A Mobility-Aware File System - file availability in a mobile-aware, context aware environment

by

Owen O'Flaherty, B.A. Mod.

Dissertation

Presented to the

University of Dublin, Trinity College

in partial fulfilment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2005

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Owen O'Flaherty

9 September 2005

Permission to Lend and/or Copy

Dedicated to the Trinity cat, who appears to have left to find a new campus since a thesis was last dedicated to him.

And, of course, to Stirling.

Acknowledgments

Many thanks are due to my supervisor, Dr. Siobhán Clarke, who provided goals, advice and good humour during this summer. Invaluable advice from Andronikos Nedos and Kulpreet Singh is hereby acknowledged with sincere thanks. I am also grateful to Pete Barron for his time, advice and equipment; to Dr. Stefan Weber for his generous provision of laptops, and to Mark Gleeson for handy tips.

I would like to thank very sincerely my family who have put up with endless amounts during the lifetime of this dissertation, and Aileen, who hasn't seen me for weeks. And to the rest of the NDS class of '05—it's been a good 'un.

OWEN O'FLAHERTY

University of Dublin, Trinity College
September 2005

A Mobility-Aware File System - file availability in a mobile-aware, context aware environment

Owen O'Flaherty
University of Dublin, Trinity College, 2005

Supervisor: Siobhán Clarke

The rapid growth in number of portable electronic devices with wireless networking capabilities in recent times has thrown up many new computing problems and opportunities. Such devices typically lack the processing power common to more familiar desktop and laptop computers. They are also constrained by their limited power supply. Given the scenario of an urban environment in which a large number of people using such devices constantly move about with a need to share information, it is clear that existing network topologies are inherently unsuited to supporting the sort of applications that they might need.

This dissertation project addresses the changing role of that most integral part of computer systems in this new environment - the file system. A distributed file system called Stirling is presented that takes advantage of both the ad hoc nature of the network environment and the structure provided by the WAND (Wireless Ad-hoc Network for Dublin.)

This filesystem offers a set of features suited for ad-hoc networking. These features, such as virtual file placement and latency minimisation, may be set and adjusted by users at a fine-grained level. It is envisaged that such a system may be put in place on a backbone network such as the WAND in order that users and their devices can be guaranteed 'reachability' to other devices from one end of the network to the other. Usability scenarios are presented and analysed, and test cases documented.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 About this chapter

This chapter explains how this dissertation is structured, and outlines the objectives of the project.

1.2 About this document

- **Chapter 2** provides essential background information on wireless networking, filesystems and the attendant terminology that is crucial to understanding the motivation behind this project.
- **Chapter 3** describes various distributed filesystems in use today that share some of the goals of this dissertation.
- **Chapter 4** details the design of the Stirling filesystem, from motivation to use cases.
- **Chapter 5** presents details on the implementation issues encountered while creating the filesystem, and explains how it was evaluated.
- **Chapter 6** concludes the dissertation.

1.3 Research objectives

The work in this project was carried out with the following goals:

- To learn more about modern, cutting-edge technologies that are shaping the way we use technology today.
- To acquire new technical skills, notably the use of the Linux operating system and proficiency in Python programming.
- To undertake a research review of a particular area, and to evaluate implementation goals based on that review.
- To present the results of the above in an informative and readable academic work.

Note: I make reference throughout this document to the ‘filesystem’. After a significant period of time investigating the various ways in which files may be structured, the two separate words ‘file’ and ‘system’ tend to become conflated in the researcher’s head and it becomes natural to think of the concept as a unit. This is the convention that is adopted throughout this document.

Chapter 2

Background

2.1 About this chapter

Motivating factors behind this project are the subject of this chapter—the reasons for developing a file system that allows users to take advantage of a mobile, ad hoc networking environment. To do this, brief synopses of the areas involved are presented, such as a quick introduction to the world of file systems and the difficulties and opportunities afforded to it by wireless networking. These will show how there is plenty of scope for a project of this kind, and will outline some of the proposed benefits of such a system.

2.2 The importance of being wireless

Using a laptop computer while wandering through the streets of Dublin's city centre is not generally considered advisable, at least if keeping the laptop intact is important to its owner. But given the proliferation of mobile phones and PDAs coupled with the advancing processing power and storage capacity of relatively small devices, this scenario does not sound quite so implausible.

Wireless technology has developed at such a pace in recent times that a sizeable gulf has emerged between the expectations of users and the growing potential of what is actually possible in this enviromment. This area is currently the focus of significant market research in what the future holds for wi-fi technology and other wireless technologies. Consultancy think-tank Ovum has forecast growth from 6 million users of some form of wireless e-commerce at the report's time of writing, for example, to 500 million users in 2005. European Information Technology Observatory (EITO) figures suggest 10 million present users to 175 million in 2005. [1]

The following sections explain some of the technologies mention here and how research is conducted exploring ways to use these technologies.

2.3 Mobile ad hoc networking

The year 2004 saw the sudden widescale adoption of wireless networking. As broadband services took over from dial-up connections in the mainstream of the internet access market, sales of wireless access points grew in a similar fashion, as households with multiple internet-ready devices sought to take advantage of an always-on connection with minimum fuss. In this way, wireless networking has become the norm rather than the exception, as 'hotspots' proliferate in urban areas.

This form of network access relies on a wireless access point which provides connectivity to the other nodes on the network, and acts as a gateway to other networks. This is commonly referred to as a 'star' topology, with the accesss point at the centre and the participating nodes at the star's apexes. When nodes communicate, they do so indirectly via the access point, which acts as a router for messages sent between the nodes. Such a configuration implies some sort of previously defined topology—typically, the access point itself is connected via a physical wire link of some sort to another network (in most broadband cases, the access point incorporates router functionality and connects to the ISP using DSL over the phone line.) However, it is quite possible for nodes possessing radio transceiver cards to talk directly to each other without the need for a centralized access point. This is what is known as 'ad hoc' mode. The early 802.11 protocol incorporated the ability for nodes using it to connect in this manner.

[1]Source: http://www.1step2europe.com/markets/web-mobile/.

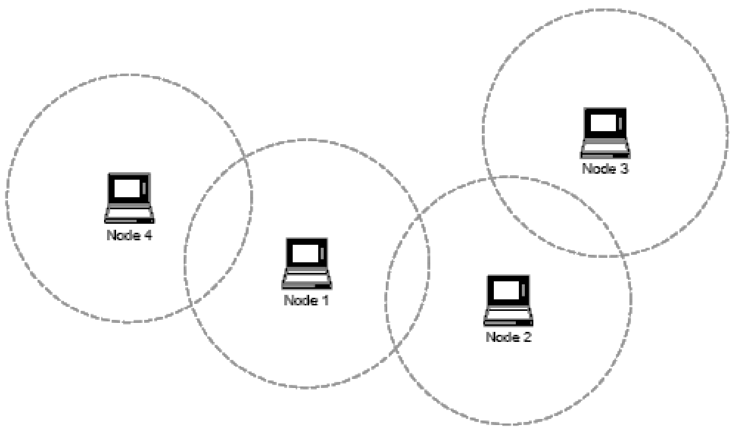

Figure 2.1: Wireless devices connected in ad hoc mode (Derived from [KB].)

A mobile, ad-hoc network, therefore, consists of a set of wireless-capable nodes with routing functionality and the location of which is not fixed (see Figure 2.1.) Hence, the network configuration is that of an arbitrary topology. The network must not be reliant on any supporting infrastructure. Nodes must be capable of routing messages, in a ‘store-and-forward’ sense, as a given node may be only able to communicate directly to a subset of the network at any given time, and so messages must ’hop’ their way between nodes until they reach a node which is within range of the destination.

It is worth noting that the use of the term ‘ad hoc’ nowadays implies routing capability, whereas the 802.11 standard from the IEEE simply specifies ‘ad hoc’ as a network lacking network infrastructure to organise participating nodes without any prerequisite routing functionality. This would imply that participating nodes must be relatively close together, so as to ensure each node is within transmitting and receiving distance of all others.

There are certain features that set ad-hoc wireless networks apart from traditional, even access point-based, networks. [CM99] gives a list of these characteristics, explained as follows:

- **Dynamic Topologies.** The most useful characteristic of wireless technology is the mobility it affords to its user. The network must be able to cope with shifts in node topology as a result of this, with its attendant effects of route alteration and differing transmission delays.

- **Energy-constrained operation.** Battery power is, remarkably, a major constraint in the development of efficient wireless computing devices, as it has lagged behind in terms of keeping up with developments in processing capability and the increased power demands that come with it. Several strategies for energy conservation in wireless networks have been mooted, including allowing devices to ‘sleep’ in a low-power state when not in use, but this has consequences for routing protocols which then cannot guarantee multihop routes for all network participants.

- **Bandwidth-constrained, variable capacity links.** Unlike wired networks, links between nodes in such an environment rely an unpredictable radio frequency medium, the characteristics of which are highly dynamic due to node mobility and external interference. For this reason, throughput capacity is highly unpredictable.

- **Limited security.** The wireless medium means that there is nothing to stop interested parties with radio equipment listening on all network traffic. Traditional network attacks become a degree easier due to this fact. Extra care must be taken when designing ad hoc protocols such that this consideration is taken into account.

A further characteristic of ad hoc networks is the lack of configuration required to administer membership and routing of the network. Nodes should be able to self-configure membership information in a given network on joining, much in the same way fixed networks allocate addresses via DHCP, for example.

2.4 About AODV

AODV, the Ad hoc On-demand Distance Vector routing protocol, is a solution to the porblem of limited radio range in wireless networks. While a transmission route can always be guaranteed in normal circumstances in a wired network (and a route that is not expected to change very often) this is not the case in the wireless domain. Using dynamic discovery, each participating node maintains a route table of other nodes it can communicate to.

This route table is guaranteed loop-free and is capable of changing as routes become stale and invalid due to interference or adjusted network topology. New routes may also arise should transmission errors occur. It is built up using a combination of node awareness of its immediate neighbours (nodes only one hop away) and ‘route request’ messages sent on from other nodes. Neigbours are discovered by periodically sending ‘HELLO’ messages on broadcast. Route Request messages (RREQ) are sent out when a packet has to be sent to a node for which no address exists in the routing table. These messages are passed on to other nodes as necessary, with a ‘time to live’ value to ensure the message will not circulate aimlessly forever around the network.

All AODV messages use sequence numbering to indicate how ‘fresh’ (and hence, valid) the information they carry is. This enables nodes to choose between conflicting route information. Route error messages (RERR) are sent out when a node discovers a bad route, and enables other nodes to pick up on this new information.

An example sequence of route discovery is shown in Figure 2.2.

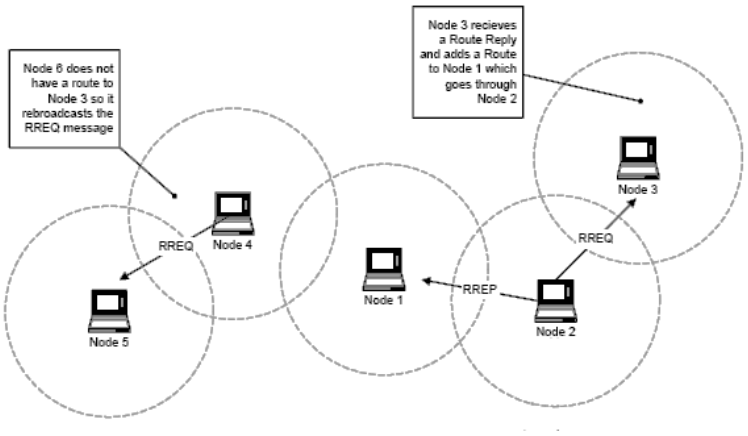
Node 6 does not have a route to Node 3 so it rebroadcasts the RREQ message
Node 3 recieves a Route Reply and adds a Route to Node 1 which goes through Node 2
Node 3
RREQ
Node 4
RREQ
Node 1
RREP
Node 2
Node 5

Figure 2.2: Route discovery in AODV. (Derived from [KB].)

Various implementations of AODV exist; those considered for use in this dissertation are those from the University of Uppsala (a user space daemon) and Kernel-AODV, from the National Institute of Science and Technology (NIST). The latter is the preferred version, as it operates as its name suggests as a kernel module, and hence benefits from performance gains. Packets queued for routing do not need to be ‘context-switched’ into a user space process to be inspected. This choice is described in detail later in this document.

2.5 About WAND

A relatively high-profile use of ad hoc wireless networking is the Wireless Ad hoc network for Dublin, or WAND for short. It is an initiative coordinated by the Distributed Systems Group in the Department of Computer Science at Trinity College Dublin in association with Media Lab Europe (MLE), the latter of which, sadly, is now defunct at the time of writing. It is an experiment in the application of metropolitan-scale wireless networking through the installation of a series of permanent wireless nodes distributed through Dublin's city centre.

These nodes consist of wireless-enabled PCs running a Linux installation with AODV. They are scattered along a route stretching from the Pearse Street end of Westland Row, up Lincoln Place, along Nassau and Dame Streets and finishing up around Lord Edward Street, located in various buildings along the way (see Figure 2.3.) This distribution provides a minimum of connectivity from one end of the two kilometre stretch to the other, and may be augmented by the presence of other dynamically participating users, such as people in the area armed with suitably-equipped PDAs. The units themselves are embedded PCs of dimensions roughly 6cm x 6cm x 12cm, containing the PC boards, power supply and WiFi equipment.

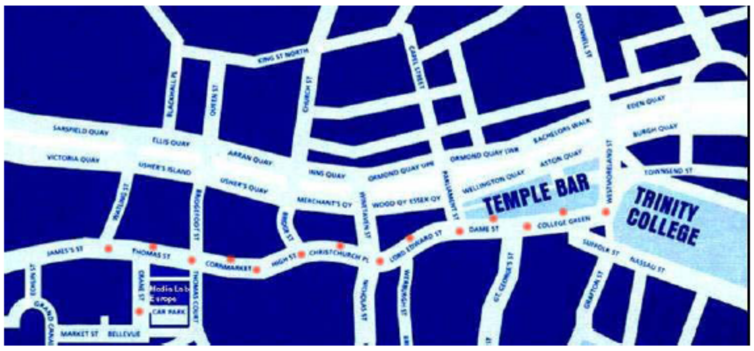

Figure 2.3: The placing of WAND nodes around Dublin.

This 'test bed' allows potentially interesting distributed wireless applications to be tested in the real world - most often demonstrating that what is most robust, stable and usable in lab conditions or in simulation quickly faces a whole new set of unanticipated problems when deployed in a useful environment.

2.6 Filesystems, old and new

From leading-edge technology to the more mundane and unexciting - this project considers the role of the file system in an environment such as that described above. It is useful at this point to consider the origins and many facets of this bedrock of computing that we all take for granted.

File systems have been at the heart of modern practical computing since its inception. The hierarchical ‘tree’ model that we are so used to has proved surprisingly durable. The first useful UNIX file systems were based on this notion of organising files using folders, or directories, which could be recursively nested and in which non-directory files themselves could be located at any level. This model has been used in a similar fashion by rival PC operating system vendors and others and, as such, has allowed for a certain level of compatibility between such systems. This approach has lasted so long because it has proved so suitable to monolithic, centralized computer systems that, of necessity, traditionally dominated the area. Subsequent advances in network technology has not yet forced a radical redesign of this paradigm that has enjoyed widescale deployment. This section describes the desirable characteristics of file systems, and how their design and interworking has responded to the network age.

2.6.1 About filesystems

It has been observed that the UNIX operating system works on the basis that ‘everything is a file’. That is, that the most important component part of a computer is its filesystem, and every other aspect of the environment—display, network, peripherals—interacts with the operating system via a file or set of files. The obvious case of this is in the `/dev` directory in *NIX systems, where many ‘special’ files reside whose names provide a key to understanding their purposes, such as ‘hda0’, referring to a hard disk.

It is natural, therefore, to examine the purpose of the filesystem itself, to understand its purpose and motivations behind the designs of various filesystem implementations, and see how advances in technology and the way computing devices are used today affect the traditional view of how files are accessed and used. We are used to using the static hard disks on our computers, with their comforting and familiar hierarchies of directories containing sets of files, and nowadays almost as used to accessing such hierarchies on remote computers through the use of facilitating technologies like NFS on Windows Workgroups and Samba. Few, however, outside academia have much experience using filesystems developed for use in far less monolithic, predictable environments, where location of resources and availability of those resources at a given time can often be volatile factors. Such systems include the Andrew filesystem, designed to minimise communications costs in a fairly stable distributed environment, and Coda, which takes into account additional factors such as disconnected operation, replication and mobility. As modern usage patterns change, however, thanks largely to the widespread deployment of wireless technologies such as Wi-Fi and Bluetooth, such filesystems will feature more prominently and play a wider role in providing users with the resources they require. This is the motivation for this dissertation—given a certain non-traditional computing environment, how best might its file-related requirements be addressed?

To do this, we must go back to examining the fundamental static filesystem that we all take for granted and determine its purpose and implementation, and reevaluate this in light of modern usage patterns. It is hoped that this will lead to a better understanding of the requirements posed by large-scale computing environments and provide a better basis for possible design solutions.

2.6.2 What is it?

Large numbers of files require organisation. It is necessary for users to access resources in a uniform and convenient manner, while hiding from them the intricacies of storage allocation and other housekeeping details. It is the filesystem's job to do this organisation through efficient storage, directory maintenance, security management, file naming and sharing access. It must also provide an abstraction layer on top of the files to allow a common scheme of access. To facilitate all this, files themselves are considered to be more than just the raw data that they contain. A typical file consists of its contained data and an amount of metadata the latter consisting of naming information and attributes governing its type, access permissions, owner, and so forth. Directories themselves are special kinds of files in themselves consisting simply of a version of this metadata that groups ordinary files together.

A file is a sequence of data that is governed by attributes. This is not much use by itself, as an operating system would find it very inefficient to access files, particularly as the system scales. Therefore, the job of the file system - a layer of abstraction that hides the physical layout of the data storage from the operating system - must provide the following services with regard to these files:

- **Storage and organisation.** The most obvious requirement of the system, this entails the efficient distribution of files, each not necessarily contiguously, across the storage medium while organised into folders for easy access. The end user need not know that the file they are currently working on is not actually stored as a discrete block—in fact, it may be broken into several chunks stored at various locations in the medium. The system must trade off responsive file access to files for the user against efficient use of space on the storage medium. This view of isolating the user from the storage reality can begin to break down in cases such as in an environment where the files on the medium have become heavily fragmented—say, due to having been subjected to a lengthy period of continuous use and user reorganisation—or where a significantly large number of files and directories exist relative to the size of the actual files themselves, in which case the system can declare its free capacity to be surprisingly small as a result of so much space 'vanishing' for administrative purposes.

- **Retrieval.** Speed is most important from the user's point of view - as consistency and information corruption prevention are generally taken for granted by users! It is only in relatively recent times when accessing files across network links became feasible that speed became an issue again. Previous to this, it was safe to assume that file access time would be restricted to being a matter of milliseconds, given the physical proximity of storage to processor. Retrieval is also closely linked to hierarchy organisation, and given the geometric possibilities offered by just a few layers of nested directories, this is generally a satisfactory solution. When taking into account the requirements of modern, distributed file systems, however, it becomes clear that these assumptions do not necessarily hold any longer, and that considerable effort must be put into this area.

- **Naming.** The combination of a hierarchy of directories followed by a user-chosen name is the standard way to refer to a file. It is easy to see how restrictive this method is given the distribution of information today across, say, the Internet, where hyperlinks enable users to effortlessly jump around information sources in a much less constrained fashion. It is also unfortunately true that users are not good at the business of naming. As noted in [Nie96], files tend to suffer from the "'premature classification problem': the name normally has to be generated long before the content is created, and thus users may not fully understand what they're naming". In a distributed environment, this can be crucial where an arbitrary number of users need not be in agreement on a file naming convention. The naming of a file cannot be relied upon to convey a sense of its content. This creates untold problems for subsequent retrieval.

- **Security.** It is a file system's job in a multiuser environment to restrict access to files based on user privileges. Indeed, even in a single-user environment, is can be necessary to protect that user from inadvertently damaging important system files. This is achieved through the use of file attributes, governing at various levels of granularity the questions of who may read, write, delete and otherwise use files. In a distributed environment, extra issues must be considered such as eavesdropping on the network links during file manipulation, an aspect of everyday use that becomes ever more important with the current fast rise in the adoption of wireless network technology.

Such a system must be highly scalable and provide a relatively simple and efficient API for manipulating it. Such requirements have been well documented and, while not wishing to discount their importance, we choose to concentrate instead here more on those characteristics that distinguish the distributed aspect of a distributed filesystem. Such characteristics include:

- **Transparency.** This includes both location transparency and access transparency. The view of the filesystem presented to the user must not be dependent on either the user's physical location or the physical location of the actual files themselves.

- **Performance.** The necessity of network communication for ordinary file operations adds significant latency issues to operations that could hitherto be relied on to return almost-instantaneous results. This poses potential problems for time-sensitive applications.

- **Fault tolerance.** As with the previous point, network communication adds an element of uncertainty to file operations, uncertainty that can manifest itself as simple congestion delays or actual unbridgeable connections. Additional strategies must be employed to combat this factor, such as caching and replication.

- **Additional security requirements.** The distributed nature of the system adds other points of attack for intruders, an further consideration in an wireless environment.
- **Concurrency control.** Concurrency control, or locking, must be incorportated into the filesystem to contend with the added issue of processes on other machines vying for resources as well as local processes.

2.7 Converging distributed filesystems and wireless technology

This chapter has introduced the basic issues that concern the areas of wireless networking and filesystems research. It is probably clear from this discussion that these are two subjects that are not naturally reconciled. The environment that best suits the traditional filesystem is the polar opposite of that put forward by the challenges of the wireless network.

However, it is perhaps ironic that it is the stability and familiarity of the filesystem that is most needed in the chaotic, unpredictable world of the ad hoc network. Organising information in a community sense is essential if a coherent use is to be made out of such a dynamic medium. A filesystem offers a well-known and trusted foundation for organising a host of applications that depend on it. Many scenarios that examine potential applications of the WAND network would require, or at least benefit from, the presence of such a filesystem. The next chapter introduces some areas of current research in addressing this need.

Chapter 3

The competition - state of the art

3.1 About this chapter

This chapter analyses several file systems based on some of the characteristics outlined in the previous chapter. We focus mainly on NFS, Andrew, Coda and AdHocFS.

3.2 Some features of distributed file systems

Accessing information while on the move is a complicated process. To some extent, the filesystems outlined in this chapter all attempt to answer the question: what is different about mobile computing?

Most computer users nowadays are familiar with the idea of a distributed filesystem in the form of the fileserver model, which is in effect a use of the traditional client-server model. Instead of accessing files on a local hard disk, the user accesses a file across a network from a known machine, often using a transparency mechanism that allows a syntax closely mirroring that used for local access. In this way, programs running on client machines are able to easily choose files to work with both locally and remotely. This does require prior knowledge, however, of file location. As user requirements change, we are seeing many more computer usage scenarios involving unpredictably ad-hoc groups of computers coming together, most notably in WiFi environments, the given membership of which at a given moment tends to be highly volatile.

Several solutions have been developed to cater for the intermediate stages on the way to the latter extreme. As the use of computer networking has evolved, so has the number of solutions for information storage. In general, distributed file systems have been designed to suit the characteristics of the local-area networks that they run over. “Their design is adapted to the performance and reliability characteristics of local networks and hence they are most effective in providing shared persistent storage throughout an intranet.” [CDK94] This way of looking at distributed storage has been been extended a long way in recent years however, with both the scale of the Internet and the rise of wireless computing contributing to this. The internet is indeed the ultimate distributed filesystem, an enormous repository with almost a non-existent naming or filing system. On a smaller scale, efforts such as AFS (Andrew File System) and its successor Coda take considerations such as caching and replication into account using strategies previously considered unnecessary in more standard network filesystems. The tradeoff occurs in that these systems are more complex and require more maintenance, and hence can be not as robust.

3.2.1 Storage in ad-hoc wireless networks

We see today an explosion in WiFi capability in almost all handheld hi-tech devices, such as the latest, the Sony PSP. In such environments, what use are traditional filesystems, even the distributed sort? Given the tendency of peers to form, leave, regroup and partition networks it is clear that the old model of filesystem as shared repository needs a rethink.

The advantages of being able to access data while on the move are quite clear. Fast access to shared information gives competitive advantage to workers on the go. In emergency situations, quick access to medical records can be crucial. Telecommuting becomes far more flexible when considered in a ubiquitous computing environment.

Realising these scenarios is not a hardware problem--the technology certainly exists to address these issues (if we ignore inconvenient details such as battery power consumption problems.) A lack of standards and commonly-accepted software support is lacking. Certain problems still beset the wireless user—the fact that mobile devices boast poor resources compared to static ‘fixed’ devices, the danger associated with showing off expensive technology in public (theft, mugging, breakage), potentially volatile network availability and so forth. It is also worth considering the sheer growth in mobile device numbers—scalability is an unavoidable issue even in these early days of mobile applications.

Returning to basics, the following section gives a brief look at the most successful LAN-based distributed file system today, Suns NFS and gives a good overview of the basic distributed requirements for such a filesystem.

3.3 The Network File System (NFS)

Principally aimed at UNIX, NFS supports heterogeneous platforms by specifying a contract to be supported on participating machines. By designating certain areas of the local filesystem as being shared, a user of that machine can advertise what files should be available to other clients, and can browse the shared areas of all other machines. This of course requires individual machine intervention and implies that the system is only supported when the supporting machine is switched on. It also assumes some prior knowledge of network topology, or at least some search service bolted on top.

NFS provides a virtual file system, adding an extra virtual inode to denote a remote file. It mirrors the UNIX file system model where it can, except for the notable exception in that its operations are idempotent, i.e., it does not keep files open as UNIX does. Extra operations are provided to provide seamless operation path name translation, automounting and client/server caching for performance gains. NFS has been adapted for use with Kerberos for extra security against malicious attacks and unauthorised access. Overall, NFS has been a success, providing support for heterogenous platforms, good scalability and significant of user-perceived transparency in its operation. Its naming syntax is compatible with local pathnames, allowing existing applications to use it without loading extra libraries and undergoing rewrites. It also exhibits UNIX file system semantics, making for intuitive user operation.

3.4 The Andrew File System (AFS)

The AFS takes the abstract distributed file system model requirements a step further with its support for disconnected operation. This is achieved by using a caching strategy that leaves files on client machines until an update of some sort is made known to the server. While this minimises network communication, it throws up obvious consistency issues that require a robust protocol to administer. By using whole-file caching where possible, most client requests will be serviced by local copies. When a close operation or some equivalent is invoked, the AFS is responsible for updating the master copy back at the server. This approach is based on observation of file access behaviour on UNIX systems whereby locality of reference is strong in other words, most files on a system are not considered collaborative, and users tend to modify their own files and theirs alone in general—obviating the need for update resolution for long periods of time and so freeing network resources.

This is achieved in practice by having the AFS client (Venus) use a portion of the local filesystem as the shared AFS area, accessed via symbolic links from the rest of the filesystem, so providing transparency. Kernel modifications allow hooks into file system calls in order to redirect requests to the shared area if necessary. The main concern to the system is therefore cache consistency—if another user updates another cached copy of the same file, all other caches must be updated. This is achieved using a callback promise mechanism.

Numerous improvements and updates have been made to both NFS and AFS systems and their variants as computing requirements change. It should be becoming clear, however, that a postentially globally connected wireless world requires a fresh approach to the notion of an information repository. Indeed, [CDK94] notes that future requirements include support for mobile users with disconnected operation, and automatic reintegration and quality-of-service guarantees to meet the need for the persistent storage and delivery of streams of multimedia and other time-dependent data. The next section will examine current efforts that exist in this area and explore strategies for advancement.

3.5 Coda and others

Coda is a successor to AFS in that it aims to support more disconnected operation. It claims that it strives for constant data availability. [SKK$^+$90] It is resilient to failures through its two mechanisms of **server replication** and **disconnected operation**. In this way, caching sites may temporarily become server sites. A user may therefore work independently of failures in the network system. A related goal, as noted by [SKK$^+$90], is "to gracefully integrate the role of AFS with portable computers"—a concept of much interest to this document. Work in this area is currently continuing, mainly at Carnegie-Mellon University under Prof. Satyanarayanan [Mis], where projects such as Odyssey and Aura further these goals.

Much related documentation abounds in the literature that addresses the issues involved managing mobile file systems. [HH95] makes the point that proper management of disconnected operation is the key to efficient file management. Minimising traffic is not the goal, but "adapt[ing] to adverse network conditions". The authors discuss a modified form of AFS for their purpose.

Similarly, [HLM03] puts forward a proposed mobile, peer-to-peer information retrieval system. It is contended that ad-hoc routing is not robust, and that a sparse distributed filesystem will work best in an environment without it, where nodes will only contact each other directly. Fault tolerance is addressed through file replication. This sort of system, the authors explain, is aimed at a scenario such as, say, an emergency response, where timeliness is of the essence in one, reasonably confined, space. While a relevant and useful study, the findings and conditions for those findings do not match the envisaged test conditions for this dissertation which will involve the Wireless Ad-hoc Network for Dublin (WAND).

Chessa and Maestrini [CPM] concentrate on a more mathematical approach to file storage in an unreliable, ad-hoc network environment, which, by fragmenting files, boosts confidentiality. The system's main effort is to break away from the outlook of competing systems such as Coda and Petal, say, with their emphasis on adapting the client/server model for todays changed environment. While a technically innovative system, it does not specifically address the notion and issues surrounding file mobility that we are so interested in.

3.6 Extending beyond the filesystem - application adaptation

Coda is an application-transparent system—which is to say, it requires no changes to existing applications in order to run successfully. Applications need not be aware of the difference between a conventional filesystem and an application-transparent system. While this is certainly convenient, it has its disadvantages. Chief among these is the inability of an application to specifically tailor behaviour to the extra opportunities afforded by the mobile nature of the filesystem.

To explore this idea, researchers at Carnegie-Mellon are currently working on Odyssey—a set of extensions to UNIX for mobile computing. Odyssey is portrayed as an environment rather than a filesystem, a pervasive computing environment that would incorporate Coda as part of a greater whole. At the moment Odyssey represents the application-aware side of the area of mobile filesystems. It presents itself, most bluntly, as an API for resource negotiation. Underneath, the Odyssey system itself manages access to scarce resources. Its premise is that the application itself is best suited to deciding a course of action in the face of changing network resources, and offers the user/application of possible choices.

This usefulness can be seen in the example of video player which, if during playback the connection were to deteriorate, could switch to black-and-white or could drop the frame rate rather than simply stutter. An application opts to request resources **at a given quality level**. Odyssey informs the client of best available, and the client can then choose a resource level if it wants. The actual Filespace broken into 'tomes' (analogous to volumes), each of which holds objects of a specific type. Applications access files via the **viceroy**, which controls **wardens**, each of which is responsible for accesses to one particular object type. In this way, each warden can know about acceptable fidelity levels. Odyssey also uses a new concept known as 'dynamic sets', where groups of related files are grouped for prefetching. The application discloses hints to the OS/Odyssey about possible related groups of files, so the filesystem can perform fetch optimizations. This all sounds very esoteric, and it probably is. The abstruse nature of Odyssey documentation available would suggest that the system is still in the development stages.

A different approach to the problems posed by mobile, distributed operation is to modify existing solutions. In the face of weak connectivity, modifications to Coda have been implemented that address the following:

- **More compressed, robust protocols for cache validation**, including background servicing of cache validation. Coda originally relied on callbacks for update propagation, which is not a tenable strategy in an environment of weak connectivity.
- **Volume versioning** as well as file versioning--if no files have changed in a volume, then naturally the volume itself will not be changed, thereby implying that a coherency validation can often be reduced to a single date comparison which saves checking all files in a volume.
- **Trickle reintegration**--as opposed to a 'log optimizations' approach, which can be wasteful. Consider, for example, and update operation on a file followed by a delete operation on that file—the first operation is rendered obsolete by the second.
- **IOTs**—UNIX filesystems have no read-write conflict resolution. Satyanarayanan has proposed extensions of Coda with a solution based on isolation-only transactions (IOTs.) These may be used in a special IOT-aware shell to run programs, or can modify apps with an API.

All these Coda modifications are currently in deployment in real-world installations and have been reported successful. Drawbacks to these approaches mainly occur due to environmental factors—conflict resolution, for example, is good for text but not for comparing multimedia files. Caching also not useful if there is no temporal locality to exploit (i.e., the system is reliant on caching being useful.) This can hit performance if cache misses remain high, for example, given mobile clients doing a lot of searching on remote file servers.

3.7 The AdHocFS filesystem

The AdHocFS system [BI03] is a significantly interesting proposition in its use of a collaborative file sharing system that adapts to users changing network conditions. The system is written in Objective Caml, building on the UNIX ext2 filesystem, and utilises the ad-hoc group management capabilities offered by OpenSLP (Service Location Protocol.) It is of great interest given the similarities demonstrated in its network test conditions to those that, it is envisaged, will be those of the work of this dissertation.

AdHocFS claims to "realize transparent, adaptive file access according to the users specific situations" ([BI03]). In doing so, it provides collaborative file sharing to groups of users who are united by having the same security level. It is characterised by its policy of caching files on the filesystems of local nodes, in its use of naming and location services, in its use of secure links between nodes, and a coherency management service that organises replicated files and facilitates file sharing and updating.

Rather than concentrate on support for disconnected operation though predictive means (i.e. by prefetching), AdHocFS prefers instead to use a log-based system of update propagation, and an optimistic replication scheme. This enables ad hoc groups to come together to work on certain files, and then share those changes with other groups when connectivity is established.

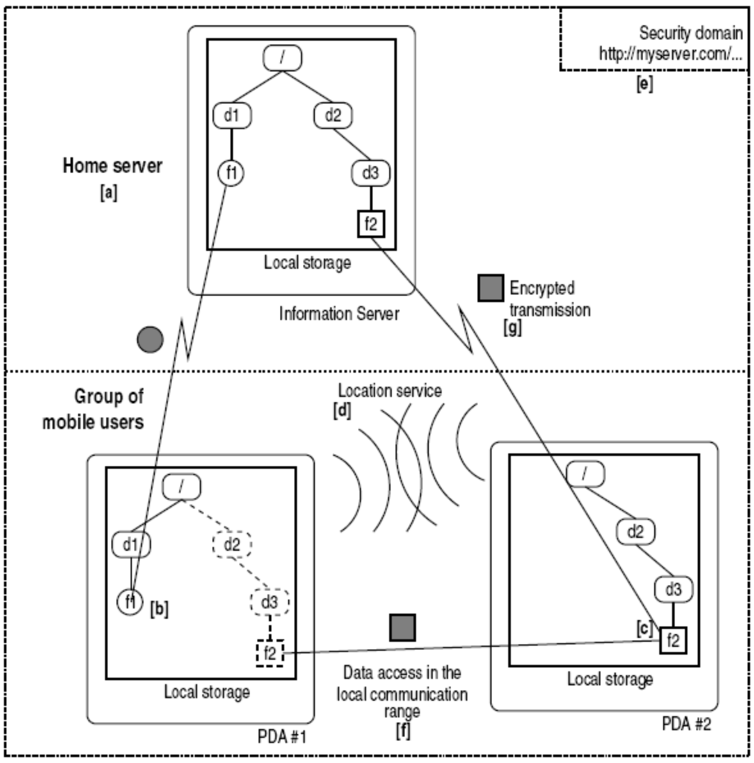

Figure 3.1: File access in AdHocFS. (From [BI03])

Figure 3.1 illustrates file access in the system. When a file is found not to be cached locally, it is necessary to connect to the home server and fetch it from there. Subsequent updates are applied to the local copy, and cache coherency management ensures that a coherent view of shared data is enforced across the system.

It will be seen in the next chapter, where the design for a putative file system for WAND is outlined, how certain of the more suitable aspects of this and previous systems have been incorporated into the architecture of the proposed system.

3.8 Current thinking on mobile computing research

If we return to the question that was posed earlier—what is different about mobile computing?—we may now suggest some answers. As before, it is the constraints imposed by mobile computing that dictate its characteristics. These constraints include caching metrics, semantic callbacks and validators, resource revocation, analysis of adaptation, and policies of global estimation based on local observations.

Mobile devices in wireless networks, as has been onserved, are relatively resource-poor. They are also physically vulnerable, suffer unreliable connections and are further constrained by a relatively weak power supply. For this reason, much research in the field points to the ability to adapt dynamically to circumstances the most important factor of mobile nodes.

Is this adaptation, therefore, best done by the individual application or by the underlying system? As with all things, this needs compromise. It depends on the goals of the application. Application-level transparency has been extensively tested through deployment of Coda and its subsequent modifications, as described above. Odyssey, on the other hand, explores application-**aware** adaptation—trusting in the agility of clients to respond best to changes in the network according to their needs.

This is redolent of the traditional balance between autonomy and interdependence in distributed systems. The addition of the mobility factor into the mix only exacerbates this. One of the bigger questions concerning the future of mobile computing, therefore, is of deciding the relative merits of application-transparent versus application-aware adaptation.

In the spirit of current thinking at Carnegie-Mellon, given their emphasis on Odyssey-based research, it will be shown that the Stirling filesystem tends more towards the area of application awareness.

Chapter 4

Design

4.1 About this chapter

In order to test the usefulness or otherwise of a distributed file system based on an urban ad-hoc network such as WAND, it is necessary to implement a test system that strikes a balance between the features regarded as desirable and the reality of the supporting infrastructure. The WAND architecture, summarized in a previous chapter, provides a backbone of sorts to the the target ad-hoc environment. This is immediately useful, as by its nature a distributed filesystem involves potentially very large chunks of data being transferred across its network. In an ad-hoc network, any member node must agree to act as a router should traffic come its way, and so there arises the potential for low-power, low-bandwidth devices being unwittingly coerced into acting as bridges for other nodes' large data transfers. By ensuring, through the WAND design, that a route will always exist between two non-WAND nodes that does not involve any other non-WAND nodes, this problem can be largely discounted.

This chapter outlines the features sought for such a filesystem, followed by discussions of implementation options (programming languages and platforms) and of the planned architecture of the result.

4.2 High-level design and goals

The goal of the implementation section of this dissertation is the creation and testing of a prototype filesystem that allows users to set extra attributes on their files, attributes that are geared towards taking advantage of a mobile ad-hoc network environment. For reasons of time and simplicity, this initial version limits itself to implementing the following two main areas:

- Minimum latency for time-critical access
- Virtual location of files

These attributes and their implications will be described in depth in later sections.

4.2.1 Filesystems in a mobile environment

Given that there is no management infrastructure required in a strictly ad-hoc mobile network, the idea of an organised filesystem does not fit well into this environment. Filesystems by their nature are repositories, designed for availability, stability, and to be well-known. These requirements are quite the opposite of conditions that are tolerated in the ad-hoc environment. A discovery protocol is essential in order to find out where files are stored at a given time, as is a replication protocol, if files are required to be persistent and available after their creator has left the network. Many, therefore, of the assumptions made when considering filesystem design must be reevaluated when porting these requirements to the wireless world.

4.2.2 Filesystems using WAND

The design of WAND eases these difficulties to some degree. By strategically arranging Linux boxes with ad-hoc wireless capability in a 'spine' layout throughout the city centre, each node is guaranteed one-hop communication with two of its neighbours, one 'upstream' and one 'downstream', so to speak. This way, any WAND node can be guaranteed a fast, efficient route to any other, and any transient device connecting to WAND's network can take advantage of this.

WAND nodes can essentially be regarded as fixed nodes in the ad-hoc network - cheating to some degree, but the preferable viewpoint is that this feature adds stability to the network. From the filesystem designer's point of view, however, this is most welcome. The presence of invariable, always-on and immobile nodes implies an excellent place to store the actual files that comprise the filesystem.

This frees user (client device) nodes from the responsibility of actually hosting the files themselves, and also goes a significant way to ensuring that user nodes will not have to bear the burden of large file transfer routing. Existing routes between WAND nodes should direct data along the 'backbone' if possible, rather than via client nodes. This model is illustrated in Figure 4.1:

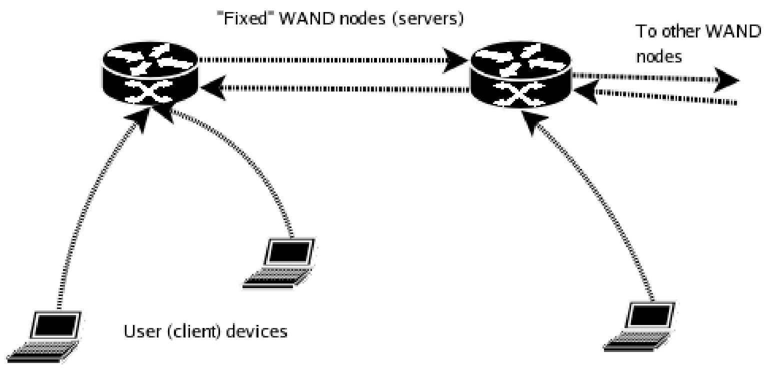

Figure 4.1: The basic mode of operation in the proposed filesystem.

Since this distinction is to prove so important in the subsequent design of the filesystem, it becomes convenient to refer henceforth to backbone WAND nodes as 'server nodes', and the user devices that move unpredictably between them carried by people in their vicinity as 'client nodes'.

4.3 The approach

4.3.1 List of features

The envisaged environment for the proposed filesystem system is of course a dynamic one, containing many mobile devices operating in an ad-hoc manner. The backbone system, WAND, whereby a set of ‘server’ nodes (connected in ad-hoc mode) form a fixed chain through the operating area, guarantees a path from one end of the system to the other. This backbone system can support a transient population of mobile, handheld nodes. These latter nodes can carry the rule sets for their users’ preferences, but it is assumed, correctly or incorrectly, that these nodes will not host actual files directly.

Instead, the files will be stored on backbone nodes (in a manner that is not necessarily related to the virtual filesystem layout.) The ‘server’ daemon application will reside on these nodes and will govern file inter-node transfers via sockets and AODV. A secondary client application (for users’ portable devices) performs the role of ‘file system API’. The system redirects ‘file open’ requests, ‘read’ requests, etc., to a server node, which obliges by accessing files on the virtual filesystem.

With these factors in mind, we can approach a list of features desirable for the filesystem:

- *Global view.* It should follow the UNIX-like model of a one-tree filesystem, starting from a ‘/’ directory, rather than incorporating the network structure itself into the naming convention (à la NFS.) This provides location transparency, and simple ease-of-operation, to the client.

- *Persistence.* If a user ‘creates’ a file on the filesystem, it must be subsequently available, file permissions allowing, to other users, after the creator has left the network.

- *Filesystem operations.* The filesystem should support traditional filesystem operations, such as open, read and close calls. Desirable features would also include move, copy and directory creation operations.

- *Permissions.* It must be possible for users to maintain a degree of access control over the files they create on the system. Files may be deemed accessible by all, or by a subset of the system's users.

- *Mobile attributes.* To make the filesystem truly useful and original, the files must support extra metadata attributes that take advantage of the mobile environment that users operate in. Such attributes include directing the file to physically 'follow' the user around the network, in order to guarantee fast (low-latency) access. This would be particularly useful for large media files. A different attribute would be one that allows a file to have a 'virtual location' instead, whereby users could 'carry' a set of objects around with them as they move through the network, ensuring that those files would only be visible to other users in the immediate vicinity.

As files do not physically reside on client nodes, the notion of 'available only at a certain location' means that files being 'carried around' in this way would be associated with one server node at a time. Client nodes that deem themselves closest to this server node of all server nodes would be granted access to these files - other clients would simply not 'see' them until they move into range.

The codename given to the subsequent development of this system was 'Stirling'. The name derives from a certain dog, native of Holland Park, Brisbane, whose distinguishing characteristic is that of following any available person around with various objects in his mouth - a perfect summary of how a distributed filesystem should present itself.

4.3.2 Basic design decisions

It was originally envisaged that the system would intercept file system calls (`fopen`, `read`, etc.) using a dynamically loaded library written in C, but subsequent investigation led to a shift towards a *middleware* approach. Early proof-of-concept demos and prototypes written in C led to the realization that, were a significant number of goals to be achieved in the timeframe available, a shift to a higher-level language would be needed. For this reason, the Python language was selected. The basis for this was its portability, ease-of-use and extensive built-in libraries.

4.3.3 About the Python language

Python is a high-level, object-oriented interpreted language. Strictly speaking therefore, it is a scripting language, a term which has pejorative connotations among the mainstream application developer community, which tends to regard a low-level feature set and performance considerations as being essential for enterprise-level requirements. Python, however, boasts advantages that counteract such criticism. Arising from originator Guido van Rossum's work in the late 80s on language design, Python has undergone subsequent development since its posting to Usenet in 1991. Its features include "modules, classes, exceptions, very high level dynamic data types, and dynamic typing" [VR] and, most crucially, a comprehensive set of useful libraries that are part and parcel of the language and are very simple to use.

These libraries include all kinds of networking protocols, system resource access facilities, string parsing (including regular expressions), multimedia processing, as well as introspection options for investigating the state of the Python interpreter itself. The Internet-related protocols libraries in particular are an excellent example of the ease-of-use provided by the platform; a customised HTTP server can, for example, be a matter of only a few lines of code.

This is only the briefest of introductions to the capabilities of the Python language, but for the sake of the requirements of this dissertation it proved that both the easy learning curve associated with the platform, combined with the sheer power of its built-in resources, was enough to prompt a move in implementation choice from C++. A certain amount of preliminary proof-of-concept work had already been completed in C++ when this decision was taken, but it was not enough or of a sufficient degree of complexity as to require a substantial degree of reimplementation in Python. Most of it concerned the interception of file system calls, a feature that was subsequently removed from the final system.

There is great scope for interoperating between Python and C, and hence there is the possibility of extending the scope of this dissertation to including the original aim of filesystem call interception. This is discussed in further detail in a later section of this document.

4.3.4 The filesystem view

The filesystem hierarchy presented to the user is that of a very pared-down UNIX system. It consists of one level of directories:

```
/stirling/public
/stirling/username1
/stirling/username2
...
```

The environment variable $\$USER$ is utilised by the client to identify the current user, and a directory with this name is used to store private file (not viewable by other clients.) Thus, to a given user `geoff`, the Stirling filesystem's hierarchy consists of two directories, `/stirling/public` and `/stirling/geoff`, the former containing an arbitrary number of files placed there by Geoff and others and which are accessible by all, and the latter containing files placed there solely by Geoff and which are only accessible by him.

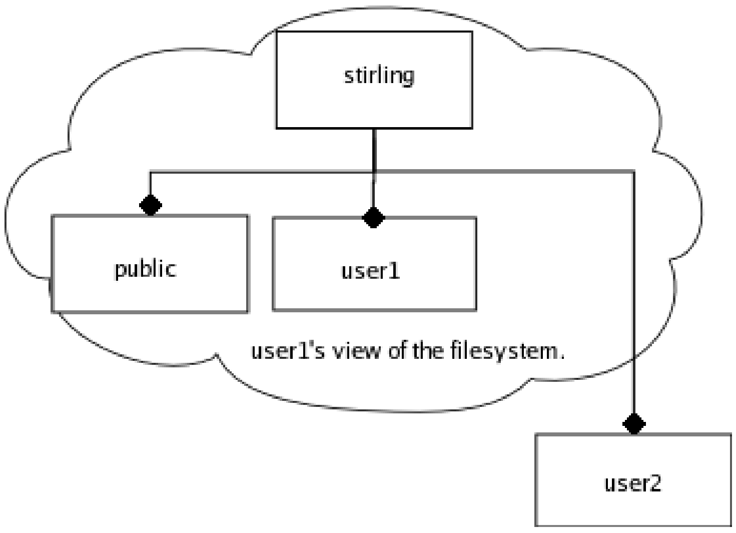

Figure 4.2: Stirling filesystem layout.

Given the time constraints of the project, it became apparent quite early in the design phase that implementing a fully-featured filesystem would be beyond the scope of the goals of the dissertation. For this reason, it was decided to restrict the filesystem to read-only files only, i.e., files could be deleted, but no write access would be allowed to those files while they resided on the Stirling filesystem.

4.3.5 The filesystem state

Stirling maintains state of the global filesystem view through an XML document. XML—the eXtensible Markup Language—is a framework for structuring universally-available data. It has become ubiquitous on the Internet, not least because of its widespread adoption in XHTML as the World-Wide Web's language of choice. Its use, however, extends far beyond web content presentation. It is intended to serve as the medium of structured information of the future, which includes traditional information repositories such as databases, spreadsheets and so forth.

Hence, an XML user may specify an information structure that suits a set of particular requirements, and no more. This may subsequently be used to verify that an XML document is 'valid', i.e. that it matches the required structure. Without this, the information parser cannot ensure that coherent meaning will be reliably extracted from the data. In this way, applications may depend on a universally-accessible data source. Critics have pointed out the high level of 'syntactic fluff' that necessarily pads out XML documents, especially if tag names tend to be lengthy, but this is typically offset in practice by cheap disk storage and effective compression algorithms, the latter of which can be found built into modern network transmission protocols such as HTTP 1.1.

Stirling specifies a set of tags that are used to maintain information about and organise the files that reside in its virtual filesystem. These tags include the file name, owner and location in the virtual file tree. Most usefully, extra attributes may be added to these file descriptions that reflect a user's desire that files behave in a certain way. This behaviour is intended to take advantage of unique features of the filesystem environment. In the case of Stirling, this is an ad-hoc network with a distinguishing backbone of nodes. The following section describes certain attributes that are appropriate.

4.3.6 File attributes

So far, the design has concentrated on availability and location transparency. Such characteristics, while essential for a distributed filesystem in the target environment, have been implemented successfully in many current filesystem solutions. Stirling aims to distinguish itself by allowing file *attributes* that allow the user to take advantage of the mobile nature of the network environment. As part of the filesystem API, users can ‘set’ or ‘unset’ these attributes on individual files or groups of files.

The prototype of Stirling concentrates on two possible mobile file attributes—virtual file location and transparent physical file location. This leads to the concept of a file being potentially in one of the states of *‘global public’* and *‘local public’*.

Global public / local public

In the ‘public’ directory, files are considered to be communal. Ownership as such is not an important attribute. A file created here is accessible by all. The distinction that is important, however, is whether or not the file is considered to have a ‘virtual location’ - that is, it may only be visible when the viewing client is physically present near a certain node. If it does not possess a ‘virtual location’ attribute, then it is visible to any user from any location, despite the fact that the actual file itself may well exist on only one node. In Stirling terminology, this attribute of ‘virtual location’ of files in the `/public` directory is referred to as being ‘local public’, whereas universally visible files are ‘global public’.

Users’ directories contain files that are visible only to their respective users. These files may optionally be set to have a ‘virtual location’ attribute as described above, in which case such files may be referred to as being ‘local private’. All other user directory files are ‘global private’ - meaning that the owning user, and only the owning user, can view and access them from any physical location.

Transparent physical file location

Another attribute that may be optionally set by a user on any file in either their user directory or the public directory is that of *following* the user. As the user roams between server nodes, the filesystem transparently copies the file to the server node closest to the user's physical location. This ensures a one-hop latency for file access, which may be of particular use for streaming multimedia files. The user's view of the Stirling file tree is, of course, unchanged.

Using this attribute on a file also has replication implications. The one file can therefore reside on several physical nodes at any given time.

4.4 Use cases

This foregoing description of the file hierarchy and its contained files' optional attributes is solely a technical one. To illustrate how this system may actually be useful, it is necessary to work through several use cases for this technology. Considering such scenarios is essential feedback for further refinement of system design.

4.4.1 The general environment

Users become Stirling clients when they enter the range of a WAND node. In practice, this implies a large portion of Dublin's immediate south city centre. The user interacts with the WAND via a wireless device equipped with 802.11 capability in ad hoc mode, and which is running Stirling-aware applications. The device automatically connects itself to the Stirling system. This process is described in detail in a subsequent section.

As the client roams throughout the network, the device continually scans for signal strength and manages the current connection to the nearest server node. Server nodes manage a list of currently-connected clients, and detect when a client has reconnected to a different node.

Examples of proposed usage scenarios put forward at an early stage of the project were as follows:

- Tourist guides providing streaming video to guests with handhelds roaming around a large area (a national gallery, for example)
- Video servers on public transport providing in-carriage entertainment, as the bus/train moves along a predefined route
- Provision of on-demand streaming music radio to roaming subscribers

A subset of these applications is discussed below.

4.4.2 Multimedia

As the technological capabilities of common mobile computing devices increases, demand for multimedia applications rises accordingly. Stirling clients may take advantage of the filesystem's ability to promote low latency by enabling the 'follow' attribute on latency-sensitive files (such as large 'streamed' audio or video files.) As the user moves between different WAND nodes, the system transparently moves the file to the nearest server node. In this way, the system guarantees a minimum transfer time for designated files, as they are located only one hop away from the user. This is important in preventing stuttering, or jitter, in playback.

This feature becomes useful when the communal 'public' directory is considered, as portable hard disk space is nowadays not such an issue that it becomes necessary to retrieve multimedia files in real time from a network, as the proliferation of personal music players such as Apple's iPod and its many rivals demonstrates. By using the 'public' directory, users may share clips, messages, etc. as needed with other Stirling-enabled users of the network. This opens the way to many possible applications using the Stirling filesystem; applications, however, that might have to consider copyright issues before public deployment.

4.4.3 Location-dependent applications

The idea that an application behaves in a certain way in response to the geographical location of its user is one that has tremendous commercial potential, and enjoys a current high profile thanks to the success of satellite navigation systems found in popular cars and trucks. Location-based services (or LBS, for short) have many other applications in areas such as mobile staff management, mapping, telecoms and GPS-based projects.

A filesystem, however, is a humbler beast in its scope and ambition, yet can provide useful location-based services. By allowing the possibility of assigning a file a ‘virtual location’, several potential applications emerge. Files marked in this way could be ’picked up’ by users in the network, and ‘set down’ again in a different location when the user chooses to do so when in that new location. This implies that other users of the network would only be able to interact with, or even simply see, that file when physically in the same location as the file’s virtual location. This has applications in mobile gaming. For example, a user might be able to ‘pick up’ a file called ‘TheFlag’ at the top of Dame Street, run down to the other end of the street and ‘set it down’ again. Users at the file’s former location would physically have to chase the first user to the new location in order to be able to see the file again. Files, therefore, may be used as game objects.

There are other areas where tailoring the filesystem view based on user location could be useful. Consider, for example, a large art gallery or museum exhibition, the experience of which is enhanced by handheld devices given to visitors. As visitors wander around different areas of the display space, the Stirling-enabled audio-visual application on the handhelds sees only the relevant content files for the location currently being examined. While this effect could be accomplished in other ways, the option of setting a virtual home location for files whose physical location is immaterial makes the application designer’s job that much simpler.

4.5 Stirling components & API

This section discusses the proposed architecture at a high level and explains how the major filesystem functions are implemented through the Stirling API.

4.5.1 Deployed components

Stirling consists of three major components:

- **Client library:** This is installed on devices that wish to access a Stirling filesystem. It consists of a library of functions that act as an API to the filesystem on WAND nodes and background housekeeping functionality.
- **Server daemon:** Installed on backbone (WAND) nodes, this daemon application services client requests and interacts with other Stirling server nodes for both file requests and XML state manipulation as necessary.
- **XML state server daemon:** Global filesystem state in the Stirling prototype is implemented using an XML tree, which resides on one designated WAND node. Other WAND nodes retrieve the state of the filesystem from this server. Although this commits the cardinal sin of using a single point of failure for a crucial system component, it must be noted that the full Stirling specification requires a consistency protocol, the precise nature of which is not fixed. Maintaining state consistency is a well-known issue in distributed systems and is the subject of much research and the target of many implementations. The structure and implementation of such a protocol is therefore outside the scope of this dissertation.

  The roles of these three components are illustrated in Figure 4.3.

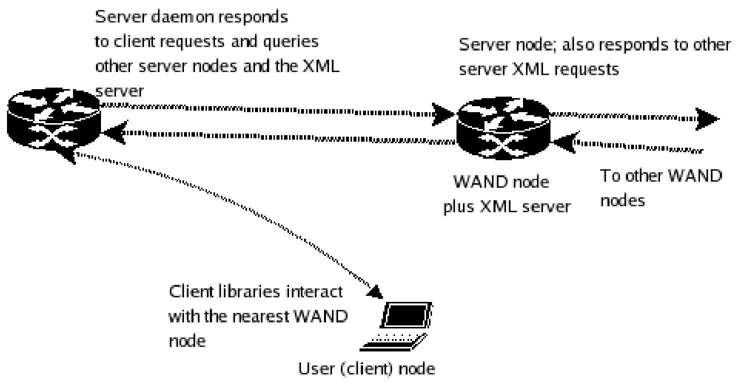
Server daemon responds to client requests and queries other server nodes and the XML server
Server node; also responds to other server XML requests
WAND node plus XML server
To other WAND nodes
Client libraries interact with the nearest WAND node
User (client) node

Figure 4.3: Stirling filesystem layout.

4.5.2 Usage and the API

Using Stirling requires an application to create a `StirlingFilesystem` object, part of the client library. Subsequent calls are made on this object, which is implemented as a singleton. The `startup` procedure begins the process of transparently joining the Stirling-enabled ad-hoc network. Calling the corresponding `shutdown` conversely stops this process. On calling `startup`, the client scans for server nodes and, having determined the closest, subsequently communicates with this node until a closer one is determined. This node becomes the current client's 'location'. At this point, API functions may be usefully called - they block if not connected successfully to the Stirling system.

It is desirable that the system act in as fault-tolerant a manner as possible. In a distributed environment such as this, which is heavily reliant on socket-based protocol exchanges, the possibility of error (and therefore deadlock) is quite high. To make the system as flexible as possible, thread pools are used for handing off individual tasks. While this increases the attention that must be given to system data vulnerable to concurrency issues, the advantages gained by ensuring that one error case does not hold up the correct running of the filesystem far outweighs the extra effort that must be invested.

The Stirling API consists of nine useful functions, aside from the required `startup` and `shutdown` procedures, listed in Table 4.1, and described briefly as follows:

Table 4.1: Stirling client API.

Command	Local-only	Alters state	Return value
stirlingcreate	No	Yes	Success
stirlingopen	No	Yes	File descriptor
stirlingread	No	No	Byte array
stirlingfpos	Yes	No	File pointer value
stirlingseek	Yes	Yes	Success
stirlingclose	No	Yes	Success
stirlingsetattr	No	Yes	Success
stirlingremove	No	Yes	Success
stirlingdir	No	No	File list

- **stirlingcreate(stirlingdestinationpath, sourcepath):** Strictly speaking, this call does not create a new file, but uploads the designated file to the Stirling filesystem (on the nearest WAND node) with the given destination name. The initial permissions granted to the file depend on the destination path: if its location is to be the user's directory, it will be private; whereas a new file in the public directory will be accessible by all. If the destination path does not specify either the public or the user's own directory, an error is raised.

- **stirlingopen(stirlingpath):** If the required path is valid and the file permissions allow, a file descriptor is returned that may subsequently be used to read from and close the file. The Stirling prototype supports read-only access. A permanent TCP socket is transparently set up from the client to the current server node for file reading. If the file is actually located at that server node, either by happy accident or because the file's 'follow' attribute is set for that user, subsequent transfers will be fast. If, however, it is located on another WAND node (the filesystem state XML server is queried for this information) individual reads are carried out across their own TCP connection to a source whose hop distance away is indeterminate. The design of a backbone such as WAND, however, guarantees that the route to the file will be as efficient as possible (over other WAND nodes) and will not burden other currently connected clients with the actual file transfer.

- **stirlingread(fd):** Read operations, as previously described, take place over a dedicated TCP connection between client and current WAND node, a connection which is only closed when the user explicitly closes the file. If it is necessary to fetch a read from the file located on a different server node, a separate connection for each read is used. This initially expensive-seeming approach is discussed in more detail in the next chapter, which deals with the implementation details. The 'file pointer', in any case, is increased with each read operation, and the bytes read are returned. Note that on the actual underlying filesystem, a Stirling read operation involves an atomic sequence of opening the file, seeking to the desired position, reading the required bytes, and closing the file again before returning.

- **stirlingfpos(fd):** The Stirling client maintains the state of the file pointer. It is not necessary to contact the server node for this reason, thus minimising response time.

- **stirlingseek(fd, newposition):** Similarly to `stirlingfpos`, this call manipulates the file pointer on the client side. Read calls pass this pointer as part of each operation.

- **stirlingclose(fd):** A close operation merely informs the server node that the client is done with the file. As read operations open, read and close the file in one go, this does not impact the actual underlying filesystem, but allows other clients potential ‘remove’ or ‘attribute change’ access to the file.

- **stirlingsetattr(stirlingpath, attributesetting):** As mentioned above, setting a file’s mobile attributes must be performed while it is not in an open state. Available in the Stirling prototype are two possible attribute settings: **follow** and **virtlocation**. The **attributesetting** parameter is of the form ”[attribute]=[true | false]”. An error is raised should the operation fail for any reason.

- **stirlingremove(stirlingpath):** Removes a file from the Stirling filesystem if permissions and current open state allow.

- **stirlingdir:** Returns a list of Stirling files visible in the directory specified.

The implementation of the filesystem object as a singleton ensures a safe and ordered approach to concurrent requests to the current server node. If multiple references to ‘StirlingFilesystem’ exist in an application that uses Stirling, this strategy guarantees that all those references will be to the one object. A locking strategy, as will be discussed further, keeps competing threads from creating interference problems.

The foregoing sections have outlined the requirements and a basic architectural outline of how this filesystem should be structured. This is the result of an extensive review of the literature of similar work in this area, and of the proposed operating environment for this filesystem. Having presented this black-box view of the system, the following chapter will discuss the implementation of the filesystem in detail.

Chapter 5

Implementation and evaluation

5.1 About this chapter

This chapter describes the work completed to implement and evaluate a working prototype of the filesystem specification in the previous chapter. The technical details include a description of the platform and equipment chosen for the project, explanation of the code by way of illustrative and class diagrams, discussion, and usage examples.

5.2 Technical background

The platform, development environment and associated considerations behind this project are largely dictated by the constraints that the target environment of WAND imposes. WAND operates as a set of 'minimal' machines running Linux (a relatively old version of Red Hat Fedora) with wireless capabilities configured in an 'ad-hoc' mode. For this reason, the primary development machine—a laptop supplied by TCD–was initially reconfigured to support a dual-boot with Fedora Core 2.92. This was chosen, first of all for compatibility reasons, as the WAND nodes run (at least) a version of this system, and secondly, because there is considerable support for the system both on the Internet and among fellow students undertaking similar dissertations. An evaluation period of several days was spent investigating the competing Debian system—a rival Linux distribution that has the reputation of being very stable, but difficult to install and configure. It was these latter aspects that prompted the reversion to Fedora, as configuring the wireless capabilities proved too time-consuming.

5.2.1 Wireless ad-hoc connectivity

Having successfully connected to College networks, the next step was to configure the wireless device in the machine as an ad-hoc node running an implementation of the AODV routing protocol, as required by WAND. Such an implementation was obtained from Uppsala University, Sweden, a version of AODV that is commonly recommended as being the most up-to-date and suitable for research purposes. Version 0.9 was obtained from their website at `http://core.it.uu.se/AdHoc/AodvUUImpl` and successfully compiled and installed. However, WAND itself runs Kernel-AODV, a competing implementation from the Wireless Communications Technologies Group (WCTG) at the National Institute of Standards and Technology (NIST), a technology spinoff of the U.S. Department of Commerce. K-AODV comprises a loadable kernel module that is designed to provide an AODV routing service for devices using WLAN connectivity equipment.

The version of K-AODV running on WAND at the time of writing is 2.1, which runs with the Linux kernel version 2.4. It complies with the IETF AODV draft as found at [PBRD03]. As the primary development machine available for the project is equipped with kernel version 2.6, preliminary testing with WAND was limited to basic connectivity testing with the WAND network using the the UU AODV implementation. Only on procurement of suitable test machines towards the later stage of the project could real compatibility testing with K-AODV begin.

Finally, it remained to set up a test connection to the WAND network itself. To achieve this, the following settings were used to obtain successful ‘ping’ results.

- The test machine is configured using `iwconfig` in wireless, ad-hoc network mode using essid ‘wand’ and static IP 172.16.x.x. In script form, the following executes the necessary steps:

  ```
  # Associate with the AP:
  ```

```
/sbin/iwconfig eth0 mode ad-hoc essid wand

# Set static address
/sbin/ifconfig eth0 172.16.10.8 up
```

- AODV should run in the background, with the machine's firewall disabled in order to enable AODV packet traffic uninterrupted passage, or at least, the firewall configured so as to allow AODV packets through. In script form:

```
#Stop firewall (temporarily)
/etc/init.d/iptables stop

#Start AODV
/usr/sbin/aodvd -i eth0 -d -r 3 &
```

- As WAND nodes' wireless transceivers tend to hop between channels, the 'local' node's channel is determined using `iwlist` and the configuration on the test machine set accordingly. In script form:

```
#Scan for the right channel
iwlist eth0 scan

echo "Above should be the current WAND channel.  Set the right"
echo "one using iwconfig channel x."
```

This was successfully achieved on a number of occasions while in WAND range. On all but one, however, connectivity was limited to a single hop. Factors for the cause of this include the 'channel hopping' phenomenon as mentioned above and transient signal blockage, such as that caused by passing road traffic.

5.2.2 The middleware approach vs. application-transparency

An early part of the implementation process was to investigate the feasibility of taking an application-transparent approach to the question of how the filesystem appears to the user. By intercepting system calls such as `fopen`, `read`, etc., no changes need be made to existing applications in order to use the new filesystem. Developers also do not need to learn how the Stirling filesystem works - it is simply accessed as if it were just another mount point on the Unix file tree.

This approach is as follows:

A set of wrapper functions is created in the C language which match the functions that must be intercepted. These functions inspect the path of the desired file, and if it matches a certain pattern agreed to be that of the special filesystem, the call is diverted to the appropriate Stirling function. If not, the normal system call is invoked instead with the same parameters.

The user, therefore, will not notice anything different about the system until paths to Stirling are passed to what were previously assumed to be normal system calls. To do this, the wrapper functions must be compiled with the following directives:

```
gcc -fPIC -rdynamic -c wrappingfunctions.c
gcc -shared -o libwrappingfunctions.so wrappingfunctions.o -lc -ldl
```

The `-fPIC` directive instructs the compiler to create position independent code; i.e., code not firmly locked into memory. `-shared` tolerates unresolved functions, as there is no `main` function required. `-c` tells it to only produce object files, which can then be bundled into a dynamic library with the second invocation of `gcc`.

If the environment variables `LD_LIBRARY_PATH=[path to .so]` and `LD_PRELOAD=libwrappingfunctions.so` are subsequently set, any calls to functions defined in this library will be directed there, hence intercepting what would otherwise be system calls.

This approach can work, provided that all possible file-related system calls are covered consistently in this way. It does, however, require that the library and environment variables be deployed on all target machines. The new filesystem code must also be accessible from these wrapper functions, ideally through C++ also, but there are several solutions to language interoperability issues.

The alternative, a middleware-based approach, was chosen, however, for several reasons.

Firstly, the choice of Python as a primary development language made it difficult to interoperate with a filesystem call approach without significant project time investment in Python-C extensions. Given the timeframe of the project, it was preferable to concentrate on implementing areas of direct interest to the research at hand rather than sideline details. Python is an excellent language for rapid prototyping, and perfectly suited to the network programming needed for a short development window. Finally, the necessity of extra deployment on all client machines balanced out the complementary disadvantage of the 'extra layer' of the middleware approach. It is comparatively easy to create demonstration applications with Python that use the Stirling API.

5.3 Implementation overview

The vision of this project is to enable network users to see and interact with a location-transparent file system, and to tag files which they desire should be available with minimum latency, or should be associated with one particular location. To achieve this, the system consists of a distributed application running on every network participant. The application runs as a daemon (background) process on WAND (server) nodes and as a user-callable middleware library on user (client) nodes. A further component maintains state across the application's network scope.

5.3.1 System state

Maintaining consistent filesystem state could be achieved using something as complex as a database, or as simple as an XML file store with, say, a Web page frontend through which users could manage their files. The files themselves are stored dormant on arbitrary server nodes until such time as a Stirling user becomes mobile on WAND, but system state must be universal.

This is a suboptimal solution given the network's ad-hoc nature. Although the WAND nodes' continuous presence are a guaranteed feature of the network, failure of just one results in an immediate partition in the backbone network, and requires the participation of client devices for subsequent routing between the two partitions.

Therefore, the final Stirling system will maintain state using a distributed consistency protocol. This subject is a substantial research area in its own right, and sources such as [SMG98] and [KS96] provide examples of how such an approach may be utilised. For the purposes of an illustrative prototype, however, the system uses a centralised approach based on XML, as will be described in a subsequent section.

5.3.2 The server node daemon

It is the server daemon's job to service requests from clients related to the Stirling API, and to initiate requests to server daemons on other WAND nodes for remotely-located files. Its ambition is to remain as stateless as possible for stability reasons. It passively accepts incoming connections and uses thread pools of workers to service these requests. This is discussed in detail in Section 5.4.

5.3.3 The client node library

The client library consists of a set of functions called by client applications, and a background thread which is used to continually scan for the closest available WAND node, connect to it and maintain a heartbeat sequence to it, until the client shuts down or another WAND node is deemed to be closer as a result of the user moving around the geographical space of the network. This is discussed in detail in Section 5.5.

5.3.4 The XML system state manager

Located on a single WAND node, the master XML manager controls the filesystem XML state tree, stored in-memory during operation and as a file while not. It acts as a server for incoming state commands. On all other WAND nodes, the XML manager transparently creates socket connections to the master XML manager node. This is discussed in detail in Section 5.6.

5.4 Operation of the server node daemon

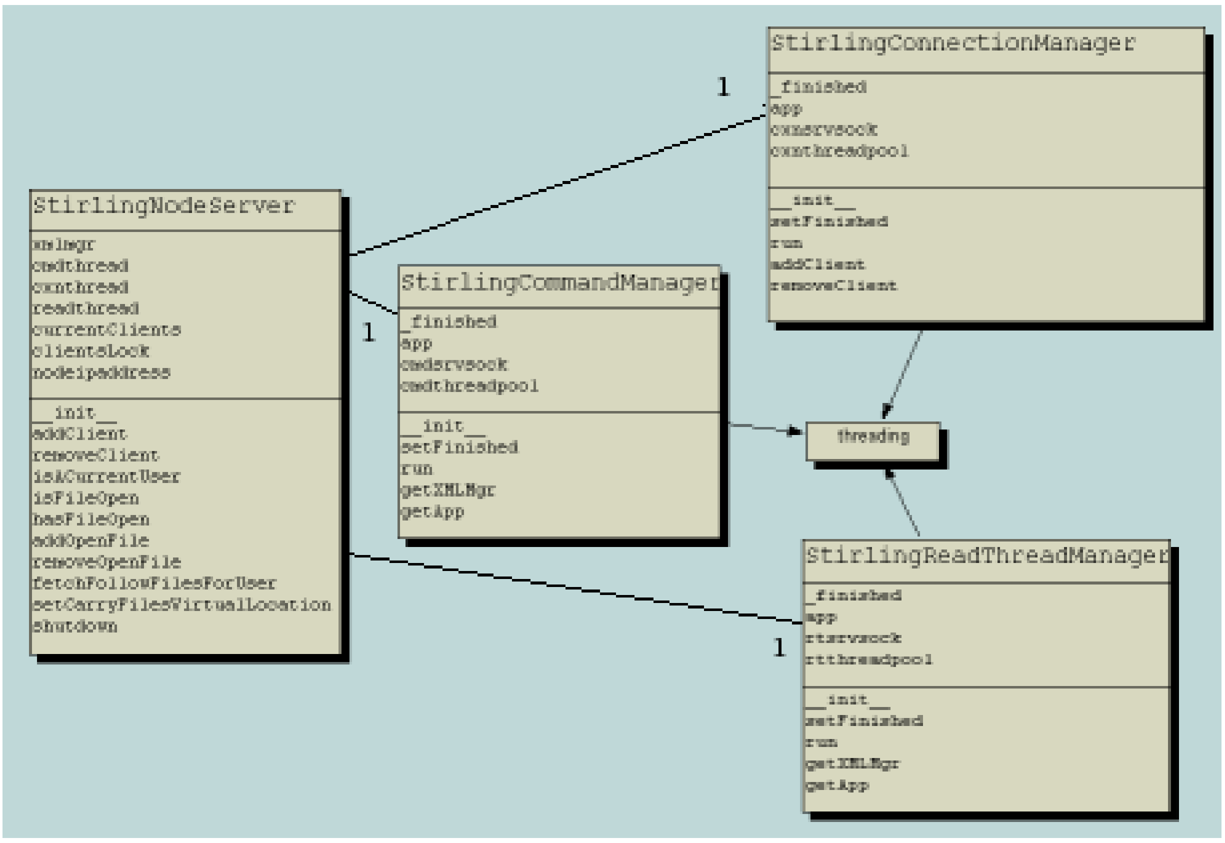

Figure 5.1: Stirling server node class view.

The server daemon's class layout is depicted in Figure 5.1. It accepts incoming connections on three ports, as illustrated in Table 5.1.

On startup, the application devotes three threads to servicing socket connections for client connections, file read connections and command connections, respectively. Only one server node acts as the master filesystem state XML server - it devotes one more to incoming state commands. All use a separate component for managing access to filesystem state. This is discussed in depth in section 5.6.

Table 5.1: Stirling's use of socket ports.

Connection type	Port number
Stirling client connection requests	8888
File read connections	8889
Stirling commands	8890
XML state commands (XML server only)	8891

5.4.1 The client connection manager

The first of these server socket threads deals with Stirling clients who have chosen on the current server node as being the closest node to them, and therefore serves as their 'location' in the filesystem's network. Incoming connections are handed off to a thread pool worker, which accepts messages of the form in Table 5.2, initiating the protocol exchange.

Table 5.2: Client login message

Message component	Length
STIRLINGLOGN	-
User	20
User's last server node	20
User's open files (length)	12
User's open files	above

Stirling messages consist of a 12-byte message type, followed by the required data. Where possible, numbers are sent as zero-padded 12-byte strings, and Stirling ID strings are padded out to 20-byte strings. Other forms of data are preceded by their length (in the padded 12-byte form) so that the receiver may obtain it correctly.

The user's open files are sent in the form of a python 'pickled' list. 'Pickling' in Python is similar to object serialization in Java and, since the output is a string, is very useful for sending over sockets. It is necessary to inform the server node about the currently open files, as the client will be using a separate read socket for each file to the new server node. By incorporating this information into the login process, this saves a separate XML state lookup.

If the client information is error-free, the server replies with a single ’STIRLINGACKN’ message, followed by a one-byte ‘1’ to indicate login success. An error state results in a ‘0’ being sent instead. If successful, the thread becomes a heartbeat receipt thread, listening for ‘STIRLINGHTBT’ messages from the client. If a certain period elapses without such a message (currently set to thirty seconds) the server assumes that the client has left or moved on to another node, and removes that client from its list of current clients.

All server node threads use threading locks to guard access to concurrency-sensitive system information such as the server’s current client lists. Python makes this comparatively easy to do, with one-line synchronization primitives.

5.4.2 The command manager

Like the connection manager thread, the command manager also accepts connections and hands off tasks to a thread pool. However, the range of message types it can deal with is much larger. It effectively deals with the whole gamut of Stirling API calls using, as above, a thread-per-request model. Where it principally differs from the above, however, is in its short-lived threads. While connection threads become heartbeat servers, command threads have a very short lifespan thanks to the ‘statelessness’ goal aspired to by the Stirling system.

The various protocol exchanges dealt with by the command manager are as follows:

- **STIRLINGCREA**. Users create a file by issuing a `stirlingcreate` command, which initiates the protocol exchange illustrated in Tables 5.3 and 5.4. The latter is sent following a successful STIRLINGACKN back to the client. The STIRLINGFILE protocol data unit is reused in other protocol exchanges in Stirling, including server node $\Longleftrightarrow$ server node data transfers.

  The file is created locally according to the desired file path hierarchy in the STIRLINGCREA message, and the XML file state updated to reflect the new file and its physical location.

- **STIRLINGREMV**. Removing a file from the system involves a message similar to STIRLINGCREA, as shown in Table 5.5. After querying the XML manager to enquire after the file’s status, a STIRLINGACKN is returned to mark the success or failure of

Table 5.3: Create message

Message component	Length
STIRLINGCREA	-
User	20
Path length	12
Path of new file	above

Table 5.4: File message

Message component	Length
STIRLINGFILE	-
Path length	12
File path	above
Full file flag	1
File start position	12
File transfer size	12
File chunk	above

the operation. Insufficient permissions, or the file being open are examples of how a ‘remove’ operation could fail.

Table 5.5: Remove message

Message component	Length
STIRLINGREMV	-
User	20
Path length	12
Path of new file	above

- **STIRLINGOPEN**. Again, the ‘open’ message resembles STIRLINGCREA. An ack is returned to the client and, if the user is allowed open the file, a ‘file descriptor’ message— a STIRLINGFDSC followed by the file descriptor integer. If successful, the client will open a read thread socket for that file, as described in Section 5.5. This thread’s work, however, is done at this point.

Table 5.6: Open message

Message component	Length
STIRLINGOPEN	-
User	20
Path length	12
Path of new file	above

- **STIRLINGATTR**. The path of the file is passed along with the new attribute in [attribute=true|false] form (see Table 5.7.) If permissions allow, and the attribute type is correct, the appropriate function in the XML manager is called. This will either be ‘setFollow’ or ‘setCarry’. Success or failure of this operation is returned to the client via a STIRLINGACKN message.

Table 5.7: Set attribute message

Message component	Length
STIRLINGATTR	-
User	20
Path length	12
Path of new file	above
New attribute length	12
New attribute	above

- **STIRLINGLIST**. As shown in Table 5.8, this is a simple exchange that returns a STIRLINGDRCT message to the user that passes a pickled list of directory entry strings back to the client. Again, permissions play a role in the returned list, as does the question of the ‘virtual location’ attribute of files ostensibly in that directory. Files with the ‘virtual location’ attribute set and having a location that doesn not match the requesting user’s will not show up on the returned list.

Actual reading of the files themselves is, as described above, handed off to separate connections for open files. How client$\Longleftrightarrow$server file transfers are effected is described in the following subsection.

Table 5.8: Directory listing message

Message component	Length
STIRLINGLIST	-
User	20
Path length	12
Directory path	above

5.4.3 The read thread manager

After a successful $stirlingopen$ call, a client will create a connection to the server node's read thread manager in order to set up a persistent connection for subsequent read operations on that file. This connection will remain active until the client closes the file or the client moves on to another node. In the latter case, the client will transparently recreate this 'read connection' with the new server node.

Contact is initiated with a simple handshaking protocol as shown in Table 5.9. This message also indicates whether the connection should remain open (PERM) or be torn down after the first transfer (TEMP). This is used because server nodes use this protocol between themselves to transfer files between server nodes as needed, and this transfer does not require a permanent connection.

Table 5.9: Read thread initializing message.

Message component	Length
STIRLINGHELO	-
User	20
File descriptor	12
Permanent/temporary flag	PERM | TEMP

Once this is successfully negotiated, the thread begins to service STIRLINGREAD and STIRLINGCLSE messages. The latter, a 'close' message from the client, causes the thread to exit gracefully. The former initiates an exchange, the complexity of which depends on whether the server node has to fetch the required file from another server node or not. The

STIRLINGREAD message is illustrated in Table 5.10.

Table 5.10: Read message (client to server)

Message component	Length
STIRLINGREAD	-
User	20
Full file flag	1
File descriptor	12
Path length	12
File path	above
File position	12
Bytes requested	12

If the file is indeed located on the current server node (determined by a check of the XML filesystem state), the protocol continues by opening the file using the regular system call, reading the required data, and streaming it back to the client using the STIRLINGFILE protocol data unit (see Table 5.4.)

The process is a little more complex if the required chunk of the file has to be fetched from another server node. However, the complexity is limited to reusing the existing protocol. In this case, a new read thread connection is created to the appropriate server node, initiating the transfer as before with a STIRLINGHELO message (Table 5.9) but, in this case, marking the connection type as temporary. Unnecessary complexity ensues if such a connection were to be kept open, and its attendant housekeeping difficulties. Instead, the STIRLINGREAD is issued as per Table 5.10, and the returned STIRLINGFILE data parrotted down to the client as if it had been read locally. It only remains, on completion of the ‘read’ transfer, to issue a STIRLINGCLSE message to the remote server node to end the temporary ‘read’ connection.

5.5 Operation of the client node library

The client consists of a set of library functions, along with a connection thread that is active when the system is started up. This section explains the function of the connection thread

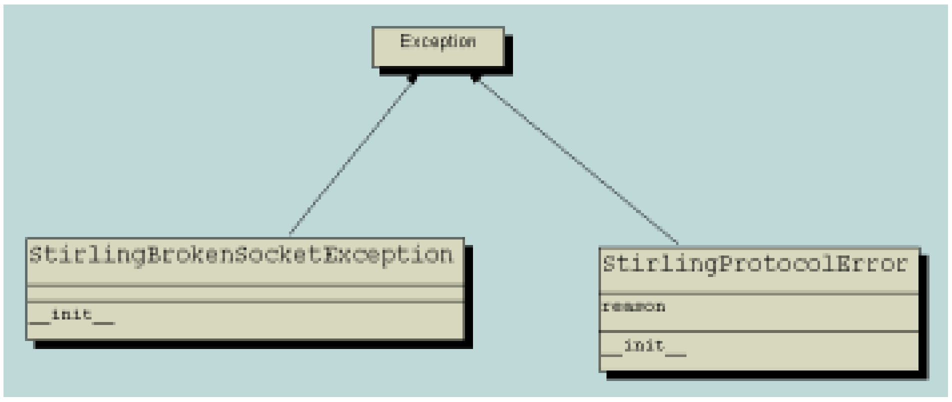

Figure 5.2: Stirling exception classes.

and gives a runthrough of the processes involved when using the API function calls.

The client is used by creating a StirlingFilesystem object, which may then be initialized by calling its `startup()` function. This initiates the connection process to the nearby WAND network. The filesystem may not be used until such a connection is made. Calling `shutdown()` ends this process.

5.5.1 The connection thread

The connection thread becomes active when the filesystem is started up via the ‘startup()’ function. Its role is to intermittently scan for available WAND nodes, compare their signal strengths and connect to the ‘closest’ (determined as the node with the highest signal strength.) Having successfully ‘logged in’ to the closest node, the thread sends heartbeat messages at fifteen-second intervals on the same login connection. There is no explicit ’leave’ protocol that the client uses. Should it determine that a different node is closer, it simply connects to this new node. The old node will detect the absence of heartbeat messages and remove the client from its list.

This process is illustrated in Figure 5.4.

The scanning process involves parsing signal information from K-AODV given in the `/proc/aodv/signal` file. Using a list of known WAND nodes to whittle away other client users that may inadvertently show up on the signal strength list, the client thereby

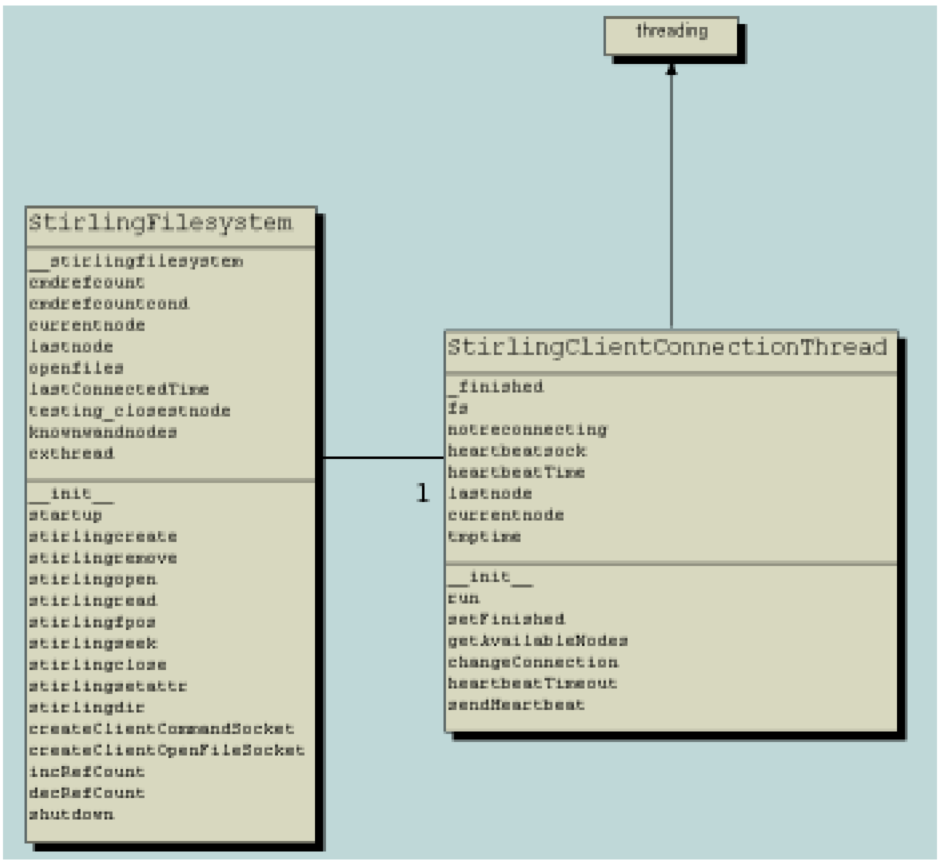

Figure 5.3: Stirling client class view.

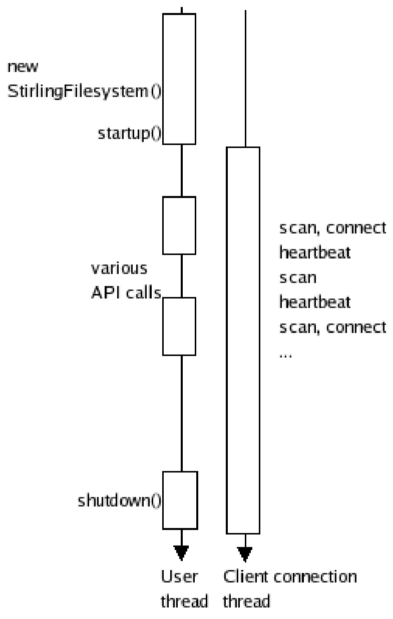

Figure 5.4: Stirling client life cycle.

decides its own location. To ensure that frequent logins back and forth do not take place when the client is in a border region between two WAND nodes, a ‘connection last used’ heuristic is used to impose a reconnection time limit for nodes to which login connections have recently been used.

The login protocol itself has been described in detail previously in Section 5.4.1.

5.5.2 The API

In order to prevent concurrency problems arising from interference, the Stirling client uses a system of reference counting. API calls are not implemented as synchronized methods, and they therefore may be interleaved. A reference counter keeps track of the calls in progress at any time, and if the connection thread signals that it wishes to reconnect, no further API calls will be allowed to proceed. The connection thread waits for currently running API calls to finish (using the `threading` library’s `Condition` construct) and then reconnects. Blocked API calls may then proceed.

Creating a file: stirlingcreate

The file creation process on a Stirling filesystem is illustrated in Figure 5.5. A more realistic term for file 'creation' is probably file 'uploading'; that is to say, as the Stirling prototype is a read-only file system, files are created in a Stirling directory as-is, with no further modifications possible. The file is created on that server node in the appropriate subdirectory, and the XML state is updated to reflect this new file's presence and the server node on which the file is actually located.

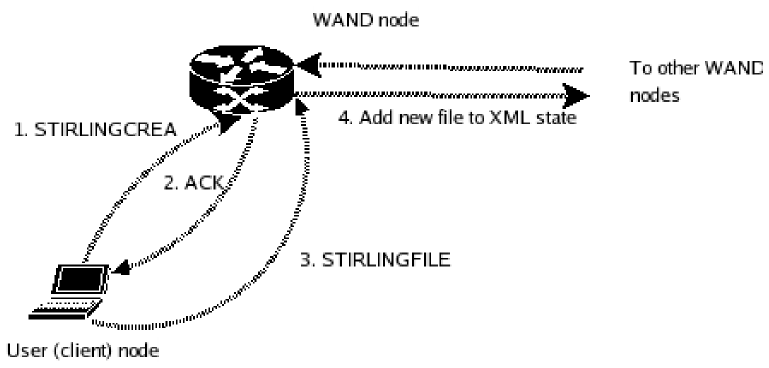

Figure 5.5: The stirlingcreate protocol.

An error is raised if the source file is unopenable, if the user has insufficient permissions to create the file, or if a transmission error occurs during transit.

Removing a file: stirlingremove

Removing a file is illustrated in Figure 5.6. An error is raised if the file does not exist, is open, or its permissions do not match the requesting user's. While it is desirable to actually erase the file itself if it is on the server node's local filesystem, this is not essential. Removing it from the XML state, however, is.

Opening a file: stirlingopen

Opening a file is illustrated in Figure 5.7. An error is raised if the file does not exist, is open, or its permissions do not match the requesting user's. A successful acknowledgement

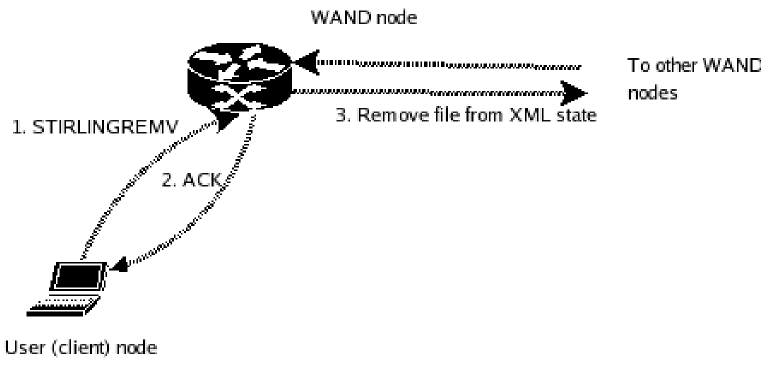

Figure 5.6: The stirlingremove protocol.

message back to the client is followed by the newly open file's file descriptor. The client then uses this file descriptor to open a separate socket connection for subsequent read operations on this file.

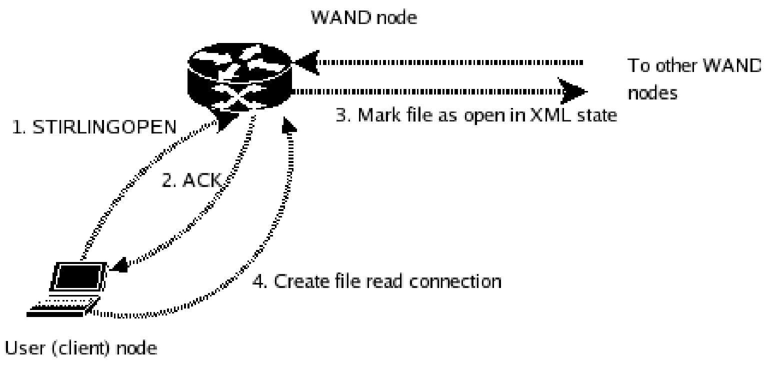

Figure 5.7: The stirlingopen protocol.

Reading a file: stirlingread

Reading a file is illustrated in Figure 5.8. A separate socket connection will already have been created for this purpose, and all protocol messages take place over this connection rather than over the command connection.

If the location of the actual file to be read is on the current server node, the desired chunk of the file is sent back to the client over a STIRLINGFILE message. If not, an intermediate protocol exchange takes place between the current server node and the actual file's host before it can be returned to the client.

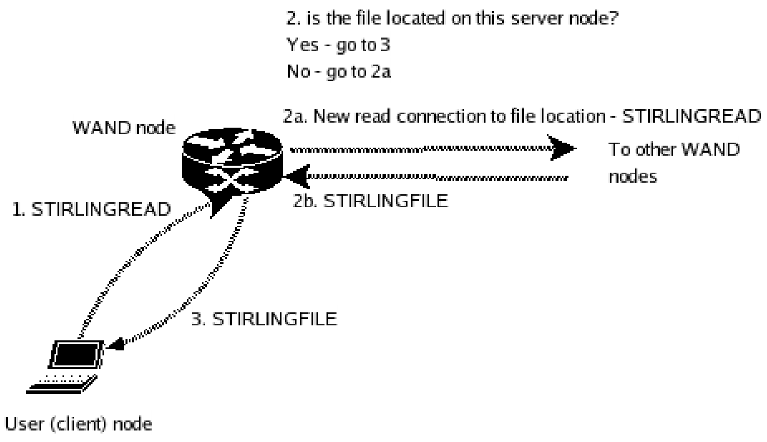

Figure 5.8: The stirlingread protocol.

Closing a file: stirlingclose

Closing a file is illustrated in Figure 5.9. It is an unacknowledged operation.

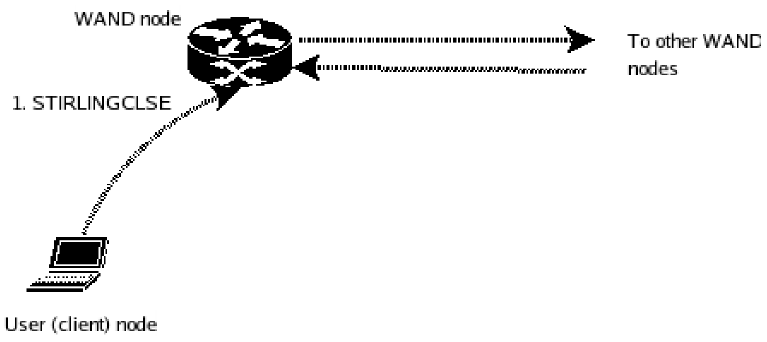

Figure 5.9: The stirlingclose protocol.

Obtaining a file's read position: stirlingfpos

Files' read pointers are also stored client-side, obviating the need for a protocol exchange with the current server node.

Seeking to a read position in a file: stirlingseek

Seeking to a read position involves only altering the client-side read pointer for the file in question.

Setting a file’s mobile attributes: stirlingsetattr

The process of setting a file’s attributes is illustrated in Figure 5.10. It is an acknowledged operation. An error is raised if insufficient permissions prevent the operation being successfully completed. The new settings take immediate effect, in that files set to ‘follow’ for the current user must be immediately fetched to the current server node, and directory listing operations must respect new ‘virtual location’ settings on files.

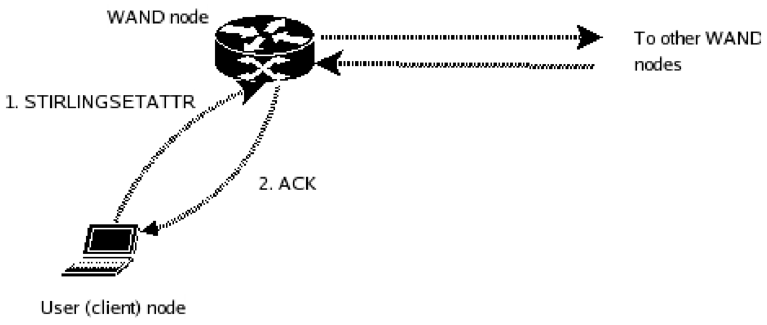

Figure 5.10: The stirlingsetattr protocol.

Getting a directory listing: stirlingdir

Getting a directory listing is illustrated in Figure 5.11. An error is raised if insufficient permissions prevent the operation being carried out.

5.6 Operation of the XML state management subsystem

It is very important that concurrent access to the filesystem state is carefully controlled. As the system presents itself as a single XML file (at least, in the current prototype version)

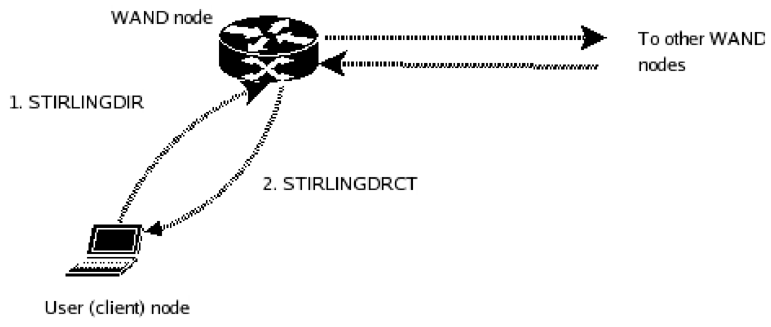

Figure 5.11: The stirlingdir protocol.

it would be very easy to introduce inconsistency through not only multiple threads on the same machine vying for update access, but also though server nodes on other host machines seeking update access over socket connections.

The solution used by Stirling is to use a 'StirlingXMLManager' object, which is the interface through which all server nodes access the filesystem state, whether for read or write access. The manager defines the following useful functions:

- addFile
- allowedRemoveAccess
- allowedAttrAccess
- allowedReadAccess
- addOpenFile
- removeOpenFile
- removeFile
- getNewFd
- queryFileLocation
- setLocation
- setFollow

- setCarry
- getFollowFiles
- getCarryFiles
- setVirtualLocation
- getDirListing

A discussion of these functions should not be necessary - their meanings should be reasonably clear given the discussion of the modules in the previous sections. From an implementation point of view, the interesting aspect is how these functions obtain and manipulate the state XML.

5.6.1 XML access

The StirlingXMLManager uses an 'xmlaccess' member variable that uses a very simple interface: get, set and release. The get and set operations obtain a read and write lock, respectively, on the filesystem state. The release function frees whichever lock was invoked. This interface is implemented by a 'ServerXMLAccess' class on the designated XML server machine (where the core XML state file is located) and by a 'ClientXMLAccess' class on all other server nodes. The interactions between the components of the subsystem are illustrated in Figure 5.12.

The 'ServerXMLAccess' object acquires/releases the lock and changes the state directly, and maintains another thread pool server besides the usual server node managers for connections, commands and read threads. This additional thread pool services requests from other server nodes sending requests via the 'ClientXMLAccess' class - transparently, it provides the same service as its 'Server' counterpart, but does so using a subset of the Stirling command protocol to make such requests over socket connections. In this way, concurrent access to the current filesystem state is restricted using a single point of access to this state.

These remote requests mimic the local operations as shown in Table 5.11.

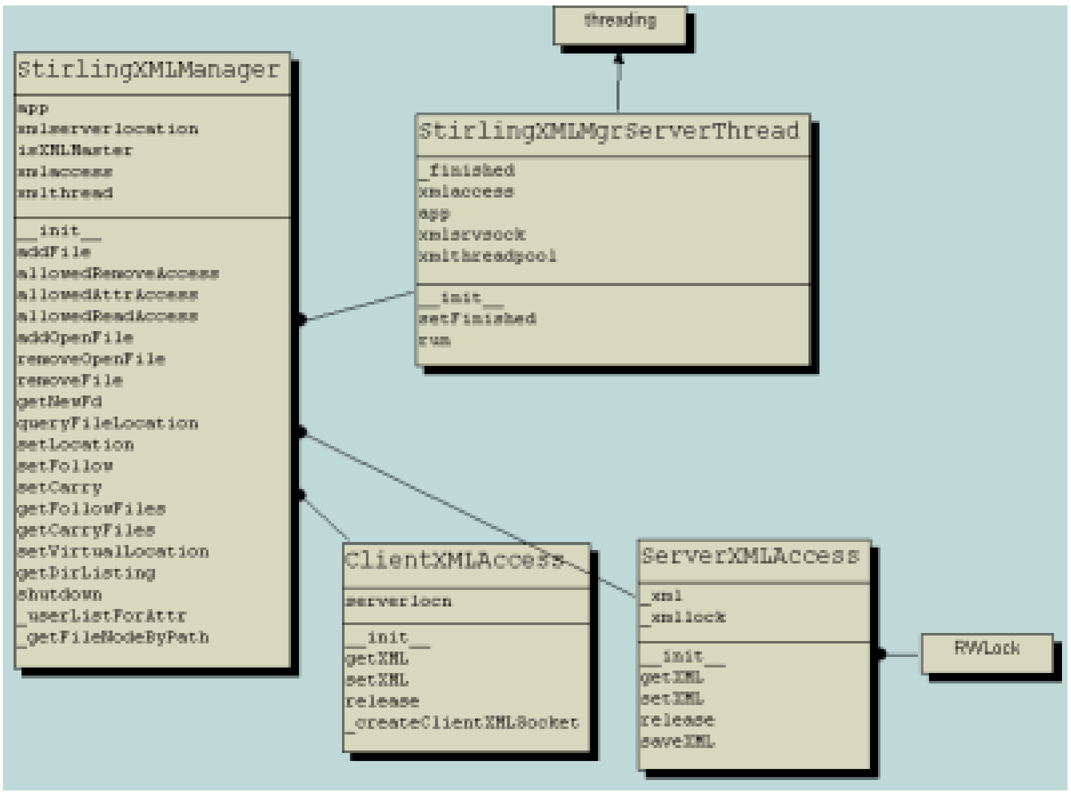

Figure 5.12: Stirling XML manager subsystem.

Table 5.11: The correspondence between local and remote filesystem state commands.

Local function	Corresponding message type
getXML(readonly)	GETRDX
getXML(write)	GETWRX
setXML	SETXML
release	RELXML

5.7 Implementation issues & discussion

The above was implemented over a period of three weeks of full-time coding. It was approached using the ethos of 'test-driven development' — an incremental approach to adding functionality and features.

The use of Python as a primary development language had its advantages and disadvantages. Its simple yet powerful syntax, as well as its comprehensive libraries, meant a very productive return on development time invested. However, programming with Python was a new experience and so the project did not benefit from the use of a familiar language, with the pluses that prior experience in a language or platform can bring. The development platform of Linux also was a learning experience. In the long term, this has been an extremely beneficial process, but at the expense of short-term investment in basic skills learning that ate significantly into project development time.

As is the case with all distributed systems development, real-world evaluation is a trickier issue than is usually faced in more traditional systems development. The feasibility of testing on a network like WAND is a significant obstacle, wherein a system that requires remote configuration and must be shared with many other users implies a far more stringent testing environment.

For this reason, 75% of testing time was restricted to two machines, rising only to a network of four nodes when suitable equipment for testing using ad hoc mode and routing protocols became available. It would have been ideal to have had the opportunity to test some of the Stirling functionality on WAND, but network instability stepped in to preclude that intention. In hindsight, given some of the results of testing on a 'mini-WAND' network, this may have been no bad thing as there exist certain areas of Stirling where fault tolerance ought to be given extra attention before deployment on a sensitive, shared network such as WAND.

5.8 Evaluation

Effective assessment of the usefulness or otherwise of a filesystem involves continuous monitoring of system performance over months of use, through periods of varying load. This luxury is unavailable to an experimental filesystem with limited testing time. Therefore, a compromise set of criteria must be established that allow for substantial exploration of its feature set.

An early goal of system evaluation was to deploy the application on WAND nodes, but this intention was abandoned due to necessary maintenance work being carried out at the time on WAND.

System testing was limited to a test suite of three machines of the following configuration:

- Dell Latitude D400, running a Red Hat Linux distribution using kernel version 2.4
- Kernel-AODV version 2.1 installed
- Cisco Aironet 350 series wireless card as principal transceiver
- Python 2.2

These units were arranged at points such that distances between them varied from 2 feet to 30 metres. Transmission power output on the cards was deliberately lowered in order to test AODV routing.

The tests used on the Stirling filesystem were as follows:

1. **Routing capability.** While not strictly part of the test suite, as K-AODV is a third-part component, routing capability is an important part of ensuring the stability of the

Stirling system. It should provide a transparent service, such that sending a packet to an existing IP address should not be an operation that requires special user space handling. In actual fact, K-AODV displayed problems in assembling a routing table for the given test suite. This may have been due to misconfiguration; either way, due to time constraints, investigating this issue was beyond the scope of the project. For subsequent tests, the machines were brought within radio range of each other,

2. **Consistency and availability.** Files were created from different node points and their availability is checked from all available nodes. A second user performed the same test and tested that file permission rules are observed. This test was performed with two concurrent users and several files covering the twin permission scenarios given by the use of public and user directories. There were no problems reported with this test.

3. **Observable transparency.** Open files were continuously read as the user moves between nodes, and read delay, or jitter, is observed. Files of varying size were used. The results of this test were predictable. The quality of response time varied in inverse proportion to the distance from the node hosting the actual file, as expected.

4. **Follow attribute.** Test 3 wass repeated with a 'follow' attributes set on the test files. Availability delay in node transit was compared with the results of test 3. The consensus was that multimedia files below a size of roughly 1 MB were observed to suffer no discernible jitter. This is because the transfer time for a file moving to the new node is small enough to generally occur between periodic reads executed by multimedia applications. Larger files than this can suffer significant delay time.

5. **Carry attribute.** The characteristic of 'virtual file location' was tested by using a creator/mobile client and an observer client. The availability of the file was observed to match the requirements' criteria as the client roamed from node to node. This feature was found to be trouble-free.

6. **Concurrency and stability.** In all of the above tests, it is required that deadlock, starvation, unreasonable delay and corruption of data must not be observed. Variations of Test 2 were applied in this testing stage, using multiple simultaneous clients. It was found that unforeseen network circumstances could arise that were not planned for in the initial fault tolerance stage of the design process. In certain cases, involving interrupted radio connection, protocol deadlock could occur. Despite extensive exception

handling throughout the application, several areas were deemed as not sufficient to meet stringent fault-tolerance criteria. This is an issue for future work.

Chapter 6

Conclusions and future work

This dissertation has explored various issues concerning ad hoc wireless networking and the effect it has had on trends in filesystem services development. The potential for applications in the mobile, ad hoc environment continues to grow, yet the gulf between the cutting-edge technology and 'killer' applications to take advantage of it continues to grow. This is partly due to the prevalence of telecoms operators in this commercial space, and partly due to the strict constraints imposed by the technology, such as limited battery power and unpredictable connectivity.

I have presented a system which is intended to be simple, in order to demonstrate how new ideas and applications may be built on a certain layer of organisation. I contend that the principal challenge to widescale adoption of modern wireless technology is a lack of focus—to kickstart a new wave of interest in technology, a catalyst application is needed that bridges the gap between what people know and use and the technology that they are not yet aware of. By offering a familiar, unified filesystem view of the network, I would hope that this would go some way towards bridging that gap.

Future work

There are several areas where the Stirling filesystem could be improved:

- **Readahead for the follow attribute.** A jarring aspect of the filesystem is the lag on large files which is noticeable when a file is set to follow a user around the network. An interesting improvement to the system would be a form of anticipation - given that the user is reading at a certain point in the file, it would make sense to transfer over that section of it that is likely to be read next before any other parts of the file.

- **Directory structure.** At present, only two possible directories are available to the user. There is much scope for extending this to meet standard filesystem capabilities.
- **Fine-grained permissions.** Similarly, file permissions only really exist at the directory level in this prototype. This could be extended to cover individual files and even to implement Unix-like user group permissions.
- **Security.** This is rather lax in the prototype, relying on environment variables for authentication. This could be revamped to include Kerberos-style authentication, and even file encryption while in transit between nodes.
- **Exception handling.** Some protocol exchanges need more stringent fault tolerance, as a small amount of instability has been observed in certain cases that adversely affects the running of server nodes.
- **Replication.** As files become scattered over the network as the system is heavily used, no tracking is made of the inevitable copies of the same file. This feature of the system could be harnessed to implement a form of replication, for example, using datestamps.

Appendix

Third-party components used in this dissertation

The Stirling server node code uses a thread pool implementation by Christopher Arndt, used under the GPL licence from `http://chrisarndt.de/en/software/python/threadpool.html`. The version used here is 1.1, from July 2005. Also submitted to the ActiveState Programmer network at `http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/435883`.

The Stirling XML manager uses a read-write lock module by Fazal Majid, placed in the public domain at `http://www.majid.info/mylos/weblog/2004/11/04-1.html`. The version used here is 1.2, from March 2005.

Bibliography

[AB04] Shyam Antony and Gautam Barua. Lean-DFS: A distributed filesystem for resource starved clients. In *IWDC*, pages 150–155, 2004.

[AVA96] P. Eastham A. Vahdat and T. Anderson. WebFS: A global cache coherent filesystem. Technical report, Dept of EECS, UC Berkeley, 1996.

[BI03] M. Boulkenafed and V. Issarny. AdHocFS: Sharing files in WLANS. In *Proceeding of the 2nd IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, USA, April 2003.

[CDK94] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[CM99] S. Corson and J. Macker. Rfc 2501: Mobile ad hoc networking (manet): Routing protocol performance issues and evaluation considerations. Technical report, 1999.

[CPM] S. Chessa, R. Di Pietro, and P. Maestrini. Dependable and secure data storage in wireless ad hoc networks: An assessment of DS.

[DHA03] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems, ICDCS2003*, 2003.

[DW] J. R. Douceur and R. P. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *Proceedings of the 20th Symposium on Reliable Distributed Systems (SRDS ’01),* New Orleans, LA, USA.

[HH95] P. Honeyman and L. Huston. Communication and consistency in mobile file systems. 1995.

[HLM03] K. Hanna, B. Levine, and R. Manmatha. Mobile distributed information retrieval for highly-partitioned networks. 2003.

[KB] Luke Klein-Berndt. A quick guide to AODV routing. Technical report.

[KS96] Ajay D. Kshemkalyani and Mukesh Singhal. An optimal algorithm for generalized causal message ordering (abstract). In *Symposium on Principles of Distributed Computing*, page 87, 1996.

[Mis] Misc. *Mobile Information Access: Coda and Odyssey*. World Wide Web, `http://www-2.cs.cmu.edu/afs/cs/project/coda-www/ResearchWebPages/`.

[Nie96] Jakob Nielsen. The impending demise of the file system. *IEEE Software*, 13(2):100–101, 1996.

[PBRD03] Charles E. Perkins, Elizabeth M. Belding-Royer, and Samir R. Das. Ietf manet working group aodv draft, http://tools.ietf.org/wg/manet/draft-ietf-manet-aodv/draft-ietf-manet-aodv-13.txt. Technical report, 2003.

[Per01] Charles E. Perkins, editor. *Ad Hoc Networking*. Addison-Wesley, 2001.

[Sat89] M. Satyanarayanan. A survey of distributed file systems. Technical Report CMU-CS-89-116, Pittsburgh, Pennsylvania, 1989.

[Sat96] M. Satyanarayanan. Mobile information access. *IEEE Personal Communications*, 3(1), 1996.

[SKK$^+$90] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[SMG98] C. Skawratananond, N. Mittal, and V. Garg. A lightweight algorithm for causal message ordering in mobile computing systems. Technical report, 1998.

[VR] G. Van Rossum. *Python Programming Language*. World Wide Web, `http://www.python.org/`.

[VVV$^+$03] Efstratios Valavanis, Christopher Ververidis, Michalis Vazirgianis, George C. Polyzos, and Kjetil Nrvg. MobiShare: Sharing context-dependent data and services from mobile sources. In *WI '03: Proceedings of the IEEE/WIC International Conference on Web Intelligence*, page 263, Washington, DC, USA, 2003. IEEE Computer Society.

[YH01] Kinuko Yasuda and Tatsuya Hagino. Ad-hoc filesystem: A novel network filesystem for ad-hoc wireless networks. In *ICN (2)*, pages 177–185, 2001.