Survivable Software Distribution and Maintenance

by

Jeff Breidenbach

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

May 1997

© Jeff Breidenbach, MCMXCVII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author .....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by .....
Clifford Weinstein
Group Leader
Thesis Supervisor

Certified by .....
Thomas M. Parks
Technical Staff
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

OCT 2 9 1997

LIBRARIES

Survivable Software Distribution and Maintenance

by

Jeff Breidenbach

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes a software distribution and maintenance system that reduces administration costs while increasing survivability. Contributions include design, implementation, and analysis. The system applies automation wherever possible and takes maximum advantages of economies of scale.

The implementation extends an existing replication tool to provide multiplatform support, user assisted cache management, and support for middlemen in the distribution system. Middlemen relieve both system administrators and end users of administration and maintenance tasks. This arrangement leads to higher survivability, lower operating costs, and allows new value added services in software maintenance.

Thesis Supervisor: Clifford Weinstein
Title: Group Leader

Thesis Supervisor: Thomas M. Parks
Title: Technical Staff

Acknowledgments

I would like to acknowledge the many people who made this thesis possible. That includes my advisors Cliff Weinstein and Tom Parks from MIT Lincoln Laboratory. Tom deserves special credit for providing detailed feedback and keeping me from going completely insane. Everest Huang did more little favors than I can ever hope to repay. Mom, Dad, and the US Air Force footed the bill, and provided encouragement. Steve Benton and Ravi Pappu helped steer me when I was looking for a topic. Charlotte Brown is going to be a fine astronaut. People not already mentioned who directly contributed to this document (no matter in how small a fashion) include Paul Van Eyk, Richard Stallman, Brian Semmes, Alan "Dr. of Pain" Au, Matt Lennon, Cyrus P. Master, Boris Raykin, Greg Richardson, Tim Piwowar, and Jean Elien. I also want to acknowledge Hugh Morgenbessor as an all around good guy and the MIT Nordic Ski Team for their patience. I was going to dedicate this thesis to all of humanity, but in the end, decided to save that for my next project.

Contents

1 Introduction **10**

1.1 Software requires maintenance 11

1.1.1 Interoperability 11

1.1.2 Adding features 12

1.1.3 Blurring of data and programs 12

1.1.4 Moore's Law 13

1.1.5 Bugs 13

1.1.6 Security 14

1.2 The onerous burden of maintenance 15

1.2.1 Quantifying maintenance costs 15

1.2.2 Download based distribution 16

1.2.3 Shrink wrap distribution 17

1.2.4 Central administration 19

1.2.5 Package management 21

1.2.6 Push technology 22

1.2.7 Summary of distribution technologies 23

1.3 The ideal world 25

2 Design **27**

2.1 Design Overview 27

2.1.1 Design goals 27

2.1.2 Environment 27

2.2 Design Issues 28

2.2.1 Survivability 28
2.2.2 Mobility 28
2.2.3 Checkpoints 29
2.2.4 Economics 30
2.2.5 Access Control and Authentication 30
2.2.6 Scalability 31
2.2.7 Version Skew 31
2.2.8 Push vs. Pull 32
2.2.9 Redundancy 32
2.2.10 Network Efficiency 33
2.2.11 Caching issues 33
2.2.12 User space versus system space 34
2.2.13 Platform management 35
2.2.14 Ease of use 35
2.2.15 Closure 35
2.3 Tradeoffs 36
2.3.1 Version skew, mobility, push/pull 37
2.3.2 Access control, scalability 37
2.3.3 Ease of use, efficiency 38
2.4 Focus 38
2.5 Putting it all together: an example 38

3 Implementation **41**
3.1 Technologies 41
3.1.1 Basic components 41
3.1.2 Castanet in detail 42
3.2 Basic distribution systems 43
3.2.1 Replication from source A to destination B 44
3.2.2 Combining two sources 47
3.2.3 Adding a checkpoint 49

3.2.4 Replicating a large source 50
3.3 Maintenance loops 52
3.4 Implementation components 54
3.4.1 Payload program 54
3.4.2 Transmitter plugin 54
3.4.3 The merge-channels program 55

4 Analysis **56**
4.1 Performance Criteria 56
4.1.1 Physical resources 56
4.1.2 Resource demands on a single machine 57
4.1.3 Resource demands on the system 59
4.1.4 When demand exceeds capacity 59
4.2 Comparison of technology bases 60
4.2.1 Castanet 61
4.2.2 Coda 61
4.2.3 WebFS 62
4.2.4 WebNFS 64
4.3 Other maintenance systems 65
4.3.1 Oil Change and Tuneup.com 65

5 Applications **66**
5.1 Value added services 66
5.1.1 General software collections 66
5.1.2 Safety checking 66
5.1.3 Specialty distributions 67
5.1.4 Frozen software 67
5.1.5 Retail stores 67
5.1.6 Modifications or customizations 68
5.1.7 Unanticipated services 68
5.2 Future work 68

5.2.1 Production quality implementation 68
5.2.2 Read/Write 68
5.2.3 Economic model 69

A Athena: an example of a large software collection **70**
A.1 outland 70
A.2 graphics 72
A.3 gnu 73
A.4 sipb 75

B Protocols and Design Standards **76**

C Source Code **78**

List of Figures

1-1 Maintenance Link 15

1-2 Download based distribution requires $M \times N$ links 17

1-3 Shrink wrap distribution requires $M \times N$ links 18

1-4 A centrally maintained system requires $M + N$ links 20

1-5 Centrally maintained systems require $(K \times M) + N$ maintenance links. 21

1-6 Relative performance of distribution technologies. A star (*) denotes distribution technologies which can be shifted slightly to the left by applying package management. The dark line represents the focus of this thesis. 24

2-1 Design tradeoffs. An **X** denotes two desired features which are in conflict. 36

2-2 Trickle down software at MIT Lincoln Laboratory 39

3-1 The xclock program 44

3-2 A two person distribution system 44

3-3 The xeyes program 48

3-4 Distribution system with a middleman. 48

3-5 Adding a checkpoint 50

3-6 A large software source 51

3-7 A two person maintenance loop 53

B-1 A simplified diagram of the standard multiplatform directory structure. 77

List of Tables

1.1 Security statistics published by DataPro 14
1.2 Focus of distribution technologies 23
4.1 Feature space of existing systems 64

Chapter 1

Introduction

A survivable system continues to function despite component failures. The human body is a survivable system: a minor injury to the foot will rarely threaten someone's life. The world economy is another survivable system: a single bankruptcy will not crash the entire world economy — we hope! Computer networks are sometimes survivable, and sometimes not. A necessary but not sufficient condition for survivability in a computer network is proper software maintenance. Unfortunately, proper software maintenance is expensive and difficult. The rapid growth of the internet has led to more inexperienced users and more misconfigured and poorly maintained computers, increasing instability.

This thesis examines gross inefficiencies and high costs in current software distribution and maintenance systems. An alternative system is described, suggesting improvements in efficiency. A proof-of-concept implementation is presented.

Chapter one describes problems with current software distribution and maintenance systems, and the unequivocal need for a better way.

Chapter two describes design objectives for a replacement distribution system. This design uses distributed maintenance to prevent redundant work.

Chapter three describes the proof-of-concept implementation. This implementation builds on several existing software components and programs.

Chapter four analyzes the implementation and compares it to other distribution systems. A metric is presented to aid in comparison.

Chapter five discusses applications for this new distribution system. Many new services are possible with a distributed maintenance system; several are discussed.

1.1 Software requires maintenance

"This software is as worthless as yesterday's newspaper." Information and software programs change rapidly and require constant maintenance. Keeping software and information current is a difficult and time-consuming task. Rather than talk about generalities, we will delve straight into a specific example.

The most popular personal computer application released in 1996 was a world wide web browser called Netscape Navigator. [22] During 1996, fourteen versions of the program were released to the public, including 2.0b4, 2.0b5, 2.0b6a, 2.0, 2.01, 2.02, 3.0 PR1, 3.0 PR2, 3.0b4, 3.0b5, 3.0b6, 3.0b7, 3.0, and 3.01. [11, 27] In order for typical users to maintain the latest copy of Netscape Navigator on their local file system, they would have to manually download, install, and upgrade their software many times.

Not all software programs change as rapidly as Netscape Navigator. However, most software programs are dynamic media that are improved, configured, patched, upgraded, fiddled with, and otherwise changed over time. This change is important, necessary, and occurs for a variety of reasons.

1.1.1 Interoperability

As new conventions and standards are adopted, software is modified to support them. For example, there are dozens of popular file formats for storing bit-mapped images. In 1996, a new file format called Portable Network Graphics (PNG) was selected and recommended by the World Wide Web Consortium for use on all web pages. [19] Existing imaging programs are being modified to support PNG to attain operability with the new standard.

Movement to new standards in file formats, datatypes, and protocols happens all the time in the software industry. Current large scale transitions include the

movement from HTTP 1.0 to HTTP 1.1 [31] (HTTP is the basic protocol of the world wide web). Microsoft Corporation just changed the proprietary file format for Microsoft Word with their latest release of the program. [9] Sun's new Java Virtual Machine Specification version 1.1.1 contains significant differences from its predecessor version 1.0.2. [38]

These changes in standards and conventions require updating millions of copies of software programs if they wish to speak the new common language, file format, or bytecode specification.

1.1.2 Adding features

Software evolves over time. Developers may wish to improve existing characteristics or add functionality to software that was not included in the original release. Adding new features can enhance software and keep it competitive in the market. While the process is occasionally abused, it is not unreasonable for a developer to make design changes or improvements over time. "Feature-itis"[1] is a major driving force for rapidly changing software.

1.1.3 Blurring of data and programs

At one time the popular perception of the software world was divided into two parts — data and programs. Data is information that is created, generated, and communicated all the time. Programs are used to display and otherwise manipulate that data. Data is dynamic while programs are static.

This mindset is disappearing as programs and data become more and more integrated. Consider a document created with Microsoft Word. Is the document data or a program? Using a macro language, the document is capable of executing in the right environment, and computer viruses have been spread in this fashion. Multimedia formats and executable content on web pages further blur the distinction between data

[1] "Feature-itis" is a common disease amongst software manifested by the continuous addition of new features, sometimes referred to as software bloat.

and programs. Is the ticker tape Java applet running across a web page a program or data? As more and more data and programs fall under the heading of "executable content," software and data become indistinguishable.

It becomes less meaningful to say "I'll update my programs every five years and deal with new data everyday." When there is no difference between data and programs, that schedule is equivalent to updating software every day.

1.1.4 Moore's Law

Moore's law states that computer hardware performance doubles every 18 months. Historically, the computer industry has experienced rapid technological development and continues to do so. Rapidly changing hardware requires software to change as well. These changes are required to attain compatibility with new hardware and new hardware capabilities.[2] Often just to keep a program usable over a period of a few years requires continuous recoding as underlying platforms change. Between 1992 and 1996, MIT's campus information system went through three generations of hardware revision. Not a single public campus workstation remained in service over the four year interval. [36]

1.1.5 Bugs

Bugs, bug exploitation, and bug fixes are critical reasons for the revision of software. It is not clear whether any large and complex piece of software has ever been written without bugs. These software mistakes and glitches are sometimes systematically fixed and removed over time. Often the debugging process for a large piece of software will require several years.

[2]Virtual machines and emulation are techniques used to lessen the shock of rapidly changing hardware.

1.1.6 Security

Bugs and other programming mistakes are of particular interest in the area of computer security. Apache, the leading web server, commands approximately 40% of the world market. [15] Up until version 1.0.3, Apache contained a serious security hole that allowed outside hackers to easily attack a web site. [7] This problem was detected and corrected over a year ago in the Apache software distribution. Unfortunately, this security hole will still be causing problems far into the future become some sites will continue to run older versions of the software.[3] Many of the frightening statistics in table 1.1 stem from exactly this problem. [23]

Estimated number of hacker attacks on Department of Defense networks in 1996	500,000
Estimated percentage that were successful	65
Average number of potentially damaging hacker attempts on Bell Labs networks in 1992, per week	6
Average number of less threatening attacks, per week	40
Average rate of attacks in 1996	no longer tracked
Percentage of existing bank web sites found to have potentially significant security holes	68
Percentage of Web sites in a random selection with such holes	33

Table 1.1: Security statistics published by DataPro

Another prime example of the relationship between security and maintenance is sendmail. Sendmail is a Unix program which has a certain notoriety for requiring frequent security updates. [6, 3, 4, 10, 5] Many systems being attacked were running obsolete versions of sendmail. This security advisory from 1995 is typical:

> The CERT Coordination Center has received reports of a vulnerability in sendmail version 5. Although this version is several years old, it is still in use. The vulnerability enables intruders to gain unauthorized privileges, including root. We recommend installing all patches from your

[3]The author's computer receives approximately two attempted attacks per month by people trying to exploit this particular security flaw.

vendor or moving to the current version of Eric Altman's sendmail (version 8.6.12) [5]

Many computer software security breaches fall into this category. A weakness is publicly known, but due to inability, lack of expertise, or insufficient resources on the part of the system maintainer, the patch or upgrade is not installed. It takes an enormous amount of time to keep system software secure; much of that time is spent keeping track of all the security patches and updates.

1.2 The onerous burden of maintenance

1.2.1 Quantifying maintenance costs

Computer maintenance is expensive. Computer security expert Dan Farmer, when talking about his own computer system, says, "If it takes an additional 5 percent to run a really tight ship, I'd just as soon go see a movie or drink some more wine." [24] Unfortunately, maintenance costs are much higher than five percent and in many cases are prohibitive. According to the Gartner group, corporations spend approximately $6,000 per year per personal computer on upkeep — and this is still insufficient to adequately maintain their systems. [32]

Figure 1-1: Maintenance Link

Software maintenance consists of copying, configuring, and updating many different programs. The basic unit of software maintenance, dubbed a "Maintenance Link" consists of the time and effort required to maintain a piece of software from a source A to a destination B, as depicted in figure 1-1. Each maintenance link has a cost associated with it, which corresponds to the dollar amount required for such efforts.

Note that the maintenance link does not represent the cost of licensing a program — only the overhead costs incurred during administration.

For example, let's consider a maintenance link for the software program Matlab. Source A might be the developer (Mathworks, Inc.) and destination B is an end user. The maintenance link describes the cost of copying, installing, configuring, and updating the software from A to B over its lifetime. These overhead costs are typically paid for by end user B.

A software distribution system requires many programs to be maintained on many separate computers. The total cost of a distribution system can be analyzed under two key criteria: the total number of maintenance links and the cost of each one. In many distribution systems, administration is expensive, redundant work.

Current distribution systems and technologies worth examining include downloading programs from the internet, shrink wrap distribution, centralized administration, package management, and push technology. Note that some of these technologies can be used in conjunction with each other. What are the relative strengths and weaknesses of these distribution systems? How can they be compared? Keep in mind that achieving the same results with fewer maintenance links removes redundant work. Reducing the cost of maintenance links makes the administrative work less expensive, less error prone, and therefore more robust.

1.2.2 Download based distribution

One popular distribution system consists of end users downloading software from developers over the internet. In this scenario, the customer downloads the software from a world wide web site, often directly from the original manufacturer.

Consider the case where several software programs are being maintained on several different computers. With M sources and N destinations, $M \times N$ links are required. For example, five end users would like to install the software programs Matlab, Framemaker, Netscape Navigator, Emacs, and Tetris on their machines. The download based distribution system requires each end user to establish five maintenance links, one to each software source, for a total of twenty five maintenance links in the

distribution system. The result is a connected graph of maintenance links between the developers and the end users. (See figure 1-2)

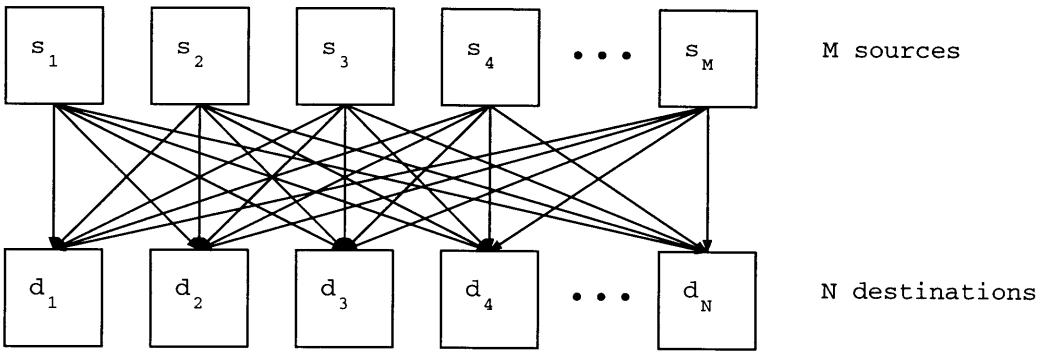

Figure 1-2: Download based distribution requires $M \times N$ links

Each maintenance link is very expensive, since a high degree of time consuming human intervention is involved. Each end user must find and individually install every program she wishes to use. Once a piece of software is located, the user will need to uncompress it, read the instructions, possibly compile and configure the program, and decide where it will reside in local storage. When the piece of software is updated, the end user must first become aware of the update, locate it, download it, and then repeat the installation process. All this manual labor makes proper maintenance of a large software collection extremely expensive and error prone for a single end user. In a distribution system containing many end users, the aggregate cost is enormous.

1.2.3 Shrink wrap distribution

Shrink wrap distribution is a traditional and very popular way of distributing commercial software. In this system a software program is written onto some portable, physical media such as a floppy disk or a CD-ROM. A glossy manual is packaged with the software and sealed in a box covered by shrink wrap plastic. These programs are distributed in a similar fashion to other commodities, in stores, retail outlets, and mail order chains.

Stores add the notion of a middleman. By collecting large amounts of shrink

wrap software together, stores make it easier to locate a particular piece of software; however, this saving is only a small part of total maintenance costs. Each end user will have to maintain, upgrade, configure, and install each piece of software separately. Thus the shrink wrap distribution system still requires $M \times N$ maintenance links despite providing a convenient central repository. (See figure 1-3)

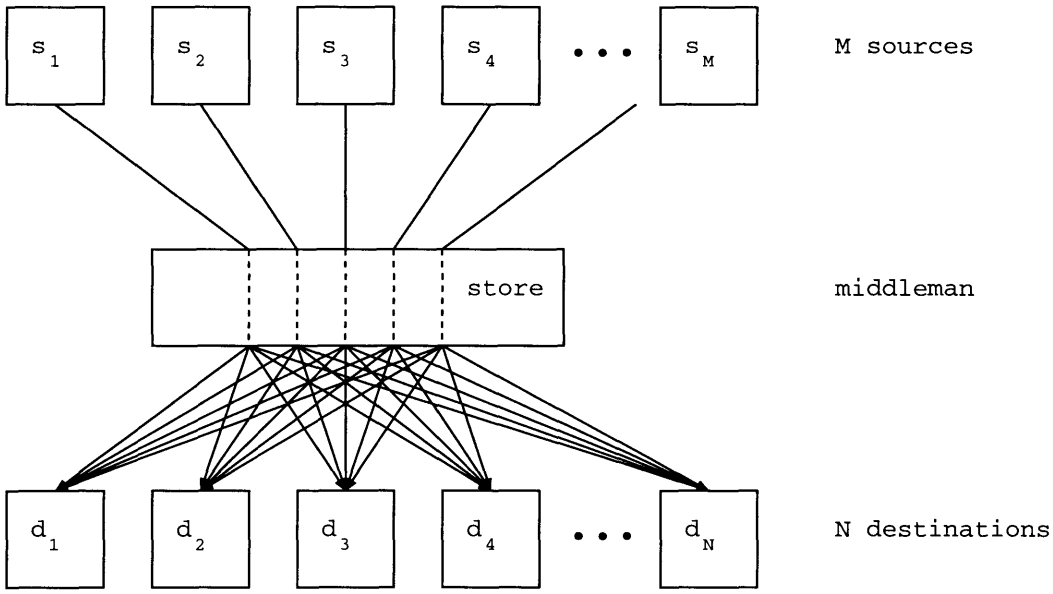

Figure 1-3: Shrink wrap distribution requires $M \times N$ links

Shrink wrap distribution maintenance links are very expensive. One factor driving up costs is the high degree of human interaction required. An end user will still need to unwrap a box, read instructions, install, and manually configure every single piece of software on their machine. These high costs are incurred by every end user in the distribution system.

The expense of transporting physical media drives up costs as well. Disks, CD-ROMs and other physical media require mass duplication, packaging, warehousing, shipment to stores, and other handling. This high distribution cost often translates into delays in software maintenance. For example, if a developer fixes a bug in a program, they could send a new copy on CD-ROM to every user via Federal Express; unfortunately that distribution channel is extraordinarily expensive. Instead new

versions slowly propagate through stores, or are not distributed at all.

Maintenance costs are further increased since store shelves will not necessarily stock the latest version of a program, possibly imposing additional administrative headaches for the end user down the road.

1.2.4 Central administration

Many academic environments, research institutions, and corporations use centrally administered software. Consider Project Athena, MIT's campus computer network. A student may simply log in to any workstation on campus and run Netscape Navigator. The newest, most stable version of the program will be executed. The student will not need to do any software administration whatsoever.

This is possible because the software system for Project Athena is centrally administered. Using a distributed file system, thousands of users and hundreds of computers access a central master copy of the software, which is maintained by system administrators. The central administrators perform all the labor intensive installation, configuration, and upgrading operations directly on the distributed filesystem. So instead of maintaining and upgrading thousands of copies of Netscape Navigator, only one copy must be slaved over.

This style of software and information distribution is typically available only to people connected over a high speed local area network. There is a lot of interest in popularizing and expanding centrally administered systems from beyond their traditional domains. The current concept of "Network Computers" being developed by industry seeks to expand centrally administered systems into the domain of business software.

Central administration requires only $M + N$ maintenance links, but now some of the links are automated and cost much less. (See figure 1-4) Central system administrators wishing to provide M software programs must configure and maintain each one of them on the system. The N end users, however, need only one automated

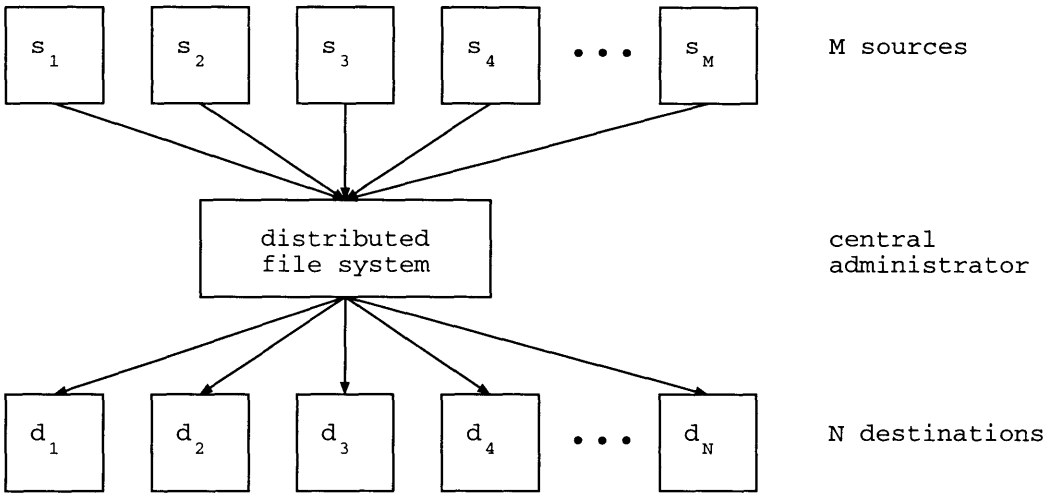

Figure 1-4: A centrally maintained system requires $M + N$ links

maintenance link apiece to the centralized repository.[4] End users are not required to even be aware of configuration settings, upgrades, or other maintenance aspects of software.

The distributed file system/local area network solution is very dependent on economies of scale. Maintaining a large collection still requires a lot of manual labor on the part of administrators; fortunately this cost can be spread out amongst end users. The more end users, the lower the administration cost is per end user.

Consider six sample distributed filesystems or computing environments. They include the General Electric Advanced Technology Lab, the MIT Media Lab, the MIT AI Lab, a subset of MIT Lincoln Laboratory, Bose Corporation, and MIT's Project Athena. Most of these organizations provide an essentially redundant set of basic software, with some variations. The only organization which could maintain both a wide breadth and current versions of software was MIT's Project Athena. Athena's success in this regard is due to legions of capable administrators composed of students, faculty and staff. The approximately 15,000 users of Project Athena outnumber the other networks by at least an order of magnitude, enough scale to

[4]For MIT's Project Athena, this maintenance link takes the form of an ethernet connection costing $300. [35]

provide adequate support for system maintenance.

Unfortunately, all current centrally maintained systems, including distributed file systems, suffer from limited scalability. From a macroscopic point of view, many different central administrators at different sites repeat each other's work. (See figure 1-5.) Central administrators at Carnegie Mellon, MIT, and Cornell might all maintain the same popular software packages, duplicating each other's work, thus driving up costs.

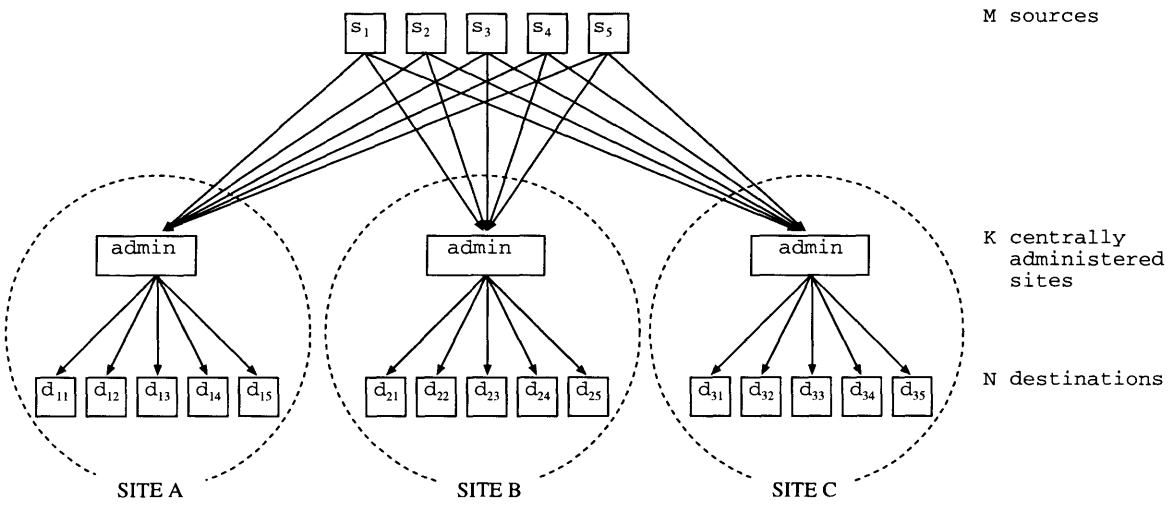

Figure 1-5: Centrally maintained systems require $(K \times M) + N$ maintenance links.

1.2.5 Package management

Package management is a tool for reducing the cost of a maintenance link. This technology can be applied to all distribution methods so far. The idea is to make it as simple as possible for a user or administrator to install or upgrade a piece of software. A package manager is a program that does housekeeping for local software. It keeps track of which files are where, where they came from, and which files depend on other files.

By employing package management, a lot of the grunge work of the installation process can be automated. The locations of various program files and configuration

settings can be stored in a central machine-specific database. Often the package manager is built into the operating system (such as the Registry for Windows, or RPM for linux.) For example, to upgrade the sendmail program using the Red Hat Package Manager, all one needs to do is obtain the new sendmail package, then type **rpm -U sendmail**. The system will then place the files associated with sendmail in appropriate places and keep track of them.

While package management makes upgrading easier, it still does not eliminate the process. Typing a single command is a lot easier than reading directions and making decisions, but the administrator or end user still has work to do. They are responsible for discovering when an upgrade becomes available, obtaining the upgrade and installing it. Keeping track of this information requires human effort that directly scales with number of software packages maintained. Thus, maintaining a large collection of software can still be quite expensive.

1.2.6 Push technology

A hot topic in software distribution is push technology. Push technology refers to the ability to subscribe to a piece of software which is periodically updated automatically over the internet. The update replaces the older version without any participation from the user.

Push technology is in the early development stage, with several competing commercial implementations and no common standards. One promising implementation is called Castanet. Castanet provides a very scalable distribution system, which supports only very limited types of software.[5]

Push technology allows a dramatic reduction in the cost of a maintenance link, since upgrades are automated. However, push technology does not reduce the total number of maintenance links. If five end users want to subscribe to five different software packages, there will still be twenty-five maintenance links.

The main cost to a single end user resides in locating and selecting the software

[5]Castanet only distributes programs which excute on the Java platform, and HTML format web pages. Many of the Java programs must be modified to explicitly support Castanet. [2]

they wish to use. These costs become significant when collections become large; locating and installing 10,000 different pieces of software requires a lot of human effort. The main cost in the distribution system stems from the fact that all of the end users are repeating each other's work.

1.2.7 Summary of distribution technologies

Current distribution technologies can be evaluated for two characteristics — their reduction of the number of maintenance links, and their reduction of the cost of each one. Table 1.2 summarizes the approaches of current distribution technologies.

technology	*number of maintenance links*	*number of non-automated maintenance links*
download from web	$M \times N$	$M \times N$
shrinkwrap distribution	$M \times N$	$M \times N$
central maintenance	$(K \times M) + N$	$K \times M$
"network computers"	$(K \times M) + N$	$K \times M$
package management	$M \times N$	$M \times N$ (semi-automated)
push technology	$M \times N$	0

Table 1.2: Focus of distribution technologies

Figure 1-6 shows the relative performance of each technology. The chart plots each technology with the relative number of distribution links along the vertical axis and the relative cost of each one along the horizontal axis. The ideal operating region is towards the origin where maintenance is cheap and administrators don't have to do a lot of redundant work.

Three common distribution technologies skirt the outer edges of the graph. Push technology makes administration work very inexpensive, but it is extremely redundant. Shrinkwrap distribution with central administration removes a fair amount of the redundant work, but what's left is expensive. Shrinkwrap distribution alone inherits the worst of both worlds!

Some distribution technologies can be combined to provide improved performance. Centralized maintenance tends to lower the number of maintenance links for a particular technology. Centralized maintenance is usually done on a limited scale, requiring

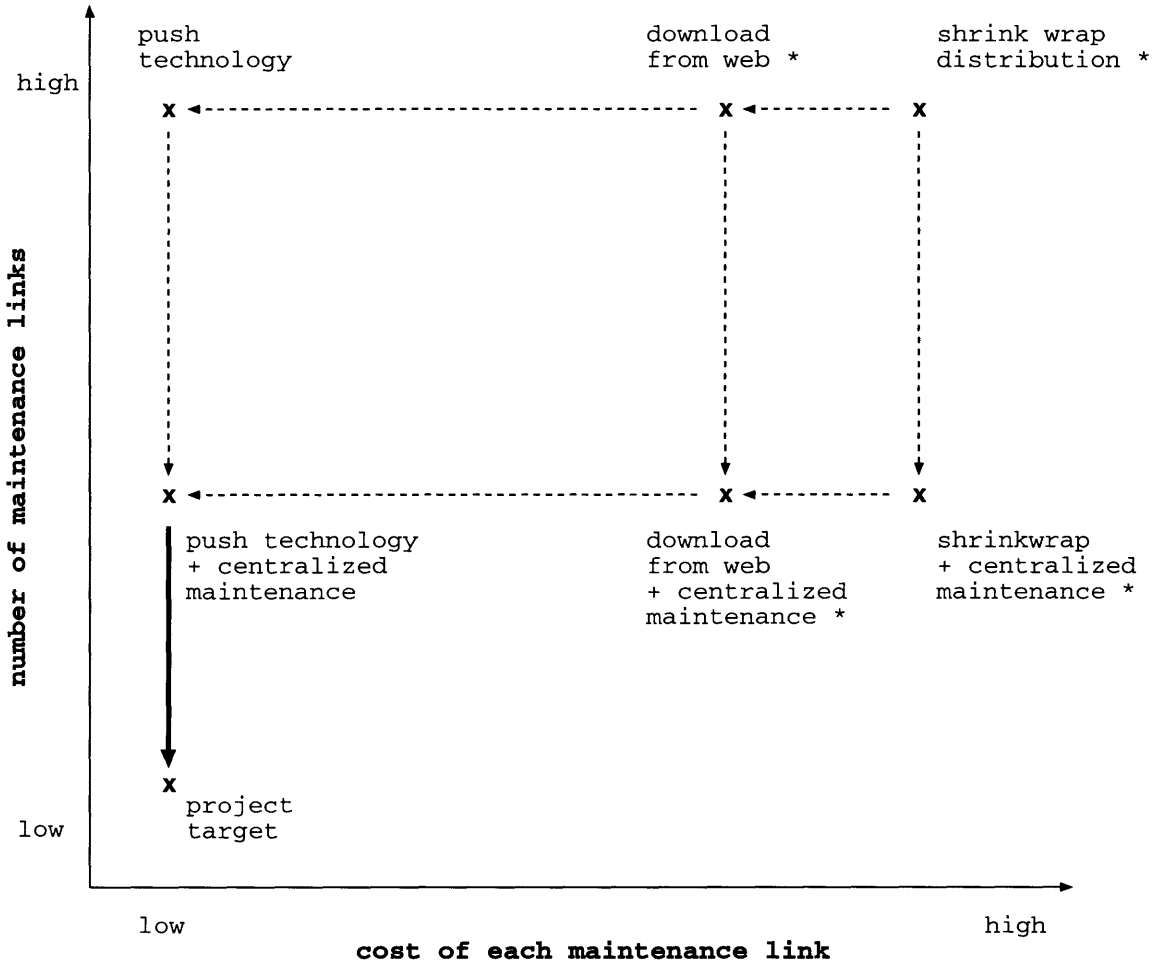

Figure 1-6: Relative performance of distribution technologies. A star (*) denotes distribution technologies which can be shifted slightly to the left by applying package management. The dark line represents the focus of this thesis.

duplicated efforts at different centrally maintained sites. Downloading from the web and package management reduce the cost of maintenance links. The improvements offered by package management or downloading from the web can at most be minor since they require a human being to be aware of updates and initiate them. People are expensive, so costs are high.

A logical conclusion from this graph is to marry push technology with a centrally administered system. This combination would at least get us into the lower left quadrant of the chart, and is a fairly easy goal. However, we can do even better by acknowledging that there is a lot of redundant work between centrally administered systems, and eliminating it. This improvement represents the focus of this thesis.

Older software quickly depreciates in value or expires. As discussed in section 1.1, maintaining current versions and proper configuration is critical for bug elimination, security, interoperability, hardware compatibility, and general happiness. The expense of inadequate maintenance is enormous. This might seem surprising since the current marketplace is filled with people running older versions of programs. Only a very small percentage of computer users keep on top of the latest releases of software; everyone else exposes themselves to the problems associated with poor maintenance.

The reason can be measured in dollars. Running version 5.003 of a program rather than version 5.002 can make an important difference, but the maintenance cost is too great. With current distribution systems, the cost of keeping current (i.e. maintenance and configuration) is even higher than the cost of being out of date.

1.3 The ideal world

An end user should be able to execute Netscape Navigator (or any other program) and always run the most current and up to date version. The user should not have to do any software maintenance or upgrading. The user would be able to run the program from any computer at any location with or without a network connection. Extended disconnected operation would be possible. The software would be reliable and available 100% of the time. Absolutely no modifications whatsoever would be

required for the Netscape Navigator program itself.

Furthermore, this functionality should be available for every program, including the operating system and the distribution system itself. In no case should the newer version of a program be detrimental; the computer should not spontaneously self-destruct as time passes. The user should be able to run a wide range of software in this fashion. Individual programs should be both immediately and easily available.

This functionality should work for any sort of information or software. If the information in question is an electronic newspaper or a military deployment schedule, the most up to date copy must always be the one available. The distribution system would not require redundant administrative work by anyone, and all administration would be very inexpensive. All of these features should be completely transparent to use. This thesis is dedicated to making this ideal world as real as possible.

Chapter 2

Design

2.1 Design Overview

2.1.1 Design goals

This design provides a framework for minimal cost software distribution and administration. The main focus is to take advantage of automation and economies of scale.

2.1.2 Environment

No design work exists in a vacuum. It is critical to consider the environment for which a system is being designed. Such a design assumes that individual computers do not operate in isolation, but can be connected to a global network (the internet) at least intermittently. Further, we assume that local storage and network bandwidth are limited resources. Finally, we assume that the wealth of software and information available far exceeds both bandwidth and local storage capabilities. These assumptions are currently true for most computer systems and are not expected to change in the near future. While computer technology has advanced rapidly, these basic relationships have been self maintaining.[1]

[1]Software demands tend to rise to the occasion to meet increased hardware capability. For example, an increase of local storage technology will often be accompanied by increased sizes of computer programs. As system administrators occasionally say, "You can never have too much memory/speed/disk space/bandwidth."

We assume that automation in distribution can be provided by push technology. We assume economies of scale in maintenance are possible because many people in the world run, or would like to run, similar sets of software. These assumptions are both reasonable and unlikely to change with foreseeable advances in technology.

2.2 Design Issues

Design criteria include survivability, mobility, checkpoints, economics, access control, scalability, version skew, push vs. pull, redundancy, network efficiency, caching, user vs. system space, platform management, ease of use, and closure.

2.2.1 Survivability

Section 1.1.6 pointed out that software maintenance is necessary for survivable systems. Thus, a low maintenance software distribution system provides a critical resource. Without software updates, a computer system can be compromised; therefore the distribution system itself must be survivable. The distribution system essentially becomes a single point of failure.

The distribution system must continue to function despite a wide variety of component failures. Some component failures might be human induced, such as from misconfiguration or a deliberate information warfare attack. Other component failures might be caused by mechanical damage due to disasters such as malfunctions, power failures, earthquakes, or nuclear explosions. The system must have enough robustness and redundancy to continue to function in spite of such faults. Functionality should degrade gracefully as more and more components fail.

Replication of data provides a degree of survivability.

2.2.2 Mobility

A computer should not have to be continuously tethered to a network to function. Mobile users will often have an intermittent connection to the network, as they move

from place to place. Mobile users should be able to connect to the network from any location and still make use of this software distribution system.

Varying levels of connectivity need to be accommodated, including low bandwidth systems. Mobile users are often faced with low bandwidth connections such as a modem or a wireless connection. These mobile users should still be able to run programs when their bandwidth drops to zero. This design criteria also applies to non-mobile users with intermittent or low bandwidth connections.

By using a local cache, programs are available even during disconnected operation.

2.2.3 Checkpoints

An automated software distribution system allows bug fixes and minor improvements to flow automatically to all users. But what happens when someone pollutes the source? If someone dumps toxic waste into the water supply does everyone downstream get poisoned?

Sometimes users might not want their software programs updated automatically. A new version might be incompatible with previous work, untrusted, untested, unstable, dangerous, or otherwise undesirable. What if the newest version of Microsoft Word contained a malicious virus? Will compromise at a single point (like Microsoft Corporation) quickly contaminate all computers everywhere? What keeps the bad from spreading just as quickly as the good? No amount of authentication or code signing can save the day if a new version of a program happens to be malicious.

With current distribution systems, the only protection afforded in this arena is the inefficiency of the distribution system. If the new version of Microsoft Word is actually a Trojan horse,[2] it might take years for everyone to upgrade to the new copy and become infected. An efficient but naive design might quickly spread the malicious program, introducing the possibility of catastrophic failure.

A more intelligent design allows breaks to be placed in the distribution system at arbitrary points. The flow of updates can be stopped at these checkpoints and

[2] A Trojan horse is a malicious software program which masquerades as a benevolent one.

examined for problems. If the updated version is approved, it can be supplied to the next point in the distribution chain. If the updated version is unacceptable, the older version will continue to be supplied from the checkpoint. Continuing with the river metaphor, we'd like to be able to place shut off valves at any point in the water supply.

The notion of a value added middleman makes checkpoints possible.

2.2.4 Economics

How might commercial software licensing fit into this system? Will non-repudiation be a valuable feature?[3] Will software be paid for on a limited time subscription basis? If there are middlemen in a distribution system for commercial software, how might they be rewarded for their services?

Economic issues are beyond the scope of this design.

2.2.5 Access Control and Authentication

The right information needs to go to the right people. From the software or information providers' point of view, this is an issue of access control. Only paying customers should be getting commercial software. Access control is essential if expensive software is going to be distributed globally.

From the user's point of view, authentication is a major issue. Information receivers need to make sure the goods they are getting (or buying) are authentic. Authentication becomes a crucial issue for those worried about the security of their systems.

Today, most systems handle authentication and access control by maintaining a centralized database of users, a password system, and a series of cryptographic tools designed to verify identities and restrict access to material. Larger authentication systems make use of digital signatures and authentication certificates. [39, 21]

[3]Non-repudiation means the receiver can prove that the sender transmitted a particular piece of software or information.

This design uses PGP to provide digital signatures for authentication.

2.2.6 Scalability

This system must be scalable to the millions and higher. Ideally, everyone running software on the globe should be able to participate. When Microsoft Corporation makes a minor bug fix to Microsoft Word or Internet Explorer, that change should propagate to everyone who owns a copy of the software package, with no fuss whatsoever. The system might also disseminate military information to soldiers. For example, every soldier in NATO might need to receive a daily copy of a given weather map. Scalability into the hundreds of thousands (or millions) is required. Economies of scale can only occur when there is scale.

The base technology of this design, Castanet, is scalable.

2.2.7 Version Skew

Everyone should have access to the latest programs or information. Version skew occurs when someone has an older copy of a piece of software than they should. For instance, if the current version of Internet Explorer 3.02 has been released for six months, and a user is running 3.01, the version skew is six months. In this case the difference is significant because version 3.01 has a security hole which can compromise an end user's system. Version skew is an enormous problem in conventional distribution systems.

Ideally there would be no version skew whatsoever; however that is not always possible, even in a fully automated system. Another possibility is a controlled and limited amount of version skew. A related issue is whether it is important or worthwhile to retain old versions of programs or files, and if so, how many of them. This design keeps at least two copies of software around, so that one can be run while another is updating.

This design is automated to reduce version skew, and middlemen can increase version skew when desired as in section 2.2.3.

2.2.8 Push vs. Pull

The terms "push" and "pull" are commonly confused. In both cases, information is sent from a server to a client. If the transaction is initiated by the server, it is push. A transaction initiated by the client is pull. Broadcast radio is push, browsing web pages is pull. Unfortunately, systems such as Castanet, in which the client periodically polls a server, are often labeled as push.

Possible designs include push, pull, or combinations thereof. Push allows for the use of efficient network protocols such as multicast. [17] Pull allows the client to determine update schedules. A combination could either be efficient or overly complicated.

This design uses pull, since you can't push information at a disconnected machine.

2.2.9 Redundancy

Redundancy can be a mixed blessing. Redundancy means keeping duplicate copies of software, doing duplicate work, or maintaining duplicate systems. If one system fails, there are duplicates to fall back on. Thus, redundancy can provide high availability and prevent single points of failure. On the other hand, redundancy can be expensive. Maintaining unnecessary duplicates increases costs.

Current software distribution systems require a lot of duplicated effort at every stage. For example, in shrinkwrap distribution end users duplicate each other's efforts while maintaining their own machine. An error by a particular end user cannot harm more than a single machine. Shrinkwrap distribution provides high redundancy but also increases costs to the point of compromising the quality of software maintenance. In a better design, redundancy is adjustable to meet the needs of a particular application.

This design employs the concept of value added middlemen to provide adjustable redundancy.

2.2.10 Network Efficiency

Network bandwidth is an expensive resource and must be used efficiently. Even as optical fiber capacities exceed terabits per second in the laboratory, I expect software bloat to easily keep pace with transmission capacity. Non-network distribution methods can also be used, but they too must be cost efficient.

Wireless networks have significantly less available bandwidth, often supporting only tens or hundreds of kilobits per second. Network connections over a phone line can be equally slow. Caching strategies can drastically reduce the amount of information being sent across a network. Network efficiency is currently a serious problem with the world wide web. [26]

One way to dramatically increase network efficiency is through caching. Caching allows the same data to be stored at multiple points in the a network, eliminating repeated, redundant transfers of information from one point to another. Another approach for widely disseminating information is the multicast networking protocol, which reduces redundant information transfer at the protocol level. [17] Multicast is only efficient when there are multiple clients receiving information simultaneously.

This design uses caching to increase network efficiency.

2.2.11 Caching issues

The local storage space on the end user's machine is limited compared to the amount of software that is available. What happens when the user wants to run seven programs, and there is only room for six of them on the local drive?

One option is to force end users to do their own manual cache management. Manually deleting and installing new software is possible, and can be efficient from a bandwidth standpoint, but creates an extra burden for the user.

Another possibility is an automated cache system that works on the least recently used (LRU) principle. When space runs out, the dustiest[4] piece of software is deleted and new software is loaded. Other automated cache management algorithms, such

[4]Metaphorically speaking, of course.

as random replacement and flush when full are possible; a particular algorithm's efficiency will depend on the end user's patterns of use.

A third alternative matches the features of both. An automated, least recently used cache management system takes care of making space on the user's system. The user has the ability to specify a certain software package as sticky and lock it in the cache. An even more sophisticated approach would allow the user to prioritize their software. The system would attempt to maintain higher priority software in the cache, similar to the hoarding technique used in the Coda filesystem. [30]

This design uses the LRU principle and optionally allows the user to lock individual pieces of software in the cache.

2.2.12 User space versus system space

Local storage is considered an expensive resource and must be used efficiently. Is this local storage in user space or system space? The question has particular significance on a multiuser machine where storing multiple copies of programs in user space can be extremely inefficient.

Imagine two users, both running Matlab on the same Unix workstation, and storing software in user space. They would have to keep separate copies of the software in their home directories. Home directories are finite and not really the place to be storing huge amounts of software. Single user operating systems such as MacOS do not have this difficulty. Since all local software is accessable, there is no incentive for multiple users to install duplicate software on a single user machine.

From an implementation standpoint, user space is much easier to work with than system space. System space requires root privileges and may require modification to operating systems, [28] which are beyond the scope of this project. However, system space provides better performance for multiuser machines, and in an ideal implementation, system space would be used.

This design places the cache in user space.

2.2.13 Platform management

This distribution system should be able to support any number of operating systems and platforms in a transparent fashion. For lack of a better term this process will be called platform management. The distribution system must make sure the right versions of programs or information go to the right machines. All platforms and operating systems need to be supportable.

Some machines can support multiple architectures; for instance the same computer might run Solaris, Mac, and Java executables. If a computer can support multiple architectures, the end user might have a preference list. (Try to get the Solaris executable, otherwise take the Java executable.) Or the end user might wish to install multiple executables. (Get the Solaris and the Mac executable.)

This design supports multiple platforms transparently.

2.2.14 Ease of use

A software distribution system should be easy to use. Ideally the end user will not even know it exists. The user should be able to run programs, and not worry about where they reside, maintaining them, or where they came from. They should not have to adjust any parameters or make any decisions if they do not wish to. Customization should be permitted in an intuitive fashion, if desired.

End users can be completely ignorant of distribution and maintenance tasks performed by middlemen.

2.2.15 Closure

Closure means that the software distribution system can update itself. As mentioned in section 1.1.6, a program is only survivable if it can be updated. Since the software distribution system is a critical resource, it is likely to be the target of an attack. As security holes are discovered, it will be necessary to update the distribution system.

The base technology, Castanet, has the ability to update itself.

2.3 Tradeoffs

Not surprisingly, some of these design criteria are in direct conflict with each other. Figure 2-1 illustrates where tradeoffs occur.

	survivability	mobility	checkpoints	access control	scalability	version skew	push vs pull	redundancy	economics	network efficiency	system vs user space	caching	platform management	closure	ease of use
survivability															
mobility						X									
checkpoints															
access control					X										
scalability				X											
version skew		X													
push vs pull															
redundancy										X					
economics															
network efficiency								X							X
system vs user space											X				
caching										X					
platform management															
closure															
ease of use										X					

Figure 2-1: Design tradeoffs. An **X** denotes two desired features which are in conflict.

2.3.1 Version skew, mobility, push/pull

The amount of version skew, or delay a user might experience in receiving current information, is heavily tied to the bandwidth of a network connection. This makes sense since updating software requires bandwidth; a relationship especially important to mobile or intermittently connected users. More bandwidth allows software to be transferred with less delay. Thus, higher bandwidth implies quicker updates and less version skew.

Can the end user run programs if the network fails? If they are in the Wrangell St. Elias National Park in Alaska and the nearest network connection is hundreds of miles away? If so, is the end user willing to accept a delay in updating software? In this design the answers to all those questions are "yes."

All distribution is initiated via client pull. This is the only way for an intermittently connected machine to reliably retrieve information. Pushing information at a disconnected machine is unlikely to succeed. While the software source may set the default schedule and frequency of pulls, the software receiver may override that schedule.

This way, a mobile user can ignore any scheduled updates during disconnected operation. Upon re-connection to the network, updates may resume. The end user is permitted to override the update schedule, skipping updates. The choice, however, is limited to keeping whatever is installed or moving to the newest version of software. Managing multiple versions of programs and their dependencies is beyond the scope of this thesis. In general, version issues are handled by the update source, not the update receiver.

2.3.2 Access control, scalability

The traditional method of access control involves registering everyone on a system with an identity maintained in a central database. Usually a password is used to authenticate a user. Unfortunately, a single system authentication is extraordinarily difficult in a large system. Thus, individual resources will be required to perform their

own authentication and access control. Large scale and distributed access control and authentication is an active research issue for the world wide web.

2.3.3 Ease of use, efficiency

To attain maximum efficiency, the end user would be able to decide exactly what is stored locally and what is available from the network. While an automated system of cache management might be able to do a reasonable job, no system can anticipate a user's actions. The strategy of automated, but user assistable cache management is used extensively in the Coda distributed filesystem, [34] which was designed to allow for disconnected operation.

2.4 Focus

The primary design focus for this thesis is scalability. Secondary considerations include ease of use, checkpoints, and platform management. Less attention is placed on economics and authentication.

2.5 Putting it all together: an example

Consider the case of the MIT Lincoln Laboratory internal network. MIT Lincoln Laboratory is a military oriented research institution with about two thousand employees. These users are connected on a fast local area network which is connected to the internet through a firewall. In this scenario, the goal is to give them access to the following software: Framemaker, a word processing program; Matlab, a mathematics processing program; and some internal MIT Lincoln Laboratory specific software or information.

We'd like the original developers to maintain the commercial software. For instance, any bug fixes added to Matlab from the authoring company, Mathworks, Inc. should propagate directly to the users as they become available. In addition, we desire efficient use of network resources, and easy access for users. Trust is extended to

the software developers, but no others.

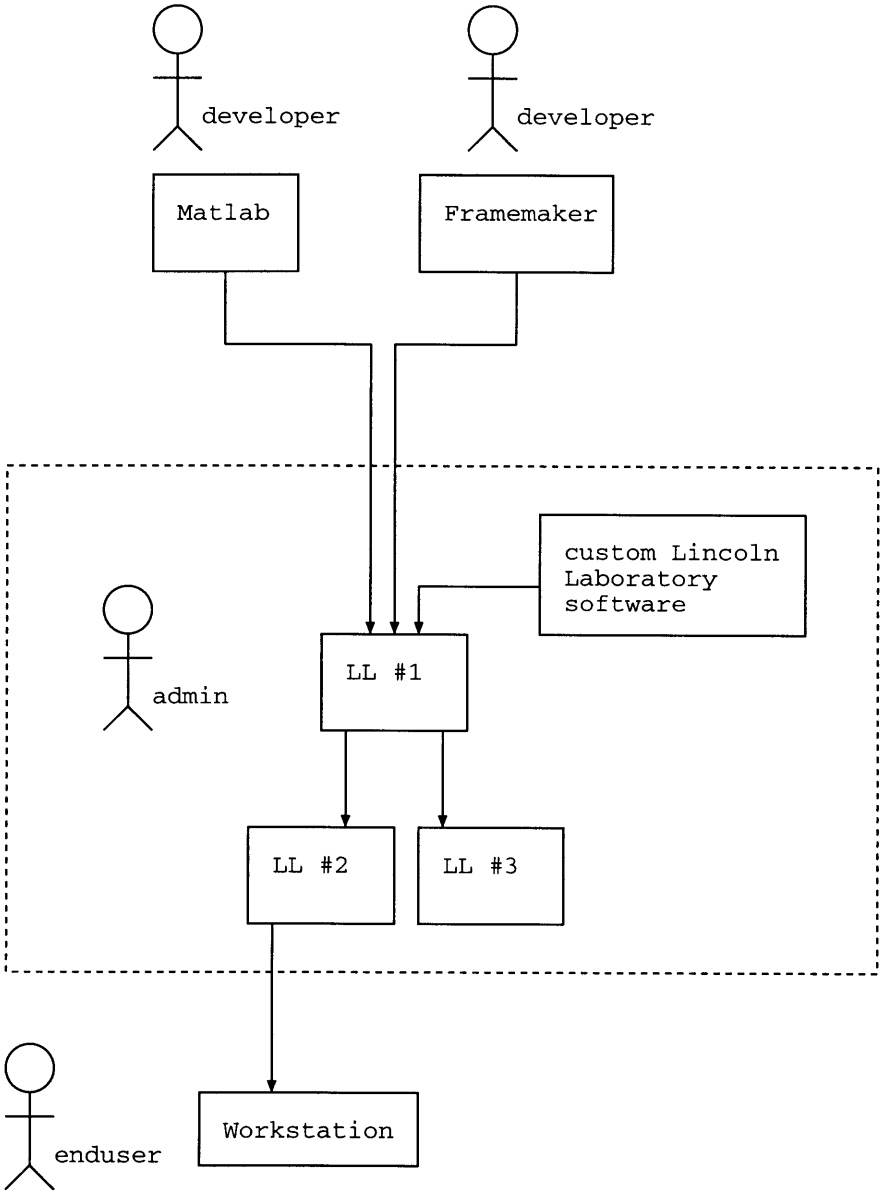

Figure 2-2: Trickle down software at MIT Lincoln Laboratory

First we set up a computer (with a really big disk) called Lincoln Labs #1 (LL1). LL1 would subscribe to the outside software sources. Copies of Matlab and Framemaker would be downloaded directly off the manufacturer's sites. This would be a subscription-based connection; any changes the developers make would be replicated on to LL1. In addition, custom MIT Lincoln Laboratory software or

information would also be placed on to the machine. (see figure 2-2)

LL1 is now a very valuable computer and a model machine. It contains essentially all the software that the lab denizens would want to use. Thus the contents are replicated to a few other machines, LL2 and LL3, which are distributed, redundant file servers. This replication provides a certain amount of redundancy to increase reliability, robustness, provides performance benefits for end users via parallelism.

An end user might connect their workstation to LL2. When the user decided to run Matlab, a copy would be replicated off of LL2 and onto the user's local filesystem.

Such a setup is not very dissimilar to having all these programs stored on a distributed file system at MIT Lincoln Laboratory. In either case, the end user does nothing to maintain software. In this particular situation, the information systems department at MIT Lincoln Laboratory would maintain only their own information and commercial maintenance would occur automatically. This makes it feasible for a small organization to provide a much wider range of software, rivaling perhaps MIT's Project Athena in breadth.

What happens in the case of failure? If the user's machine fails, then the user is out of luck. The user could go down the hall and run software off of a different machine; it really depends on whether any personal data was stored locally. If the Lincoln Lab network fails, then the user will not be able to receive software upgrades. However, the user will still be able to run software that has been cached to the local machine. If LL2 fails, the user has a fallback to LL3. If LL1 fails, then MIT Lincoln Laboratory users will not receive software updates until operation is restored, but will still be able to run existing versions of software. If the Framemaker master site fails, then there won't be upgrades to Framemaker. However, all of MIT Lincoln Laboratory will still be able to operate with the local copy. Presumably the Framemaker master site can also maintain some redundancy.

Chapter 3

Implementation

3.1 Technologies

3.1.1 Basic components

Several technologies are used in the implementation. Since one goal of this proof-of-concept implementation was to write as little new software as possible, the implementation builds on existing components. Availability, extensibility and openness were used as selection criteria. To a degree all of these component software choices are arbitrary and could be replaced with alternative technology.

Castanet, a product of Marimba, Inc. was chosen for a replication tool since it is efficient, programmable, and has a strong technology base. Its main drawback is its proprietary nature and lack of stability. Castanet provides automated file system replication and scalability to the system. Castanet 1.0 does not provide access control; a later version will support that capability. Castanet does not provide support for multiple platforms.

PGP, a freeware cryptography program, provides public key technologies for authentication and access control. PGP is highly respected and stable; its drawback of non-exportability is not an issue for this proof-of-concept system.

The **Athena locker model** was chosen for managing executables for multiple platforms. This model has a proven background on MIT's Project Athena and other

large information systems. The Athena locker model provides a standard directory structure for platform management and executable organization.

Java programs and shell scripts were used to glue these components together as necessary. Shell scripts are the simplest way to automate tasks in the Unix environment. Java was chosen where it was necessary to interact with Java based Castanet. Java programs were used in priority of shell scripts due to Java's portability.

3.1.2 Castanet in detail

The most sophisticated component for this project is Castanet. A description of Castanet is important to distinguish the work of this thesis from the underlying system. Castanet provides three essential components, the transmitter, the receiver, and the repeater. They are the server, client, and a server/client combination, respectively. These components are used to distribute and maintain channels.

For example, suppose an end user wishes to play the daily crossword puzzle using the Castanet system. First, the end user would install a tuner on his computer. The tuner provides a graphical user interface that allows the end user to subscribe to channels. Then the end user would have to find a crossword puzzle channel. Once the channel is located, the end user would subscribe. This would replicate a crossword puzzle from the transmitter to the his local storage. As long as the end user is subscribed to the channel, his tuner would periodically poll the transmitter to see if there were any new crossword puzzles available. If there were, the tuner would download any new files and update the local copy.

From the end user's perspective, once he subscribed to the crossword puzzle channel, he would always have a reasonably up to date version of the game. Castanet only supports programs written in Java. To the end user, a Castanet channel would feel quite similar to a Java applet run through a web browser. The main difference is that applets are not stored locally while Castanet channels are — thus bandwidth is only required for updates. Castanet updates are extremely efficient, both in only transferring files that are not already present, and recognizing when multiple channels use identical files.

Repeaters provide scalability for Castanet. Repeaters are combinations of transmitters and tuners placed throughout the network. This allows caching and scalability. Thus the person subscribing to the crossword puzzle might do so from a local repeater rather than the master transmitter. Castanet repeaters do not provide modification, organization, or any other form of intermediary services.

The Castanet system is very extensible, at nearly every stage. The tuner can run channels which performa administrative functions. The transmitter can run "transmitter plugins" which can also perform administrative functions. Since each component is programmable, a channel's content can be tailored to each end user. This provides a framework to add what Castanet is missing as a distribution system.

There are a lot of missing features. These include tools for organizing software collections, access control and authentication, multiple platforms support, support for middlemen, and the concept of automated cache management for the tuner. All of these things are required for the design presented in chapter 2, and have been implemented in this thesis with varying degrees of success.

3.2 Basic distribution systems

A complex system can be created using primitives, means of combination, and means of abstraction. This section presents the essential building blocks of a distribution system, and describes how to combine them into a more complex system.

Four basic examples are presented.

1. Replication from source A to destination B
2. Combining two sources
3. Adding a checkpoint
4. Replicating a large source

3.2.1 Replication from source A to destination B

Consider the following scenario. A developer wishes to distribute the program xclock. This is a simple program that displays a clock on the screen as shown in figure 3-1. The end user wishes to use that program and automatically receive new versions as they become available. (See figure 3-2)

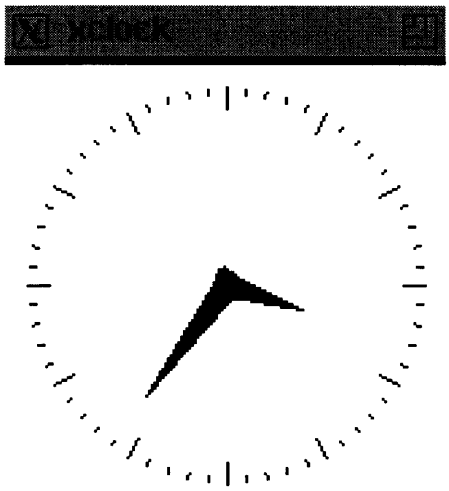

Figure 3-1: The xclock program

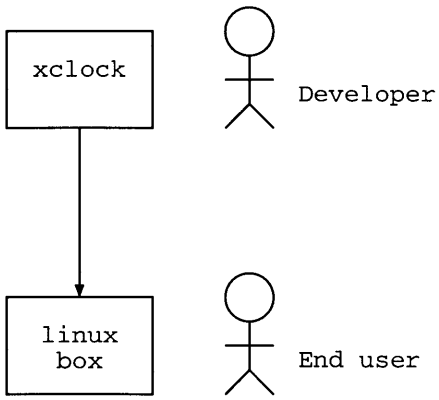

Figure 3-2: A two person distribution system

There are xclock executables for several platforms, including linux, Solaris, and IRIX. The developer prepares a standard directory structure, which contains the ex-

ecutables for each supported platform (see Appendix B). The developer then creates a software channel called *xclock*.

An end user on a linux workstation subscribes to the *xclock* channel. The appropriate xclock executable (in this case linux) is replicated onto the user's machine and can now be executed from the command line. As new versions of *xclock* are created, updates are automatically replicated onto the end user's filesystem.

Castanet 1.0 provides automated file replication but little else. Custom code does the work of adjusting the end user's PATH variable, making sure replicated executables are still marked as executables, and deciding which files get replicated.

Let's examine this process in more detail. The directory of executables is packaged up as part of a Castanet channel called *xclock*. That channel also contains two custom programs to manage the replication process. One runs on the server (or transmitter), and one runs on the client's machine. The server side program is called a transmitter plugin,[1] while the program that runs on the end user's machine is a special Castanet channel.[2] The special channel is named `payload` in this implementation.

Using Castanet Publisher, the developer places the software channel onto a Castanet transmitter. When a user subscribes to a channel, the transmitter plugin is executed on the server machine. The subscription request (along with all subsequent requests for updates) is intercepted by the transmitter plugin.

The transmitter plugin queries which platform the client prefers to receive. In this example, linux is the preferred platform. The plugin determines that the linux executables should be the only files replicated and makes sure to transfer only the appropriate subdirectories. If there were shared files between platforms, (such as common libraries or data files between platforms) they also would be replicated.

In addition to the linux subdirectory, the program `payload` is downloaded and executed on the client machine. This Java program attends to platform dependent installation on the end user's machine. In this example, `payload` adjusts the user's

[1] A transmitter plugin is a program that runs on the server side every time an update occurs.

[2] This channel is special, in that it performs administrative tasks for other channels. Incidentally, the tuner itself is a Castanet channel which performs administrative tasks on other channels.

path, and modifies the file permissions so that the operating system recognizes xclock as an executable.[3] The `payload` program is platform dependent and does not take advantage of Java's portability; thus it could also be implemented as a native executable.

The actions of setting file permissions and changing important global variables (such as the executable path) are highly privileged operations. Currently, Java programs automatically invoked by Castanet are not permitted to make such changes, due to possible security risks. In a future version of Castanet that supports code signing, `payload` could be given the necessary permissions to do its work. For this implementation, the security manager for the Castanet tuner has been disabled.

Finally, now that replication has occurred, the end user may type "xclock" and watch the program execute on the screen.

Access control and version skew deserve further discussion. In the *xclock* scenario, there is no access control. Anyone who wishes to subscribe to the *xclock* channel is permitted to do so. If restricted access were desired (for example, if commercial software was being distributed, rather than the freely available xclock program) the transmitter plugin would only authorize replication if a client could prove they were a licensee.

One way for a client to prove their identity is by supplying an identifying certificate. This might be as simple as a document stating that the software had been licensed. This license agreement would be digitally signed by a trusted authority, either the software providers themselves or a trusted third party.

As always, an access control system must be carefully designed and implemented to maintain security. In the sample implementation, the client presents a document stating its identity and permission to access a particular channel; essentially a license agreement. This document has a validating PGP signature from the provider, which is checked by the plugin stage. This design has many weaknesses (for example, it is vulnerable to replay attacks) and only represents a rudimentary access control system.

Any form of access control may be substituted in place of this proof-of-concept

[3] Castanet 1.0 unfortunately strips file permissions during replication.

implementation, including the access control system to be built into a future version of Castanet itself. Another possibility, not implemented, is for software to be distributed in an encrypted or otherwise unusable form, which may only be unlocked with a key licensed from the software developer.

Another issue is how to handle replication while an executable is running. In this implementation, the client periodically polls the transmitter to see if there is a new version available. If a new version is detected, the following strategy is used.

1. copy current version to a temporary location
2. incrementally update the copy
3. move aside the current version
4. move updated copy into place

This allows the possibility of updating xclock while it is being run, without causing interruption. Each time a new version is received the old version is moved to an alternate location; eventually creating a large collection of old software lying around. The `payload` program is responsible for managing this process.

The user may only access the latest version of the software, unless the program is already running. To the user, updates appear to happen as dictated by the replication schedule, except that an update never interrupts a currently running program. This can be maintained indefinitely, or at least until local cache space is exhausted, in which case older versions of programs are deleted.

3.2.2 Combining two sources

Suppose now that a middleman is compiling a collection of programs that decorate a computer screen. She notices that the developer from section 3.2.1 is supplying *xclock*. A different developer is supplying *xeyes* which is a decorative program in which a pair of eyes follow the mouse cursor around the screen. See figure 3-3. This middleman would like to combine the two software programs and offer them together, providing one stop shopping for customers seeking screen decorations. See figure 3-4.

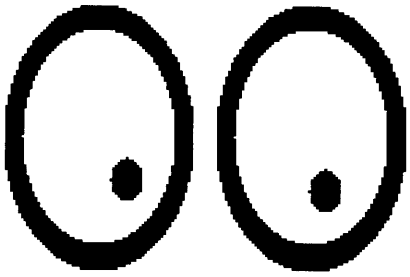

Figure 3-3: The xeyes program

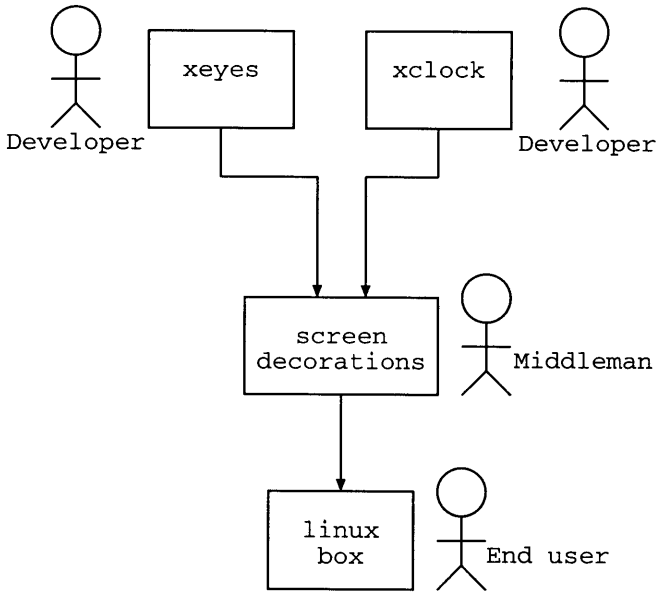

Figure 3-4: Distribution system with a middleman.

The middleman subscribes to both the *xclock* and *xeyes* channels. Instead of receiving the executables for a specific operating system, the middleman receives all the executables. The middleman requests this configuration by customizing a configuration file.

These two channels are mirrored by the middleman. In addition, the middleman creates a new channel called *screen decorations*. When the end user subscribes to *screen decorations*, they receive a program that quickly subscribes them to *xeyes* and *xclock*, unbeknownst to the end user! From the system's point of view, the end user is subscribed to *screen decorations*, *xeyes*, and *xclock*. All three channels are provided by the middleman.

From the user's point of view the only channel being subscribed to is *screen decorations*, which appears to contain two programs, xclock and xeyes. The end user is saved from having to discover and subscribe to two independent sources. The `payload` program and transmitter plugin again work to provide only the correct executables and to manage directories and files.

This process of merging two channels is crucial to the distribution system. Thus a custom program was written to automate the process. The name of the program is `merge-channels`.

3.2.3 Adding a checkpoint

Consider again the simple A to B distribution system in section 3.2.1. Again, a middleman is interested in getting involved in the distribution scheme.

This middleman is the National Institute of Standards and Technology (NIST), and is very concerned that the xclock program is virus free and safe to use in government installations. They wish to sit between A and B, checking to make sure the program is benevolent. After a thorough examination, the program is then passed on to B, with a stamp of approval attached ("Approved by NIST for your safety") Essentially this middleman wishes to stop the distribution stream, modify it, and then send it on its way. (See figure 3-5)

NIST, acting as a middleman, first subscribes to the *xclock* channel. Then they

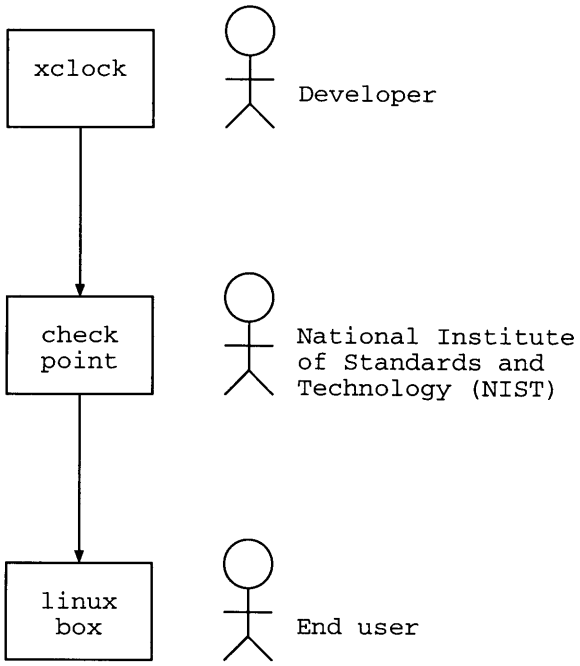

Figure 3-5: Adding a checkpoint

examine the program carefully, spending weeks looking over the code. Once NIST is satisfied, they add a certificate of approval to the channel and retransmit. The process repeats with each update of the xclock channel.

The program `merge-channels` is used to handle the process.

The modifications by NIST are either done by hand or they may automate the process themselves. Limited support for automation is provided in the extensible `merge-channels` program.

3.2.4 Replicating a large source

The previous examples have assumed very small software sources. The implementation can handle much larger sources, including sources larger than the local storage of the enduser. This is important because, in general, the amount of usable software may exceed local storage capacities.

Consider the case where a large collection of software is being distributed. MIT's Project Athena contains thousands of software programs stored on the Andrew File

System (AFS). (See appendix A) They have been organized by hundreds of people over thousands of hours. Both end users and perhaps other universities would like to subscribe to Project Athena and be able to run all of the programs.

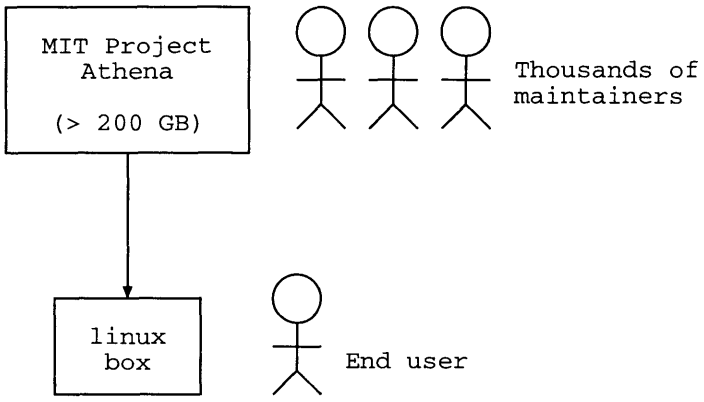

Figure 3-6: A large software source

Several issues are immediately raised. One is the name space problem. Every file must be uniquely specified, but with a large collection of software there are likely to be several files with identical names. Project Athena solves this problem by subdividing the collection into different logical units called lockers.[29]

For example, there are file lockers called "graphics", "scheme", and "6.034" which contain programs and directories for image manipulation, the scheme programming language, and files associated with the introductory artificial intelligence class which is called 6.034. The end user on an Athena workstation might type `add matlab` which attaches[4] the appropriate file locker, then type `matlab` to execute a program inside that locker.

The same logical division applies in this implementation. The different file lockers are converted to Castanet channels. One channel is called Athena, to which one may subscribe. As far as the end user knows, this is the only channel associated with Athena. Actually there are many channels, one for each for each file locker.

The Athena channel contains a `payload` that lets you access all of the other file

[4]This refers to adding an AFS volume to the directory hierarchy on the end user's workstation. Executables and man pages are added to the search path, and possibly other initialization occurs.

lockers. In the current implementation the end user is provided with a command line program where they can type commands like `add matlab`. This interface was chosen because it is familiar for current athena users.

Once the user types `add matlab` they are unknowingly subscribing to the matlab channel. If there is room, the channel is replicated to the user's computer system. If there is not room, space is cleared on the local disk and the channel is replicated.

On the transmitter side, there is a similar dilemma. A transmitter contains a local disk on which channels are stored, and access to MIT's large distributed file system. Since local storage is insufficient to store all of Athena, the transmitter must generate channels on the fly. Caching is also used.

This is significant because now Athena is more widely available than just over its distributed file system (AFS) and no longer requires end users to have a permanent network connection to run all Athena software.

3.3 Maintenance loops

Consider the following scenario. A student is working on his Master's thesis. The thesis itself is stored on a central campus computer system. The student wishes to edit the files while away on spring break on a laptop computer.

This type of collaborative setting, where a set of files can be maintained on both the laptop and the campus computer, can be accomplished through a maintenance loop. A transmitter and a receiver are installed on both the laptop computer and a campus workstation. A modification to the thesis from any point will eventually propagate to all systems on the loop.

This situation requires additional care. If a solitary student is editing the files, there is little danger of conflicting modifications. If multiple collaborators are working with the same files, they must be especially careful not to overwrite each others work. On a distributed file system, this situation is usually managed through semaphores or lock files.

A maintenance loop is much more difficult. By definition only one person may

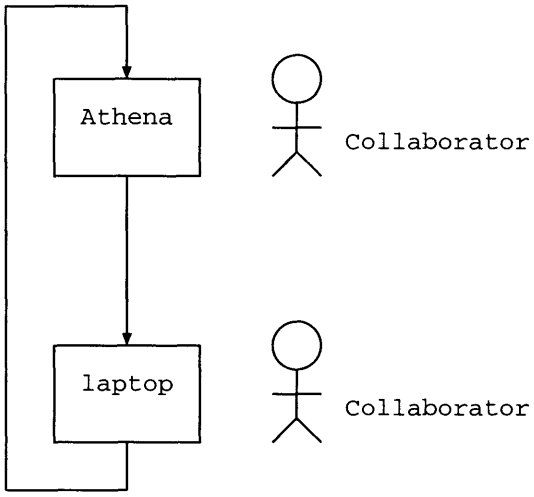

Figure 3-7: A two person maintenance loop

be granted a lock at a particular time. In order to assure unique access to files, one may still request a lock on a particular file. However, someone else might also have requested a lock at some other point in the maintenance loop.

The only way to determine if a person can receive a unique lock is to wait until the lock request has propagated completely around the maintenance loop. If there are no conflicts or existing locks in place, the lock may be granted. Unfortunately, the amount of time required to propagate around a maintenance loop (the period) can be quite long. This period will depend on the allowed version skew between each maintenance link and the number of maintenance links. If one of the machines in the maintenance loop can be disconnected from the network for extended periods (e.g. the laptop), the period can be indefinitely long.

Thus, uncoordinated collaborative file editing requires a tight maintenance loop in time and may provide unacceptable performance. Coordinated collaborative file editing (where the users coordinate amongst themselves not to edit files at the same time) is a more practical use for a maintenance loop. The ideal situation for a maintenance loop would be a single editor who wishes to work on their files across multiple systems.

Another possibility is to allow branching in the version sequence. Some version

control systems allow simultaneous edits without file locking, and conflict resolution upon check-in. This topic is beyond the scope of this thesis.

3.4 Implementation components

All of the custom programs written for this project are built on top of Castanet and written in Java. Source code for these programs may be found in Appendix C.

3.4.1 Payload program

When someone subscribes to a software channel, they will receive a program called `payload`. The `payload` program is written in Java and takes care of the platform dependent aspects of replication. The payload program works in conjunction with auxilary programs, such as shell scripts, on some platforms.

One issue that quickly arose is the incompatibility of different file systems. Symbolic links are handled differently between Unix, Windows, and Macintosh file systems. Permissions and access control conventions vary widely across file systems, even for Unix (NFS supports group permissions while AFS makes use of access control lists)

The `payload` program, in its Unix incarnation, is specifically designed to handle executable permissions, symbolic links, modifications to the executable path, and movement and copying of the replicated file system during updates.

3.4.2 Transmitter plugin

The transmitter plugin is responsible for platform management. When an end user subscribes to a channel, they may specify a preference for a particular platform and platforms. The tuner plugin makes sure that only the desired platforms are distributed.

3.4.3 The merge-channels program

The merge channels program is responsible for combining different Castanet Channels. This program allows a user to subscribe to multiple channels and provide a single channel; it handles updates automatically and may be extended for custom updates.

Chapter 4

Analysis

4.1 Performance Criteria

Efficiency can be measured in terms of how well network bandwidth, local storage, and computational resources are used. In addition there are more subjective considerations such as waiting time for a program to execute, how often a network communication channel is required, how easy disconnected operation is, and how much knowledge a user must have of the underlying system. From a systems perspective we would like to know the ultimate scalability of the system, and how performance is affected with a large number of users.

4.1.1 Physical resources

Network bandwidth availability has huge variation. Wireless systems run from zero bits per second to hundreds or thousands of bits per second and can change instantly depending on environmental conditions. Modem users communicating over a single standard telephone line are currently limited to 56 kilobits per second, with an absolute theoretical upper bound of 64 kilobits per second. An Ethernet connection provides 10 to 100 megabits per second, while ATM networks can provide over 100 megabits per second. The world record for sending information through a single optical fiber is over three terabits per second; however this is only in a laboratory

setting. [25]

Storage capacity in terms of magnetic disk drives, is also changing rapidly. A typical new personal computer includes a hard disk of approximately two gigabytes. Nine gigabyte disk drives are easily available from vendors on the order of one thousand dollars. [14] However, "Network computers" are being designed with no significant local storage at all.

Another resource is **processor capacity**. Processor speed is difficult to measure, and many benchmarks are available. In 1998, all new personal computers are expected to meet or exceed the processing capacity of a single Pentium MMX chip running at 200 megahertz, according to industry recommendations developed by Microsoft and Intel. [13] The fastest general purpose processor on the market has a 600 megahertz clock and employs limited parallelism. [8] Multiprocessor machines, containing two to six processors, are starting to enter the consumer market.

4.1.2 Resource demands on a single machine

Bandwidth is most stringent requirement since the design must support wireless or temporarily disconnected systems, including modem based personal computers. Thus, bandwidth considerations received priority when making design tradeoffs. The bandwidth required by a particular machine in the distribution system depends on many things.

First, consider an end user's machine. The end user will likely run several different software programs. One strategy is to receive software over the network every time it is used. A better strategy, from a bandwidth standpoint, is to cache software in local storage. Network bandwidth is only required for cache misses; i.e. software is requested that is not in the cache, or the software that is in the cache becomes stale.

Bandwidth requirements become to first approximation

$$B = rC + \frac{\tau S}{C}$$

B bandwidth

S aggregate size of software in user's repertoire

r average rate of update of that software

C local cache size

τ miss rate $\times$ access rate

This design attempts to keep material in the cache reasonably up to date at all times. Stale software is updated by the system on a regular basis. The first term, rC, refers to the amount of bandwidth required to update stale software in local cache. For example, if the software in the local cache changes 10% per year, and the local cache is one gigabyte, the updates will require 100 MB/yr of bandwidth. The larger the cache, the more bandwidth required, assuming the cache is full.

The second term, $\frac{\tau S}{C}$ has to do with cache misses. If $S > C$, the end user makes use of more software than fits on the local cache, and occasionally the machine will have to fetch a program from the network. Bandwidth requirements increase with S, the total amount of software used. Bandwidth requirements decrease with C, the size of the cache. Finally, bandwidth requirements are affected by human usage patterns τ which is the rate at which the user accesses software, times the miss rate.

Note that the first term penalizes a larger cache while the second term rewards a larger cache. Thus caching only makes sense when the benefits outweigh the costs. Caching is most successful with slowly changing frequently used software. Caching is least successful with frequently changing, rarely used software.

Computational resources on a single machine were not a major factor in the design in this system. From an end user's point of view, computational resources are negligible. The computational cost of running the Castanet tuner is constant. The computational cost of cache management is constant for many replacement algorithms, such as random replacement.

4.1.3 Resource demands on the system

The concept of middlemen allows tree-like distribution patterns. End users may subscribe to middlemen. Middlemen may subscribe to either sources or other middlemen. If a middleman can support b subscribers, and there are k layers of middlemen, b^{k+1} end users may be supported. This scalability is already available through Castanet, by making use of repeaters.

The system gains additional efficiency because a separate distribution tree for each piece of software is not necessary. Since software packages can be merged, separated, or modified by any middleman at any level, distribution trees are likely to become highly intermingled. For example, consider the software programs Apache and Analog. Apache is a web server, while Analog analyzes web server logs. These programs are complementary and might be used together by n end users.

One possibility is to set up two parallel distribution trees with each with b^k middlemen. Another possibility would for a single middleman, at the top of the distribution tree, to merge the two programs into a single channel. Then, only one distribution tree size b^k would be required. If the programs had only partial overlap, e.g. only some endusers would want both programs together, the combination might occur further down the distribution tree.

Since this system supports arbitrary distribution networks, efficient distribution structures can be achieved. While finding an optimum structure for a large number of channels might be extremely difficult, market forces should promote efficiency.[1]

4.1.4 When demand exceeds capacity

Resource allocation for a given machine is performed using the following assumptions.

1. Demand for network bandwidth may exceed capacity.
2. Demand for local storage may exceed capacity.

[1]This thesis was nearly titled "An Economy of Software Maintenance"

Performance degrades gracefully as demand exceeds resources. We've already seen this happen when demand for local storage exceeds supply. For example, we'd like to store copies of software in local storage. Assume that a user has capacity for six programs in local storage. If the user runs a seventh program, one of the six earlier programs that was stored on disk will be erased. The penalty for insufficient local storage is poorer performance by the system as programs are swapped in and out of the cache.

Overdemand for bandwidth resources are handled in the same fashion. For example, a software channel that requires ten megabytes of updates every five minutes may exceed total available bandwidth. Or, perhaps the aggregate bandwidth requirements of all the programs stored locally will exceed available bandwidth. When bandwidth demands outstrip supply, some updates must be dropped.

Castanet already handles the first case, where a channel attempts to update itself faster than bandwidth can handle. If a channel is being updated, it will cannot request another update until the current one is completed. This effectively reduces update rates to a manageable level.

There are several strategies for deciding priority for multiple competing channels. This design uses least recently used. Thus a channel not used in a long time will be the most likely to be out of date. Using LRU for updates complements the use of LRU for local storage.

Since Castanet allows the user is allowed to directly adjust cache management (both for bandwidth and local storage) greater efficiency may be achieved at the cost of ease of use.

4.2 Comparison of technology bases

This thesis is about software distribution and maintenance. It is also about managing collections, automating updates, and taking advantages of economies of scale. To take advantage of economies of scale, a globally scalable infrastructure is required. Several candidates exist, and Castanet was chosen. Other candidates would have also worked.

4.2.1 Castanet

Castanet is a replication system developed as a commercial product by Marimba Corporation. It allows files to be automatically copied from filesystem to filesystem, at programmable intervals. Castanet supports incremental updating to allow for more efficient use of bandwidth and allows repeaters[2] to provide global scalability.

Because Castanet uses HTTP as it's transport protocol, the system will work through most existing firewalls.

4.2.2 Coda

Coda [34] is a distributed file system derived from AFS developed at Carnegie Mellon University. It supports a large-scale distributed computing environment composed of Unix workstations. Disconnected operation is being developed to support fully mobile computing.

Much of the caching and disconnected operation strategies used in this thesis were modeled after Coda. Coda includes automated and user assisted cache operations and reintegration upon connection; I had to add this functionality to Castanet.

Coda is a filesystem operating at the Unix vnode layer and must be integrated into the operating system. Simply installing Coda requires root privileges. This make Coda a little more unwieldy to work with than application layer software like Castanet for casual development.

Coda is more general than Castanet, allowing the full read-write operation of a distributed file system. It also takes a different approach to scalability. Castanet provides replication between different local file systems. Coda, on the other hand, allows multiple, replicated file servers to provide access to a single file.

Unfortunately, performance degrades with increasing numbers of replicated file servers; a single replicated server causes a performance drop of 5% [34] and additional replications further degrade performance. Thus it would not be possible, for example,

[2] A repeater is a program that receives a channel and rebroadcasts it.

for everyone in the world to access Netscape Navigator from a single location[3] in the Coda file system. It is not surprising to see performance degrade with the use of replicated file servers. While replication offers advantages in parallelism, it also requires overhead for synchronization. Since Coda is a read/write system, the file servers must be kept closely synchronized. Any changes to a file must be immediately communicated amongst the file servers. A read only system might not incur these overhead costs.

Thus it would make sense to copy a given program to several locations in the Coda name space, as opposed to keeping each piece of software at only one canonical location. It would certainly be possible to use Coda instead of Castanet as the underlying namespace for this thesis project.

Finally, in a practical sense, Castanet has an advantage from using HTTP as its transport layer. For global operation, it will be important for this system to operate through firewalls. Virtually all firewalls already have a hole in them for HTTP; the use of Coda would require an additional modification to firewalls.

4.2.3 WebFS

WebFS [18] is a file system being developed at the University of California, Berkeley. It is a global, cache coherent file system which allows unmodified applications to read and write to the URL name space. WebFS is built at the Unix vnode layer and supports HTTP as a transfer protocol. A prototype implementation exists for Sun workstations. WebFS would make a promising basis for this thesis project if it were a little more mature.

WebFS is designed to allow simple filesystem manipulation across the URL namespace. For example, consider this scenario envisioned by the WebFS developers. [37]

```
cd /http/www6conf.slac.stanford.edu
cat index.html
cd img
```

[3] A single name in the Coda namespace.

```
cd logo
xv 200x200.jpg
```

WebFS works with two protocols when manipulating files. If a machine is running the WebFS, a WebFS protocol serves files with performance rivaling other distributed filesystems. If WebFS software is not present, but a (nearly ubiquitous) HTTP server is, WebFS can serve files through that protocol. Thus WebFS has a strong advantage in the ability to work with systems supporting HTTP, without requiring WebFS specific software.

While WebFS does not provide replication tools like Castanet, the filesystem attempts to address scalability difficulties by integrating multicast into the file system. Multicast is used to make caching more efficient; caches need only flush their contents when receiving an invalidate signal over multicast. This eliminates the need for polling, increasing the number of clients a WebFS file server can handle. Still, the load on a WebFS server grows with the number of clients since multicast is not used for transmitting all data.

WebFS does nopt specify replicated servers like Coda. Thus it would not be possible for everyone in the world to access a program from a single location in the WebFS namespace. Like Coda, it would make sense to copy file to several locations.

If a future version of WebFS were able to service global demand from a single location in namespace (through judicious use of the multicast protocol), an automated copying tool would still be required. Middlemen require automated, periodic copying to provide organizational services. (It will be necessary for middlemen to copy canonical copies of software into their own collection and namespace if they wish to provide modifications, checkpoint services, etc.)

Castanet provides a slightly higher level of maturity and very efficient copying tools. Castanet also supports disconnected operation while WebFS does not explicitly do so. WebFS provides a high degree of ubiquity. They both make excellent candidates for technology bases.

4.2.4 WebNFS

WebNFS [20] is an initiative by Sun Microsystems to use NFS as a transport layer for the world wide web, as opposed to HTTP. The NFS transport layer has been tuned over many years and is significantly faster than HTTP. Other speed advantages of WebNFS come from low connection overhead, and tight integration with the operating system.

While WebNFS allows faster access to a distributed namespace than HTTP, it does not provide any additional functionality. In theory anything that is currently implemented on top of HTTP would also be implemented on top of WebNFS.

The higher performance of WebNFS, including a throughput of over 6MB/s for some implementations, is not so dramatic as to allow global access for a single file. WebNFS does not provide automated replication tools, multicast, or other features that might provide global scalability.

Thus, Castanet, while built on top of the slower HTTP protocol, provides the tools needed to achieve scalability. It is important to note that HTTP was chosen as a transfer protocol by Castanet (and therefore this project) not because of speed, but rather ubiquity. HTTP is supported by an enormous amount of software, and nearly all firewalls are configured to allow HTTP packets to slip through. WebNFS is not currently in wide use, which means it is not generally supported in the software infrastructures. This makes WebNFS limiting and thus poor choice for this design.

	access control	globally scalable	efficient copying tools	disconnected operation	cache management
Requirements	x	x	x	x	x
Coda	x			x	x
WebFS	x	x			x
WebNFS	x	N/A			
Castanet		x	x	x	

Table 4.1: Feature space of existing systems

4.3 Other maintenance systems

4.3.1 Oil Change and Tuneup.com

Oil Change [12] and Tuneup.com [16] are some of the first companies to provide value added services in terms of software maintenance. These are commercial companies that keep track of thousands of software programs, and allow automation of updates to locally stored software.

Oil Change and Tuneup.com only offer maintenance services; they do not provide organization or distribution services. For example, an end user will be required to locate and install all of their own software, piece by piece. By signing up with these organizations, the user allows the service to update individual programs. Thus the number of maintenance links are still $M \times N$ for M machines installing N pieces of software. The cost of the maintenance links are reduced, however.

Neither of these services allow the user to maintain a larger amount of software than local storage will hold; there is no automated cache management. A user who only has local storage capacity for six pieces of software will need to install and remove software manually if they wish to make regular use of seven pieces of software.

Neither organization allows additional levels of middlemen. Tuneup.com and Oil Change don't distribute software. Thus, there is no way to create specialty distributions based on their general collection.

Chapter 5

Applications

5.1 Value added services

5.1.1 General software collections

The software distribution system as presented provides distributed maintenance. For instance, the author of each program can maintain and update every copy, everywhere. Further organization is possible. An organization such as MIT might gather together a collection of software from various sources and provide access to it in a convenient way. Instead of subscribing to Framemaker, Matlab, Maple, gnuplot, transgif, emacs, scheme, gcc, Netscape Navigator, lynx, EZ, gif2tiff, traceroute, xcalc, and five hundred other channels, I might just subscribe to MIT Project Athena.

5.1.2 Safety checking

Checkpoints can reduce vulnerability to computer viruses, and offer a measure of quality control. Instead of subscribing directly from the author of individual programs, someone might subscribe to a special "Virus Safe" distribution point, which rigorously checks against unsafe programs. Checkpoints might subscribe to a general software collection and only pass on software that meets security standards. The introduction of checkpoints allows a host of value added security services.

5.1.3 Specialty distributions

Another application would be specialty distributions of software. For instance a government office such as the Department of Motor Vehicles might have a standard collection of software, including a spreadsheet, word processor, and custom driver database software. This set of utilities could be gathered into a single package and replicated across the state. Another example might be computer manufacturers. Computer companies such as Gateway 2000, Dell, or Micron often bundle a set of software with their machines. This package could be made available as a replication unit. The US Military might be interested in their own special, "military approved" software distributions.

5.1.4 Frozen software

Any form of maintenance a customer might wish for can take the form of a value added service. For instance, many people absolutely fear change, and do not for any reason want software to change out from under them. Completely frozen software sets (where no updates occur) can be made available. The end user would not receive the benefit of automatic updates, but would still be able to take advantage of economies of scale. The end user will also have access to more software than can be stored on the local disk.

5.1.5 Retail stores

A commercial retailer is already in the business of organizing and selling software. A retailer taking advantage of this distribution system will blur the line between a commodity and service based industry. A store which presently sells shrink wrap or downloadable software of many varieties could sell the same packages, and offer maintenance services for them.

5.1.6 Modifications or customizations

Another possible service is modification of code. The value added might be a software patch, a more capable library, or some other customization to a general software package. Imagine a service which receives an English language version of a newspaper, makes modifications, and then transmits a French language version of the paper.

5.1.7 Unanticipated services

A large, multi-tier distribution system with arbitrary numbers of middleman will allow arbitrary organization of programs and maximum efficiency. Many of the services which can be provided cannot be anticipated; the question is almost akin to predicting applications for the World Wide Web during its inception.

5.2 Future work

5.2.1 Production quality implementation

The implementation presented in this thesis serves as a proof-of-concept. A more robust system would be required to gain wide acceptance in the market. Other factors, such as open design and participation by standards bodies would also be important.

5.2.2 Read/Write

This thesis has focused on read-only. Returning to the river metaphor, while lots of tributaries and waterways have been explored, there has little consideration on the two way flow of water. Important questions of coherency and fragility of data arise in a bi-directional distribution system.

5.2.3 Economic model

A value added middleman should be able to receive compensation commensurate with his services. Does the middleman have to resell software or can distribution and maintenance be provided without the burden of resale? A solid economic model for value added services is necessary for deploying this distribution system on a large commercial scale.

Appendix A

Athena: an example of a large software collection

MIT Project Athena maintains a wide range of software. An organized description of major packages is summarized in online documentation. [1] To provide a feeling for the sheer number of programs being maintained, this appendix lists a subset of the programs available. The following programs are available from the outland, graphics, gnu, and sipb lockers for the linux platform. This listing covers only a fraction of all the software on Project Athena.

A.1 outland

Mosaic-2.7b5	linecheck	tupload
Mosaic-BETA	lynx	txconn
Mosaic.real	lynx-2.4	unclutter
Mosaicclient	nc	uncompface
Mozilla301-gold	nedit	units
Mozilla40	nethack-qt	uudeview
acroread3	netscape-4.0	uuenview
agrep	netscape-beta	vosaicbg
arena	netscape-gold	vrweb

```
arena-1.0b2
arena-BETA
bongo
bongo-player
c1541
castanet-admin
castanet-publish
castanet-transmitter
compface
cryptclean
dvi3812
dvilj
dvilj2
dvilj2p
dvilj4
dviljp
ec
ee
fs2xbm
fvwm95-2
fvwm95-2.old
giftrans
glimpse
glimpseindex
glimpseserver
globe
ikon2xbm
infocom
jmacs
joe
jpico
jstar
octave
octave.bin
ogg
paranoia
pdftops
petcat
pico
pmake
prtgif
pstoedit
qix
qservers
qstat
raplayer
raplayer-3.0
rclock
readcomics
rjoe
rocks
rolodex
slocat
smtpmail
tdownload
term
termidx
texi2html
texi2html-menu
tgif
tgif-2
tgif-2.16pl4
tgifwww
tkfibs
vrweb-1.3
vrweb.real
vrwebnet
vt
wine
wine.real
wine.sym
winestat
x64
xarchie
xbm2ikon
xbmcut48
xbmsize48
xcolor
xdiff
xdla
xearth
xfishtank
xpaint
xpdf
xpmroot
xpr
xquake
xscretched
xshower
xsnow
xsurface
xteddy
xtoolwait
xvclient
xxgdb
xzewd
```

keyboard
keymap
knews
ksh
latex2html
less
lesskey
lha
tmon
trdate
trdated
tredir
trsh
tshutdown
tudpredir
tuner
xzewd-zephyr-baby
xzewd-zephyr-baby.old
xzewd.old
xzul
xzul.beta
z5
zcrypt

A.2 graphics

anytopnm
asciitopgm
atktopbm
aub
bggen
bioradtopgm
bmptoppm
brushtopbm
cmuwmtopbm
crystile
fitstopnm
fractile
fs2xbm
fstopgm
g3topbm
gemtopbm
gifmerge
giftool
giftopnm
giftoppm
giftoppm~
mpeg_encode
mpeg_play
mpeg_vga
mtv
mtv.real
mtvtoppm
munpack
pbmclean
pbmlife
pbmmake
pbmmask
pbmpscale
pbmreduce
pbmtext
pbmto10x
pbmto4425
pbmtoascii
pbmtoatk
pbmtobbnbg
pbmtocmuwm
pbmtoepsi
pgmenhance
pgmhist
pgmkernel
pgmnoise
pgmnorm
pgmoil
pgmramp
pgmtexture
pgmtofs
pgmtolispm
pgmtopbm
pgmtoppm
pi1toppm
pi3topbm
picttoppm
pixmap
pjtoppm
pktopbm
pnmalias
pnmarith
pnmcat
pnmtops
pnmtorast
pnmtosgi
pnmtosir
pnmtotiff
pnmtoxwd
ppm3d
ppmbrighten
ppmchange
ppmdim
ppmdist
ppmdither
ppmflash
ppmforge
ppmhist
ppmmake
ppmmix
ppmnorm
ppmntsc
ppmpat
ppmquant
ppmtoyuv
ppmtoyuvsplit
psidtopgm
pstopnm
pstopnm~
qrttoppm
rasttopnm
rawtopgm
rawtoppm
rgb3toppm
sgitopnm
sirtopnm
sldtoppm
spctoppm
spottopgm
sputoppm
tgatoppm
tifftopnm
transgif
vdcomp
whirlgif

gimp	pbmtoepson	pnmcomp	ppmquantall	xanim
gouldtoppm	pbmtog3	pnmconvol	ppmqvga	xanim-BETA
hipstopgm	pbmtogem	pnmcrop	ppmrelief	xbm2ikon
hpcdtoppm	pbmtogo	pnmcut	ppmshift	xbmcut48
icontact	pbmtoicon	pnmdepth	ppmspread	xbmsize48
icontopbm	pbmtolj	pnmenlarge	ppmtoacad	xbmtopbm
ikon2xbm	pbmtoln03	pnmfile	ppmtobmp	xcmap
ilbmtoppm	pbmtolps	pnmflip	ppmtogif	ximtoppm
imanimate	pbmtomacp	pnmgamma	ppmtoicr	xli
imcombine	pbmtomgr	pnmhistmap	ppmtoilbm	xloadimage
imconvert	pbmtopgm	pnmindex	ppmtomap	xpaint
imdisplay	pbmtopi3	pnminvert	ppmtomitsu	xpaint-BETA
imgtoppm	pbmtopk	pnmmargin	ppmtopcx	xpmtoppm
imidentify	pbmtoplot	pnmnlfilt	ppmtopgm	xroot
imimport	pbmtoptx	pnmnoraw	ppmtopi1	xv
immogrify	pbmtox10bm	pnmpad	ppmtopict	xvminitoppm
immontage	pbmtoxbm	pnmpaste	ppmtopj	xwdtopnm
imsegment	pbmtoybm	pnmrotate	ppmtopjxl	ybmtopbm
include	pbmtozinc	pnmscale	ppmtopuzz	yuvsplittoppm
lib	pbmupc	pnmshear	ppmtorgb3	yuvtoppm
lispmtopgm	pcxtoppm	pnmsmooth	ppmtosixel	zeisstopnm
macptopbm	pgmbentley	pnmtile	ppmtotga	
mgrtopbm	pgmcrater	pnmtoddif	ppmtouil	
mpack	pgmedge	pnmtofits	ppmtoxpm	

A.3 gnu

autoconf	gcsplit	gmerge	gsdj	gwhoami
autoheader	gcut	gmkdir	gsdj500	gxargs
autoreconf	gdate	gmkfifo	gsed	gyes
autoscan	gdb	gmknod	gshar	gzcat
autoupdate	gdc	gmt	gsize	gzcmp

bash	gdd	gmv	gsleep	gzdiff
bashbug	gdf	gnice	gslj	gzexe
bdftops	gdiff	gnl	gslp	gzforce
bison	gdiff3	gnm	gsnd	gzgrep
c++	gdir	gnohup	gsort	gzip
c++filt	gdircolors	gnuan	gsplit	gzmore
cvs	gdirname	gnuchess	gstrings	gznew
cvsbug	gdu	gnuchessc	gstrip	ifnames
egrep	gecho	gnuchessn	gstty	info
expect	ged	gnuchessr	gsum	ld.bad
fgrep	genclass	gnuchessx	gsync	makeinfo
flex	genv	gnuplot	gtac	objcopy
flex++	gexpand	gnuplot_x11	gtail	objdump
font2c	gexpr	god	gtar	pdf2dsc
g++	gfalse	gpaste	gtee	pdf2ps
g[	gfind	gpatch	gtest	postprint
gar	gfmt	gpathchk	gtic	printafm
gas	gfold	gperf	gtime	protoize
gasp	ggprof	gpr	gtoe	ps2ascii
gawk	ggrep	gprintenv	gtouch	ps2epsi
gbasename	ggroups	gprintf	gtput	ps2pdf
gbc	ghead	gpwd	gtr	rcs2log
gcaptoinfo	ghostname	granlib	gtrue	runtest
gcat	ghostview	grcs	gtset	screen
gcc	gid	grcsclean	gtty	screen-3.6.2
gchgrp	gident	grcsdiff	guname	tclsh7.5
gchmod	gindent	grcsmerge	gunexpand	texi2dvi
gchown	ginfocmp	grecode	guniq	texindex
gci	ginstall	gred	gunshar	unprotoize
gcksum	gjoin	greset	gunzip	updatedb
gclear	gln	grlog	gusers	wdiff
gcmp	glocate	grm	guudecode	wftopfa

gco	glogname	grmdir	guuencode
gcomm	gls	gs	gvdir
gcp	gm4	gsbj	gwc
gcpio	gmake	gsdiff	gwho

A.4 sipb

Mosaic	fortune	newsgroups	rn	where
Mosaic-2.6	fptocksum	nex	rrn	wm2xmcd
Pnews	fptomd5	nvi	sc	xarchie
Rnmail	fptosnefru	oneko	scqref	xcal
archie	freeze	pfrom	sdate	xcol
bfinger	ftp.expect	pic2tpic	skill	xdvi
bwrite	fwhois	pmail	snefru	xfig
cda	getactive	pop	snice	xfig.old
char	iap	prtgif	tex	xftp
cksum	idraw	psc	tex.new	xmcd
cmmf.new	inews	psify	tgif	xrn
dclock	latex	qedx	tgif-3.0pl7	xrn-new
descrypt	latex.new	rcbook.n	tgrind	xscreensaver
dsc_notify	latex2e	rcbook.t	th	xshower
dvips	lndir	rcextract	thesaurus	xstapler
enscript	mcvert	rcindex	tin	xvile
exmh	md5	rcintro	transfig	xwebster
exmh-async	melt	rckeep	trn	xzwrite
exmh-bg	mf.new	rckeepnew	type	ytalk
fcat	nawm	rcnew.n	unfreeze	zcat
fig2dev	ncftp	rcnew.t	unzip	zpunt
fig2ps2tex	ncftp-new	rcnroff	vile	zunpunt
fig2ps2tex.sh	nenscript	rcshow	webster	
fingerprint	newsetup	rctypeset	whats	

Appendix B

Protocols and Design Standards

The MIT Athena directory model was adapted for organization of executables. Two directories have special names: replication and arch. Replication contains everything that a channel might distribute. Arch contains platform specific files, following Athena's Unix manual page lockers(7) [29], paraphrased below.

> In order to avoid any sort of clutter in the top level directory of a directory, all machine dependent directories are placed under a directory called arch. Under arch is one directory, for each supported platform. These directories are named by concatenating Castanet's OSname, OSversion, and OSarch values (Windows 95 4.0 x86, Solaris 2.x sparc, etc.) Under each of these directories are directories containing a specific type of machine dependent data, such as binaries or libraries (bin, lib, etc.).

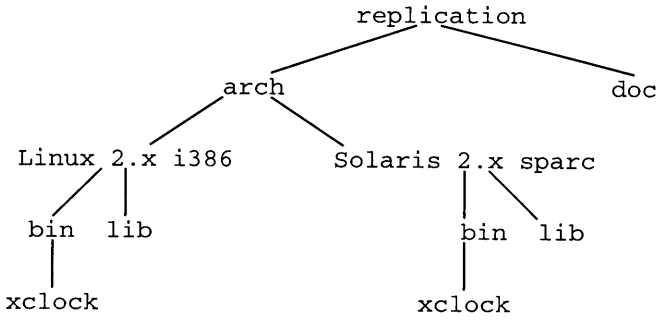

Figure B-1: A simplified diagram of the standard multiplatform directory structure.

Bibliography

[1] Summary of available athena software. Athena On-Line Help (OLH), Cambridge, Massachusetts. <http://web.mit.edu/olh/Software/index.html>.

[2] Castanet white paper. Marimba corporation web site, December 1996. <http://www.marimba.com/developer/castanet-whitepaper.html>.

[3] Sendmail daemon mode vulnerability. Advisory CA-96.24, CERT, CERT Coordination Center, Pittsburg, Pennsylvania, November 1996. <ftp://info.cert.org/pub/cert_advisories/CA-96.24.sendmail_vul>.

[4] Sendmail group permissions vulnerability. Advisory CA-96.25, CERT, CERT Coordination Center, Pittsburg, Pennsylvania, December 1996. <ftp://info.cert.org/pub/cert_advisories/CA-96.25.sendmail_vul>.

[5] Sendmail v.5 vulnerability. Advisory CA-95.08, CERT, CERT Coordination Center, Pittsburg, Pennsylvania, August 1996. <ftp://info.cert.org/pub/cert_advisories/CA-95.08.sendmail.5.vulnerability>.

[6] Sendmail vulnerabilities. Advisory CA-96.20, CERT, CERT Coordination Center, Pittsburg, Pennsylvania, November 1996. <ftp://info.cert.org/pub/cert_advisories/CA-96.20.sendmail_vul>.

[7] Vulnerability in ncsa/apache cgi example code. Advisory CA-96.06, CERT, CERT Coordination Center, Pittsburg, Pennsylvania, August 1996. <ftp://info.cert.org/pub/cert_advisories/CA-96.06.cgi_example_code>.

[8] Alpha fact sheet. Technical Report EC-QP97C-TE, Digital Equipment Corporation, 1997. <http://www.digital.com/semiconductor/21164-fact.html>.

[9] Microsoft word 97 document converter. Technical report, 1997. <http://www.microsoft.com/word/freestuff/converters/wrd97cnv.htm>.

[10] Mime conversion buffer overflow in sendmail versions 8.8.3 and 8.84. Advisory CA-97.05, CERT, CERT Coordination Center, Pittsburg, Pennsylvania, February 1997. <ftp://info.cert.org/pub/cert_advisories/CA-97.05.sendmail>.

[11] Netscape release notes. Netscape Web Site, 1997. <http://home.netscape.com/eng/mozilla/3.0/relnotes/>.

[12] Oil change. Web site, 1997. <http://www.cybermedia.com/products/oilchange/ochome.html>.

[13] PC 98 Design Guide. Review Draft Rev. 0.6, Intel Corporation and Microsoft Corporation, April 1997.

[14] Price Watch, 1997. <http://www.pricewatch.com/>.

[15] The Netcraft Web Server Survey, April 1997. <http://www.netcraft.co.uk/Survey/>.

[16] Tuneup.com. Web site, 1997. <http://www.tuneup.com/>.

[17] S. Armstrong, A. Freier, and K. Marzullo. Multicast Transport Protocol. IETF Request for comments 1301, February 1992. ftp://ftp.cs.tu-berlin.de/pub/local/kbs/mtp/related_work/RFC/rfc1301.

[18] Eshwar Belani, Alex Thornton, and Min Zhow. Security and Authentication and in WebFS. Technical report, University of California, Berkeley, December 1996. <http://now.cs.berkeley.edu/WebOS/>.

[19] Thomas Boutell and Tom Lane et all. PNG (Portable Network Graphics) Specification. W3C Recommendation 1.0, World Wide Web Consortium, <http://www.w3.org/pub/WWW/TR/REC-png-multi.html>, October 1996.

[20] Brent Callagahan. WebNFS. Technical report, Sun Microsystems, April 1997. <http://www.sun.com/sunsoft/solaris/networking/webnfs/webnfs.ps>.

[21] Wayne W. Chou, Joseph M. Kulinets, Laszlo Elteto, and Frederik Engel. Method of software distribution protection. US Patent 5337357, Aug 1994. <http://patent.womplex.ibm.com/details?patent_number=5337357>.

[22] Netscape Communications Corporation. Netscape Navigator 3.0 reviewer's guide, 1996. <http://www.plexon.com/nn.html>.

[23] Dan Farmer. Technical report, 1996. <http://www.datapro.com/>.

[24] W. Wayt Gibbs. Profile: Dan Farmer: From Satan to Zen. *Scientific American*, 276(4):32–34, April 1997.

[25] George Gilder. Fiber keeps its promise. *ASAP: Forbes Supplement on the Information Age*, April 1997. <http://www.forbes.com/asap/97/0407/090.html>.

[26] Van Jacobson. How to kill the internet. In *SIGCOMM '95 Middleware Workshop*, Berkeley, California, August 1995. Lawrence Berkeley Laboratory.

[27] Fred Langa. Dribbleware, take 2. *Windows*, pages 19–20, February 1997. <http://www.winmag.com>.

[28] *Microsoft Announces Zero Administration Initiative for Windows*, October 1996. <http://www.microsoft.com/corpinfo/press/1996/Oct96/ZAWinpr.htm>.

[29] MIT Athena. *lockers(7)*, December 1994. Unix manual page.

[30] Lily B. Mummert, Ebling Maria R, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[31] Henrik Frystyk Nielson and Jim Gettys. HTTP — Hypertext Transfer Protocol. W3C Technical Area, World Wide Web Consortium, <http://www.w3.org/pub/WWW/Protocols/>, 1996.

[32] Salvatore Salamone. Reducing the cost of PC connectivity. Whitepaper, WRQ, 1996. <http://www.wrq.com/whitepap/costsupp/resup.htm>.

[33] M. Satyanarayanan. Fundamental challenges in mobile computing. *Fifteenth ACM Symposium on Principles of Distributed Computing*, May 1996. <http://www.cs.cmu.edu/afs/cs/project/coda/Web/docs-coda.html>.

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

[35] Jeffrey Schiller. Personal Interview, 1995.

[36] MIT Information Systems. Athena 8.0 release notes, August 1996. <http://web.mit.edu/olh/Release/8.0/>.

[37] Amin M. Vahdat, Paul C. Eastham, and Thomas E. Anderson. WebFS: A global cache coherent file system, 1997. <http://www.cs.berkeley.edu/ vahdat/webfs/webfs.html>.

[38] Kathy Walrath. Writing compatible programs. JDK 1.1 Documentation, Sun Microsystems, 1997. <http://java.sun.com/products/jdk/1.1/compatible/index.html>.

[39] Gidean A. Yuval and Michael Ernst. Method and system for controlling unauthorized access to information distributed to users. US Patent 86186, July 1994. <http://patent.womplex.ibm.com/details?patent_number=5586186>.