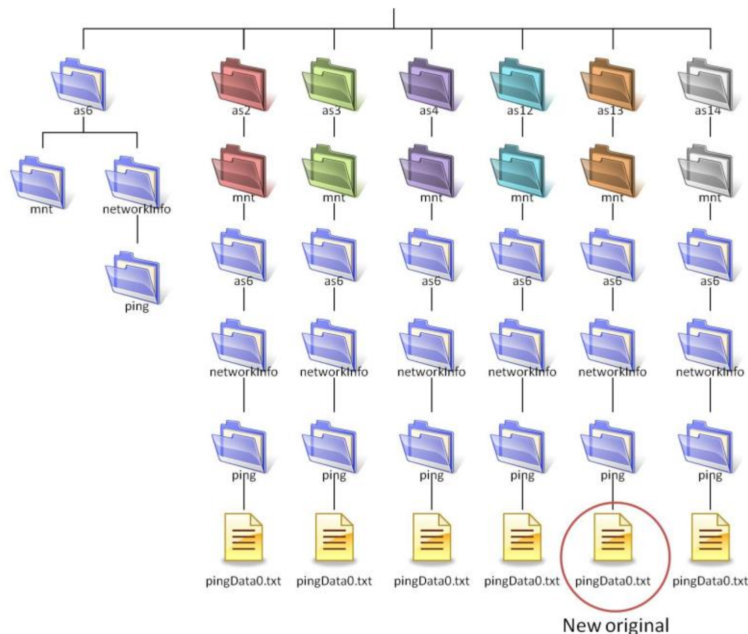

Figure 20 File space after deletion of original pingData0.txt file from AS6

Recall that we have a DirectoryWatcher on the /as6/networkInfo/ping directory that notifies our system of changes to that directory such as deletions and creations. Upon notified of the deletion of pingData0.txt from the network, the system removes the path from the LocationsToFileUID object. The path removed would be /as6/networkInfo/ping/pingData0.txt . The system takes note of the corresponding FileUniqueID from the LocationsToFileUID object right before the path is deleted, which is “de893e1e.” The system uses this FileUniqueID to retrieve from the FileUIDToLocationsAndRule object the set of all paths for this FileUniqueID. The system removes the path /as6/networkInfo/ping/pingData0.txt from the set of all paths. The FileUIDToLocationsAndRule object also checks to see if the removed path was the original file by checking the path against the Main AS field. If the path was in the Main AS, we know it was the original file. The FileUIDToLocationsAndRule object then randomly selects one of the remaining copies of the file to become the original file. It randomly selects AS13 in this case to be the new original. AS13 is removed from the Copy AS set and appointed to be the Main AS. The > 3 rule no longer applies and the system annuls the rule. The result is Figure 21.

FileUIDToLocationsAndRule

FileUniqueID	Paths	Main AS	Copy AS's	DistanceAwayRule
de893e1e	/as2/mnt/as6/networkInfo/ping/pingData0.txt; /as3/mnt/as6/networkInfo/ping/pingData0.txt; /as4/mnt/as6/networkInfo/ping/pingData0.txt; /as12/mnt/as6/networkInfo/ping/pingData0.txt; /as13/mnt/as6/networkInfo/ping/pingData0.txt; /as14/mnt/as6/networkInfo/ping/pingData0.txt	as13	as2, as3, as4, as12,, as14	

LocationsToFileUID

Path	FileUniqueID
/as2/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as3/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as4/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as12/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as13/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as14/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e

Figure 21 Tables after original pingData0.txt file in AS6 is deleted

4.2.5 Case 5: Copy a file, remove the original, and set a new rule on the remaining copies

Case 5 is a continuation of Case 4. In Case 4, the system deletes the original pingData0.txt file, and we see that the system updates so that there is a new original file, and the previous rule for distribution is annulled. Now that there is no more rule for distribution, the system is able to impose a new rule on the remaining copies of the file, even with the original file gone. The system executes the new rule in relation to the new original file. After Case 4, the original file now resides in AS13. If the new rule is to distribute copies of the file at exactly one AS hop away, the system determines the ASs relative to AS13. The ASs that fulfill the rule's requirement are AS15 and AS11.

In order to implement the "= 1" DistanceAwayRule for the file in AS13, the system first deletes all copies of the original file. If we look at Figure 21 from Case 4, these copies reside in AS2, AS3, AS4, AS12, and AS14. The deletion process is identical to Case 3. The system checks paths against the set of Copy AS, and removes those files. The implementation of the rule is then identical to the mechanisms outlined in Case 1. In Case 1, the rule was associated with the file immediately. Here, the rule is associated with the file at a later time, but once the association is made, all the same sequence of events happen to implement the rule. The ASPathDatabase is queried for qualifying ASs, the copies are made, and the LocationsToFileUID and FileUIDToLocationsAndRule objects are updated. The final file space and objects appear as in Figure 22 and Figure 23. Remember that since the paths are copied over as is from the original AS to the new ASs, the path from AS13 will have a directory named "as6" in the pathname since that portion of the path was originally copied over from AS6.

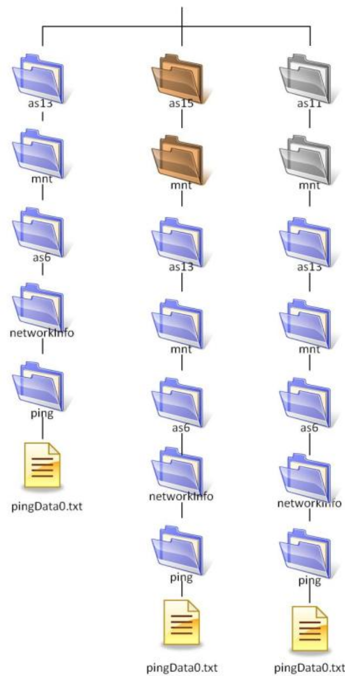

Figure 22 File space after new rule is imposed on the new original file in AS13. Copies placed in AS15 and AS11.

FileUIDToLocationsAndRule

FileUniqueID	Paths	Main AS	Copy AS's	DistanceAwayRule
de893e1e	/as13/mnt/as6/networkInfo/ping/pingData0.txt; /as11/mnt/as13/mnt/as6/networkingInfo/ping/pingData0.txt; /as15/mnt/as13/mnt/as6/networkingInfo/ping/pingData0.txt	as13	as11, as15	= 4

LocationsToFileUID

Path	FileUniqueID
/as13/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as11/mnt/as13/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e
/as15/mnt/as13/mnt/as6/networkingInfo/ping/pingData0.txt	de893e1e

Figure 23 Tables after new rule is set on the new original file pingData0.txt in AS13.

4.2.6 Case 6: Copy a file to a specific AS

In Case 6, we discuss the ability of our system to copy a file to designated ASs. Sometimes a file's owner does not need to replicate the file according to some rule. There are scenarios where an owner of the file may not want to distribute it at some distance away but at some absolutely location. For example, an owner of a

file might want to replicate the file to exactly two ASs of his choosing. These ASs are not at some distance away; they are absolute, desired locations. Our system has support for the copying of files to a specific AS, which we call spot duplicating. In order to spot duplicate a file to a particular AS, we use the copyFileToLocation method of the FileUIDToLocationsAndRule object. Given the file's path and the desired destination AS to duplicate the file to, the method replicates the file into the AS. Again, copying uses the path of the original file and places it as a subpath in the mnt directory on the destination AS. For example, if the owner of the file /as2/public/route4147/august/data2010Aug08.txt wanted to replicate the file to AS4 from AS2, the owner could spot duplicate the file to AS4. The system would create the copy /as4/mnt/as2/public/route4147/august/data2010Aug08.txt, and the LocationsToFileUID and FileUIDToLocationsAndRule objects would reflect this new copy. Figure 24 shows the resulting file space. Figure 25 shows the resulting tables.

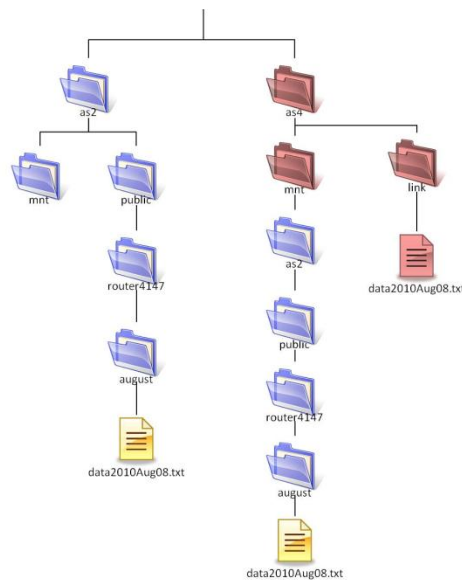

Figure 24 File space after spot duplication of data2010Aug08.txt from AS2 to AS4

FileUIDToLocationsAndRule

FileUniqueID	Paths	Main AS	Copy AS's	DistanceAwayRule
6e6f6f9a68	/as2/public/route4147/august/data2010Aug08.txt; /as4/mnt/as2/public/route4147/august/data2010Aug08.txt	as2	as3	

LocationsToFileUID

Path	FileUniqueID
/as2/public/route4147/august/data2010Aug08.txt	6e6f6f9a68
/as4/mnt/as2/public/route4147/august/data2010Aug08.txt	6e6f6f9a68

Figure 25 Tables after spot duplication of data2010Aug08.txt from AS2 to AS4

Support of spot duplication also works as a proof of the validity of the state of the network in Case 4 after the original file is deleted. In Case 4, we delete the original file, and with it, the original rule associated with the file. After the deletion of the file, the network exists such that there are copies of the file residing in ASs around the network, but there is no rule tying the multiple locations together. One might question whether this is a legal state to have seemingly random identical files distributed around the network with no rule for why they are there. We argue that this is indeed a valid state because if we look at Case 4, after the original file in AS6 is removed, the copy of the file in AS13 becomes the new original file. We can now view the copies as AS2, AS3, AS4, AS12, and AS14 as copies of the file in AS13. It would seem as though the file's copies should not be allowed to occupy AS2, AS3, AS4, AS12, and AS14 because there is no longer any rule for the file that say those copies should be in those locations and that the resulting state is an illegal state. We claim that this state is actually possible because it is reachable by spot duplicating the file in AS13 to AS2, AS3, AS4, AS12, and AS14, thus negating the need for a rule for the copies to reside there.

4.3 Addendum to the Rule

We have discussed in great detail the mechanism for replication of a file in our system. As a somewhat significant postscript, we would like to note that it would be hardly difficult to add to our DistanceAwayRule a "total count of copies" field that would define the total number of copies that need to exist in the network at

any given time. Currently, our implemented system replicates the file without taking into account how many copies are in the network; our system simply wants to achieve the goal of the rule. It is not complex for the system, during the copying process, to either cap the number of copies of a file it makes, or to make extra copies to place in more ASs. Although our implementation currently does not enforce a minimum/maximum number of file copies that need to exist within the network, to include that qualifier is a simple step added during the replication step and would not increase the complexity of our system. This can be a feature neatly and easily added to extend our system.

However, although it is relatively simple to create the correct number of copies, it is a much more complex problem to guarantee that number of copies of a file to continuously exist. If we imagine a hypothetical scenario where an AS is more dynamic than average, it might become difficult to promise that the copy of a file we place in that AS will promise to be there. Because machines are constantly coming and going in this AS, it is entirely possible that the machine the file sits on somehow leaves the AS due to failure or some other cause. Considering this scenario introduces the need for some sort of monitoring mechanism on an AS to guarantee that a file really does exist within the AS it wants. There may be a need for additional replication of a file within an AS, and some sort of garbage collection mechanism to take care of extra files that are created temporarily.

4.4 Recapitulation

The above six cases illustrate the workings of an implementation of our system on a real network of sixteen ASs. Our system is able to handle the duplication of files according to five different types of distance rules. The rules are $<$, $>$, $\leq$, $\geq$, or $=$ to a certain number of AS hops from the original AS. If the owner of a file deems a rule no longer necessary for a file, the rule can be removed from the file, and our system knows how to delete the copies of the file that were made

according to the now obsolete rule. The case studies also demonstrate that after a file is duplicated, the original can be removed and our system will automatically know to reassign a new original file out of the remaining copies of the file. Also, removing the original file means a new rule can be set on the remaining copies, and the system will make the proper corrections so that the file is duplicated to the correct ASs. Finally, we show that our system does not necessarily have to make copies of a file according to a rule. The system can copy a file to a very specific Autonomous System named by the file's owner without needing a rule.

5 Conclusion

5.1 Overview

Network management today is faulty because of its fragmentation. However, creating a unified system for network management necessitates using information about the network itself. Our goal at the outset was to propose and implement a network-wide file system for managing and organizing the files within the network that contain the information necessary to manage the network itself. The system would be able to supply these files to be used as an aid in a higher level of network management. The pivotal point of our system is that it not only provides a mechanism for supplying the required files, but it creates an entire system of replicating important information and placing copies at desirable locations such that these files are readily available even when one location for the file is inaccessible.

In order to build the necessary components of our file location, replication, and distribution system, we partitioned the problem space concerning network information and file management into two spaces. The first partition is the decision-making process that determines which network locations files should be copied to in order to maximize accessibility while minimizing redundancy. These decisions are made with the objective of ensuring a file is available in the case of failures in a network. The second partition is the enactment and execution once the decision has been made. Our system supplies a mechanism for the second part of the problem space. That is, for network-level decision makers who have developed a set of rules to distribute files containing network management information, our system provides a means to implement those rules across the entire network.

In our system, the decisions that external entities make manifest themselves as rules called DistanceAwayRules. These rules help to distribute files at some

Autonomous System hop distance away from their original AutonomousSystem location. The concept of Autonomous System hop distance is important because if our goal at the onset was to try to guarantee availability of files, we have to ensure that the locations where we place these files will not fail simultaneously. Our system argues that Autonomous System hop distance is a sufficient level for distributing files such that we achieve these independent failures because it is highly unlikely that two Autonomous Systems will fail concurrently. The same cannot be said for placing copies of files at the machine level since there remains a significant probability that machines in the same area will abort synchronously.

This system is a file location, replication, and distribution system. By executing a desired rule for a file, our system fulfills the replication and distribution requirement for a file by copying a file to the necessary destination locations that conform to the rule. Our system provides the location service for a file by maintaining a listing of where a file and all its copies reside. This file and its copies are all identified by a single unique identifier called its FileUniqueID. Using this FileUniqueID, the system is able to look up the file in the FileUIDToLocationsAndRule table to provide a list of all locations for that file. The LocationsToFileUID table is another utility that allows an external user of our system to look up all the copies for a given file if the particular file the user is looking for is unavailable. These tables provide the location portion of our system. The end result is that the system is capable of taking a file in the network, replicating it, and distributing the file according to its associated rule. It is the integration of the various components of our system that helps it achieve the end goal of managing network information.

Our system proves that it is possible to implement a framework for managing network information and maintaining its availability according to guidelines and rules. The system we proposed has strengths in that, as a functional system, it is still modular and adaptable. The underlying components of our system can

change in implementation. Our system provides a general framework and design for how information can be managed as opposed to a concrete, fully-specified solution. As an illustration, the tables in our system for looking up where files are located are simply abstractions. We chose to implement them using tree data structures, but it is just as possible that the underlying implementations used could be databases or other data structures. The central principle is that we abstract away these details in our design by using the table abstraction and are therefore able to talk about our system at a higher level knowing that the pieces of the system will still integrate smoothly.

The file location, replication, and distribution system we built is a functional model for how to manage information in the network so that it can be readily available for network management. The system accomplishes the goal of making information available by placing it in intelligent locations based on a set of rules, and it is robust in that it allows for modification of rules and replication to new locations as needed. Overall, we successfully created a design and implementation for a system that manages the location, replication, and distribution of network information..

5.2 Going Forward

Looking forward, there are still related problem spaces that will eventually affect our system, and there may be a need to re-examine tradeoffs that were made to create our system.

For the near future and next steps, it would be beneficial to simulate our system on larger and larger networks. We worked with a system of 16 Autonomous Systems with three machines each, but for the sake of simplification and experimentation, much of the copying of files across ASs was copying files between delegate machines for each AS. Instead of copying files from one AS to any random machine in the destination AS, it was more illuminating to copy the file to a designated machine in an AS in order to measure the performance of our

system. In the future, we can simulate our system to incorporate an increasing number of machines in order to see how our system would perform on a larger network of computers. Our goal was to show a proof of concept and working model of the general framework. Now that the model for file management exists, a judicious follow up will be to test with intense scaling. Scaling is an especially crucial part to future work because many of the design choices made were simplifications or the most obvious first choice. As of now, not all the simplifications scale well, though they do demonstrate features clearly. A key challenge in the future will be to design for scaling up both in the network itself and in the number of information objects or files. An interesting future project may be to watch performance of the system as many files are created simultaneously all across the network in order to see if our system can handle heavy load. By scaling to larger and larger magnitudes, we hope to learn how our system will perform in real world networks.

For the future of our design, we foresee more extensions to our system particularly in the rule space, as well as further optimizations. For now, our system allows support for duplicating files at some AS hop distance away from the origin. One possible extension to our system would be to iterate the concept of hop distance down to the machine level, or perhaps to some other level of network organization. The principles that hold at the AS level would be the same but would instead now execute at the machine level or some other level. Also, the support of many types of rules as well as more complex rules will play a factor in the growth of our system. An example of a potential rule would be to support replication to ASs where the AS exhibits attributes 1, 2, and 3. As for optimizations to the system, there is the possibility that we will need to modify the concept of the “original” file when it comes to replication. In particular, it is not impossible to envision a scenario where our system will need to support the ability to set which file is considered as the original file, regardless of what file came into existence first. The capability to select which file is the original file can

be of great assistance when the first genuine file is removed from the network. As of now, our system treats all copies of the same file equally, and its current policy is that any file can become the new original file. It is possible that we can improve our system by providing some method to give some copies priorities over the others.

Machine learning is another possible addition to our system. One possible place where machine learning could play a role is in dealing with heavy load in the network. An AS might be unavailable often because that particular AS often experiences heavy load, and machine learning could be a method for learning which ASs endure greater than average duress. Using knowledge of differential network load, the system could provide recommendations for where information should reside if we want it to be readily and constantly available. The system could eventually incorporate a method for ranking ASs based on how light or heavy their load is, then preferentially avoiding placing information in ASs that bear larger loads or more stress.

One last consideration is how the real world will affect implementation of our system. Our system is a working model and functions successfully on experimental networks, but a real computer network exists in the very human world of policies and laws with security concerns and other various constraints. Our system is able to place information in different ASs, but this is only possible in the real world if there are no restrictions. However, some ASs might not allow another AS to place information within itself, so though our system has that ability, it does not have the liberty. These policy issues could possibly affect the rules of our system, depending on whether we choose to implement the policy constraints or whether we leave policy as an external entity that acts out in the decision-making domain of the problem space.

We want to move toward a world with better, smarter network management. This desire drives the need for the management system to leverage the

information within the network it is managing. Our system provides a means for organizing that information, but it is crucial to remember that the human world is a very complex realm whose decisions and laws will affect how our system can actually be implemented. However, we view our system as a design model to build upon. It is our hope that our system will successfully adapt to whatever constraints the human world imposes, and that this work will ultimately aid in the creation of a stronger Internet network.

6 Appendix A: File Management Objects and their Methods

6.1 ASPathDatabase

- getInstance()
- containsPath(AutonomousSystemPath)
- containsPath(AutonomousSystem, AutonomousSystem)
- addPath(AutonomousSystemPath)
- getAllPathsForStartingAS(String)

6.2 AutonomousSystem

- AutonomousSystem(int)
- AutonomousSystem(int, Machine…)
- AutonomousSystem(int, List<Machine>)
- getASNumber()
- getASName()
- addMachine(Machine)
- removeMachine(Machine)
- getNumOfMachines()

6.3 AutonomousSystemGenerator

- getInstance()
- createAS()
- getTotalNumOfAS()
- getASByName(String)
- asExists(String)

6.4 AutonomousSystemPath

- AutonomousSystemPath(AutonomousSystem…)

- AutonomousSystemPath(List<AutonomousSystem>)
- getPathStart()
- getPathEnd()
- getPath()
- getPathLength()
- getASxHopsFromStart(int)

6.5 BinaryTree

- BinaryTree()
- getSize()
- add(E)
- addKeyValuePair(E, String)
- addUIDLocationRule(E, String, DistanceAwayRule)
- remove(E)
- getNodeForKey(E)
- get(E)
- getValueforKey(E)
- getValuesForKey(E)
- toList()
- toListValues()
- toListMainValues()

6.6 DistanceAwayRule

- DistanceAwayRule(int, RuleOperator)
- getDistance()
- getOperator()

6.7 FileLocation

- FileLocation(String)
- getASOfLocation(String)

6.8 FileUIDToLocationsAndRule

- FileUIDToLocationsAndRule(ASPathDatabase, LocationsToFileUID)
- addNewUIDLocRuleEntry(FileUniqueID, String, DistanceAwayRule)
- addUIDLocEntry(FileUniqueID, String)
- addLocationForFile(FileUniqueID, String)
- addRuleForFile(FileUniqueID, DistanceAwayRule)
- removeRuleForFile(FileUniqueID)
- removeLocationForFile(FileUniqueID, String)
- containsFile(FileUniqueID)
- getLocationsForFile(FileUniqueID)
- getMainASForFile(FileUniqueID)
- getCopyASForFile(FileUniqueID)
- getRuleForFile(FileUniqueID)
- getNumOfUniqueFiles()

6.9 FileUniqueID

- FileUniqueID(String)
- createIDForFile(String)

6.10 LocationsToFileUID

- LocationsToFileUID()
- getUIDForPath(String)
- addLocationFileUID(String, FileUniqueID)
- removeLocation(String)
- getSize()

6.11 Machine

- Machine(int)
- Machine(int, String)
- Machine(String)

- setName(String)
- getName()

6.12 **Node**

- Node(E)
- Node(Node<E>)
- Node(E, String)
- Node(E, String, DistanceAwayRule)
- getLocations()
- getMainAS()
- getCopyAS()
- addLocation(String)
- removeLocation(String)
- getRule()
- setRule(DistanceAwayRule)
- getKey()

6.13 **RuleOperator**

- LESS_THAN
- GREATER_THAN
- LESS_THAN_OR_EQUAL_TO
- GREATER_THAN_OR_EQUAL_TO
- EQUAL_TO

7 Appendix B: Directory Watching Objects and their Methods

The objects located here are open source code that has been modified. The original source for the code can be found at http://twit88.com/blog/2007/10/02/develop-a-java-file-watcher/. Because these objects are open source, we only list the methods of those objects that were modified.

7.1 AbstractResourceWatcher

7.2 BaseListener

7.3 DirectorySnapshot

7.4 DirectoryWatcher

7.5 FileListener

7.6 IFileListener

7.7 IntervalThread

7.8 IResourceListener

7.9 IResourceWatcher

7.10 MyFileListener

- MyFileListener()
- onStart(Object)
- onStop(Object)
- onAdd(Object)
- onChange(Object)
- onDelete(Object)

8 Appendix C: File I/O Objects and their Methods

8.1 FileSystemManipulator

- copyFile(String, String)
- copyFile(File, File)
- copyFileToMultipleLocations(String, List<String>)
- deleteFile(String)
- renameFile(String)

8.2 LongToByte

- longToByteArray(long)

8.3 MD5Calculator

- md5OfString(String)
- altMD5(String)
- calculateFileMD5WithDate(String)
- calculateFileMD5WithoutDate(String)

9 Works Cited

[1] Greenberg, A.; Hjalmtysson, G.; Maltz, D. A.; Myers, A.; Rexford, J.; Xie, G.; Yan, H.; Zhan, J.; and Zhang, H. 2005. A clean slate 4D approach to network control and management. *SIGCOMM Comput. Commun. Rev.* 35, 5 (Oct. 2005), 41-54. DOI= http://doi.acm.org/10.1145/1096536.1096541

[2] Clark, D. D.; Partridge, C.; Ramming, J. C.; and Wroclawski, J. T. 2003. A knowledge plane for the internet. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (Karlsruhe, Germany, August 25 - 29, 2003). SIGCOMM '03. ACM, New York, NY, 3-10. DOI= http://doi.acm.org/10.1145/863955.863957.

[3] Sollins, K. R. 2009. An architecture for network management. In *Proceedings of the 2009 Workshop on Re-Architecting the internet* (Rome, Italy, December 01 - 01, 2009). ReArch '09. ACM, New York, NY, 67-72. DOI= http://doi.acm.org/10.1145/1658978.1658995.

[4] Thaler, D. G. and Ravishankar, C. V. 2004. An Architecture for Inter-Domain Troubleshooting. *J. Netw. Syst. Manage.* 12, 2 (Jun. 2004), 155-189. DOI= http://dx.doi.org/10.1023/B:JONS.0000034212.53702.30.

[5] Moore, J.; Chase, J.; Farkas, K.; Ranganathan, P. 2004. A Sense of Place: Toward a Location-aware Information Plane for Data Centers. HPL-2004-27.

[6] Lee, G. J. 2007 *Capri: a Common Architecture for Distributed Probabilistic Internet Fault Diagnosis*. Doctoral Thesis. UMI Order Number: AAI08194.42., Massachusetts Institute of Technology.

[7] Kulik, J. 2003. Fast and flexible forwarding for Internet subscription systems. In *Proceedings of the 2nd international Workshop on Distributed Event-Based Systems* (San Diego, California, June 08 - 08, 2003). DEBS '03. ACM, New York, NY, 1-8. DOI= http://doi.acm.org/10.1145/966618.966635

[8] Jokela, P., Zahemszky, A., Esteve Rothenberg, C., Arianfar, S., and Nikander, P. 2009. LIPSIN: line speed publish/subscribe inter-networking. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication* (Barcelona, Spain, August 16 - 21, 2009). SIGCOMM '09. ACM, New York, NY, 195-206. DOI= http://doi.acm.org/10.1145/1592568.1592592

[9] Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., and Lilley, J. 2000. The design and implementation of an intentional naming system. *SIGOPS Oper. Syst. Rev.* 34, 2 (Apr. 2000), 22. DOI= http://doi.acm.org/10.1145/346152.346192

[10] Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., and Braynard, R. L. 2009. Networking named content. In *Proceedings of the 5th international Conference on Emerging Networking Experiments and Technologies*(Rome, Italy, December 01 - 04, 2009). CoNEXT '09. ACM, New York, NY, 1-12. DOI= http://doi.acm.org/10.1145/1658939.1658941.

[11] Jacobson, V., Smetters, D. K., Briggs, N. H., Plass, M. F., Stewart, P., Thornton, J. D., and Braynard, R. L. 2009. VoCCN: voice-over content-centric networks. In *Proceedings of the 2009 Workshop on Re-Architecting the internet* (Rome, Italy, December 01 - 01, 2009). ReArch '09. ACM, New York, NY, 1-6. DOI= http://doi.acm.org/10.1145/1658978.1658980

[12] http://psirp.org

[13] Madhyastha, H. V., Isdal, T., Piatek, M., Dixon, C., Anderson, T., Krishnamurthy, A., and Venkataramani, A. 2006. *iPlane*: an information plane for distributed services. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington, November 06 - 08, 2006). Operating Systems Design and Implementation. USENIX Association, Berkeley, CA, 367-380.

[14] Madhyastha, H. V., Katz-Bassett, E., Anderson, T., Krishnamurthy, A., and Venkataramani, A. 2009. iPlane Nano: path prediction for peer-to-peer applications. In *Proceedings of the 6th USENIX Symposium on Networked*

Systems Design and Implementation (Boston, Massachusetts, April 22 - 24, 2009). USENIX Association, Berkeley, CA, 137-152.

[15] Eugster, P. T.; Felber, P. A.; Guerraoui, R.; and Kermarrec, A. 2003. The many faces of publish/subscribe. *ACM Comput. Surv.* 35, 2 (Jun. 2003), 114-131. DOI= http://doi.acm.org/10.1145/857076.857078.

[16] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. 2003. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.* 11, 1 (Feb. 2003), 17-32. DOI= http://dx.doi.org/10.1109/TNET.2002.808407