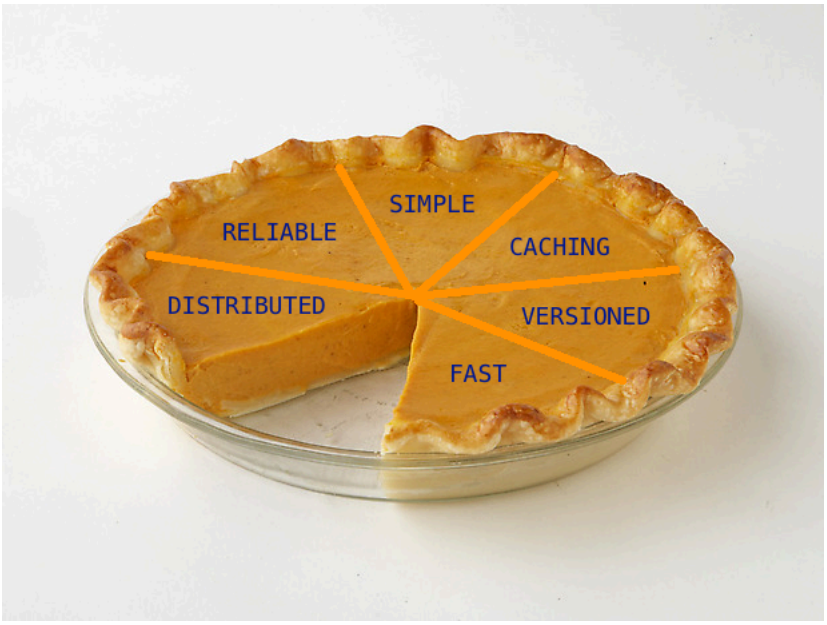
πp

A New Network File Protocol



Anant Narayanan

Supervised by:
Prof. Sape Mullender; Bell Labs, Alcatel Lucent; Antwerpen
Dr. Herbert Bos; Computer Systems Group, Vrije Universiteit; Amsterdam

Amsterdam, August 2010

Master Thesis, Parallel and Distributed Computer Systems

Abstract

We have witnessed an explosion in the dominance of the network in computer systems today. This trend is only poised to grow further in the future, and it is thus important to analyze and perhaps improve the methods by which we utilize the network in order to improve the computing experience.

The UNIX operating system introduced the concept of representing all information as files, and this model has proven to be a very powerful one. The concept has since then been adopted by many subsequent operating systems, most notably by the Plan 9 operating system where the philosophy was extended to the network. The most basic necessity in order for us to be able to interact with files over a network is a protocol that dictates how this occurs. There are several such protocols in use today, and in this thesis, we propose a new, simple network protocol that we think improves on many of them.

πp may be considered a revision of the 9P2000/Styx protocol used by the Plan 9/Inferno operating systems, which was last updated in the year 2000. In the last decade, the networking landscape has changed significantly and brought on us new challenges that we aim to address with this new protocol.

Acknowledgements

Designing and implementing a protocol like πp is definitely no one-man job. A lot of hard work from several people has gone into it, not to mention the input of several brilliant ideas from others. We would like to acknowledge the significant contributions of: Pascal Wolkotte, Noah Evans, Francisco J. Ballesteros, Ron Minnich, Eric Van Hensbergen, Gorka Guardiola, and Priyanka Sharma.

The designers of other network file protocols, in particular 9P2000, are due of gratefulness for giving us so much to learn from and allowing us to stand on their shoulders.

Prof. Sape Mullender deserves special recognition for pulling ideas from everyone together and providing impeccable direction and eye-opening advice towards success of the project. Many thanks are also due to Dr. Herbert Bos for kindly agreeing to supervise the project on behalf of the Vrije Universiteit.

Contents

Chapter 1

Introduction

The internet is widely accepted to be ubiquitous and it is not surprising that it has a role to play in almost every sphere of today's society. However, the underlying technology is about as ancient as computer networking itself and has evolved very little with time. The conventional model of the internet is that of communicating peers, with servers providing content and the clients consuming it. A direct one-on-one connection is always established between a client and server whenever some content is to be consumed owing to the symmetric nature of communication.

However, the number of clients has been increasing by exponential amounts in the last decade and thus, this symmetric communication model places additional requirements on both network and computing resources. This has led to a wide range of research efforts which try to solve scalability problems without replacing any of the technologies that lie at the base of the internet: namely TCP/IP and application level protocols like HTTP. A good example of such an effort is the Akamai project [8], which uses existing systems like DNS to provide effective load balancing and provides a faster experience for the end user.

In addition to the problem of an increasing number of clients, we also observe that the conventional model of servers as hosting content is not always true. Clients are beginning to play a large role in contributing content, which allows for a "read-write" web as opposed to the web of the 90s which was, for the most part, "read-only".

We argue that by completely replacing some components of the technology stack that runs the internet with modern protocols and systems, we can approach such problems from a different angle leading to more efficient content distribution. Specifically, we build a case for modeling the network as a filesystem and propose a new protocol for the same.

1.1 Justification

As we have already noted, the number of people with internet access are growing by leaps and bounds everyday. Not only does the sheer number of users pose a scalability problem, we also observe a general trend in the way these users utilize the internet. Audio and video already consume about half the bandwidth in the internet and this fraction will only increase in the future (projected to be 91% of all traffic by 2014 [14]). As a consequence, traffic will increase dramatically; and the current model of a one-to-one connection between a client and server will simply require more network, storage and communication infrastructure. Can we do something to alleviate the increasing requirements for computing and networking power by looking at the problem from a different angle?

In addition, one may observe that a truly anonymous internet is proving to be elusive and is not sustainable in the long run. We already see more and more restrictions placed on communication by firewalls that now prevent almost all communication that does not use TCP over port 80 (HTTP), especially in corporate networks. These firewall implement ever-deeper packet inspection and anomalous-traffic analysis to prevent the bad guys from hurting the good ones. However, this battle cannot last forever. As an alternative, we propose to replace most Internet communication by authenticated and access-controlled communication to possibly shared objects using a file-access model.

We envision that telecom companies and internet service providers (ISPs) will begin to play an increasingly active role in delivering content to end users. With the emergence of technologies like Voice-over IP (VoIP), telecom companies who traditionally relied on profits made from telephony might instead focus on delivering digital content over the internet to their end users. It is not hard to imagine service-level agreements (SLAs) between content providers and ISPs that will allow them to exploit caching to serve home users with latencies less than their reaction times (this is already happening, debates behind net-neutrality notwithstanding). By involving telecom companies and ISPs in content delivery, we begin to break away from the traditional model of a one-to-one connection and begin to formulate a hierarchical tree-like content distribution system, which is not only a lot more scalable but also profitable, which makes it a feasible model for both ISPs and end users.

In summary, we justify the need for a new network file protocol because:

- The current technology stack powering the internet is inadequate to face the increasing number of users demanding faster and latency-free delivery of digital content.

- As already noted, video traffic comprises of a large majority of internet traffic. However, the delivery of this data is also done over legacy protocols, and we need a new protocol that specifically enables interactive-media transfer (i.e., doesn't need to run over TCP)
- Most of the research in scaling the internet focuses on backwards compatibility, and proposed solutions are "bolted" onto existing infrastructure.
- However, Telecom companies and ISPs are in a position to accept changes at the technology level in order to deliver high-performance internet connections to their end users in a profitable manner. Thus, backwards compatibility may not be as important as is generally perceived to be.

The core of the solution lies in defining a network file protocol that does not build on a symmetric communication model, but instead focuses on a more hierarchical distribution of content which is more reflective of the data flow we observe in the internet today. We believe πp is that protocol.

1.2 Problems Tackled

By modeling the network as a filesystem, and accessing all content as files through a predefined protocol, we are able to make it easier to solve the following problems.

Abstraction. Browsers spend a lot of time fetching files over HTTP. However, a single protocol like HTTP may not work well in scenarios that differ radically in network and computing resources. A uniform file system interface may be able to group together different low-level protocols and caching strategies which are tailored to the local environment (for example, a transfer over the local area network as opposed to overseas communication). The browser should never have to care if the file it wants is local or remote, all the details should be handled by the underlying filesystem.

Authentication. Access control in existing web applications is extremely *ad hoc* and relies on fickle methods like cookies. One of the biggest drawbacks of HTTP is that the specified authentication mechanism is only used by 1% of modern websites. This is because the specification has not evolved since the original HTTP/1.0 specification (methods exists to authenticate both the client and server, but are rarely used). Access control in distributed filesystems is very well understood, and by representing the internet as a giant distributed filesystem we are able to approach the problem in a clean and extensible manner.

Autonomy & Decentralization. One of the defining features of the internet is its decentralized nature, yet we observe a large amount of aggregation of user data towards a few third party services (eg. Google). By specifying a single interface to our filesystem that is easy to understand, we allow for the federation of otherwise autonomous file servers to create a distributed file system that spans the globe.

Backup & Synchronization. As we have seen a trend shift from one computer serving many users to many computers serving one, it is becoming increasingly important to synchronize and backup files to allow universal access independent of location. A distributed filesystem provides basic primitives for maintaining consistency of files across several locations in the network.

Caching. Web caching is severely restricted by the specificity of HTTP, which is only concerned with fetching web pages. Distributed filesystems have caching as one of their important design considerations, and again, these mechanisms are well understood due to a decade or more of research in the area. Caching is a central feature of any scalable solution and we believe distributed filesystems can do a very good job of it while maintaining the required consistency.

Sharing. Information sharing on the existing internet almost always requires the help of a third party service (Flickr for photos, Youtube for Video, BitTorrent for data, etc). The data stored on these third party services are protected with a myriad of usernames and passwords, most of which are incompatible with each other. Direct sharing (eg. Opera Unite [13]), on the other hand becomes simple in a distributed filesystem due to its natural access control mechanisms.

Efficiency. Traditional protocols like TCP and HTTP are well-suited to transporting the types of data they were originally designed for. However, it is only in the last few years we have witnessed an explosion in the exchange of isochronous data. Sure, you can send video over TCP, but is it really the most efficient way? There is a need for a protocol that allows different transport mechanisms to be utilized according to the type of data it is handling.

1.3 Going Forward

We hope you are convinced that representing the internet as a global distributed filesystem is a good idea. We may now move on to the business of designing the protocol that defines how exactly the filesystem operates. However, before we dive into that we must examine existing network file protocols in wide use in order to identify their advantages and disadvantages.

In the next chapter, we will do precisely that. When we establish what the strengths and weaknesses of the prominent network file protocols, we are in a better position to create overarching goals for what a new and improved network file protocol should pursue, which we do in Chapter 3. In Chapter 4, we describe how these goals were tackled and begin to define the protocol itself while defending design decisions. In Chapter 5, we talk about how these ideas were prototyped and how they may be be utilized in certain applications. In Chapter 6, we conclude with thoughts on what we learned and what more can be done in the future.

Chapter 2

Network File Protocols

We now take a look at a few interesting network file protocols so we may identify what their strengths and weaknesses are.

2.1 FTP

FTP is one of the oldest network file protocols (first specified in April 1971 [1], last updated in October 1985 [6]) that is used to copy a file from one host to another over a TCP/IP-based network. FTP is built on a client-server architecture and utilizes separate control and data connections between the client and server applications.

A client makes a connection to the server on TCP port 21. This connection, called the control connection, remains open for the duration of the session, with a second connection, called the data connection, on port 20 opened as required to transfer file data. The control connection is used to send administrative data (i.e., commands, identification, passwords). Commands are sent by the client over the control connection in ASCII. The server responds on the control connection with three digit status codes in ASCII with an optional text message, for example "200 OK" means that the last command was successful.

FTP supports two modes known as "active" and "passive" which specify how the data channel is established. In active mode the client sends its IP address and port number over the control channel and the server initiates the connection. However, in cases where the client is behind a firewall and is unable to accept incoming TCP connections, passive mode is used wherein the client opens both control and data channels to the server. When transferring data over the network, four data representations can be used: ASCII, Binary (contents are sent byte-for-byte), EBCDIC (identical to ASCII except for the character set) and local (any proprietary data format).

Data transfer itself can be done in one of three modes: "stream" mode where data is sent as a continuous stream, "block" mode where FTP breaks the data into several blocks (each block contains a block header, byte count and data field) which is then passed to TCP. The final mode is "compressed" mode in which data is compressed using a single algorithm (usually some type of run-length encoding) and then sent over one of the other two modes.

Let's list some of the advantages of FTP:

- Simple, time-tested specification with a wide range of interoperable (and cross-platform) client and server implementations.
- Clients are not required to accept incoming TCP connections.
- Very low overhead when the goal is to transfer a given file from one place to another.

What's not so great about it?

- No versioning or file metadata support.
- Weak security (prone to 'bounce' attack).
- Direct server-client architecture, no scope for caching.
- Sequential file data transfer, no range support so parallel fetching is not possible.

2.2 Coda

Coda is another network file system that has been around for quite a while. The project started in 1987 at the Caregie Mellon University. It is a descendant of an older version of the Andrew File System and offers a similar feature set with its own improvements. [2]

One of the biggest features of Coda is that is supports disconnected operation. By deploying an intelligent client-side cache, even if a client is disconnected from the network filesystem operations can continue without disruption. The cache reconciles with the remote parties once the connection is restored again. Of course, this raises the question of how conflicts are dealt with. When the changes between two file versions are trivial Coda can perform automatic conflict resolution (very similar to how modern version control systems like git or mercurial deals with conflicting commits) which is completely transparent to the user with the exception of a possibly longer access time for the file. However, this method doesn't always work, when

a file has been changed to a large extent by another party when another client was disconnected, the file is simply marked as 'in-conflict' which must be resolved manually. This is done by having the owner of the file establish which the correct/latest version is.

Coda is also a distributed filesystem, which means files can be stored on more than one server and there is no single point of failure. It also includes full support for ACLs, Kerberos authentication and all the usual filesystem operations. Code exploits the fact that most files are read-only, modifications are infrequent and are usually performed by a single user.

This brings us to the advantages Coda offers:

- Supports disconnected operation for "mobile" computing. Operation is also not disrupted in the event of partial network failures.
- Provides high performance through client side persistent caching.
- Well defined semantics of sharing, even in the presence of network failures.

Coda is not strictly a network file protocol, but rather a whole network filesystem. This includes several components in addition to the protocol used over the network itself. For instance, the local client interacts with the kernel using the usual system calls, and the kernel in turn consults the local cache (called 'Venus') by reading and writing from a character device ('`/dev/cfs0`' - this approach is also the one used by filesystem synthesizers such as FUSE). It is the local cache that finally talks over the network to the remote server and performs the required operation (if the file was present in the local cache it wouldn't even need to do that). For this reason the network protocol is strongly coupled with the system itself and cannot be analyzed usefully on its own. This brings us to why Coda may be unsuitable for deployment as the de-facto 'internet' filesystem:

- The specification as well as implementation are relatively complex (90000 lines of C++ code). All the pieces of the system are tightly coupled and there exists only a single reference implementation. [3]
- The reference implementation only runs on Linux.
- Does not provide explicit support for sharing synthetic filesystems or device files. (Although if caching is disabled for these types of files it could be made to work).
- No explicit support for versioning despite strong file sharing semantics.

2.3 NFS

The "Network File System" was first introduced in 1989 by Sun Microsystems [9]. It has since then undergone several revisions and the latest version (knowns as the "4.1+pNFS" standard [12]) enjoys moderate success and is used in several thousand deployments around the world.

NFS provides the simple and useful ability to "mount" a remote filesystem and have it appear as a local filesystem, on which operations can be performed as they were simply local files. The original NFS versions operated over UDP, but in versions 3 and 4 support for it was dropped from the RFC even though there are test implementations that allow it. The latest version of NFS allows large file access, and performs asynchronous writes on the server-side for improved performance. File metadata is also fully supported and in-fact decoupled from the file data itself. There exists a sophisticated file locking mechanism that prevents collisions but it has been shown that it does not perform very well in conjunction with regular unix file semantics - this arises from NFS clients representing remote files as regular local files [7]. NFSv4.1 adds the Parallel NFS (pNFS) capability, which enables data access parallelism which address the scalability concerns as well as making NFS a truly distributed filesystem.

To summarize NFS's strong points:

- Allows users to operate on remote files as if they were local.
- Provides sophisticated file locking mechanisms and metadata access.
- Decent performance with the usage of pNFS.
- Cross-platform implementations available with industry-backed support.

However, NFS also suffers from the complexity of projects like Coda. The protocol itself is very tightly coupled with the tools for exporting shares and clients mounting them. It also suffers from similar drawbacks to those of Coda:

- Relatively complex specification and implementation. There is more than one interoperable reference implementation, however.
- No explicit support for versioning or serving synthetic files (or devices).
- File sharing is only existent in terms of strong file locking mechanisms, i.e. no concurrent access allowed.

2.4 SMB/CIFS

Server Message Block (SMB, also known as Common Internet File System, CIFS) is a protocol used to provide shared access not only to files, but also to printers, serial ports and even allows for authenticated inter-process communication. This interesting protocol, unfortunately has undergone little scrutiny because of it's proprietary and closed nature. Only parts of the specification were ever published as IETF drafts and even they have since then expired [5]. However, because of the importance of the SMB protocol in interacting with the widespread Microsoft Windows platform, coupled with the heavily modified nature of the SMB implementation present in that platform, the Samba project originated with the aim of reverse engineering and providing a free implementation of a compatible SMB client and server for use with non-Microsoft operating systems. Much of what we know of the original SMB/CIFS protocol comes directly from the Samba project.

SMB works through a pretty standard client-server approach: a client makes specific requests and the server responds accordingly. The SMB protocol is optimized for local subnet usage, to allow better access to generally local resource such as printer but there is no inherent restriction preventing the protocol from being used over wide area networks like the internet. However, it has been noted that latency has a significant impact on the performance of the SMB 1.0 protocol, that it performs more poorly than other protocols like FTP. Microsoft has explained that performance issues come about primarily because SMB 1.0 is a block-level rather than a streaming protocol, that was originally designed for small LANs; it has a block size that is limited to 64K, SMB signing creates an additional overhead and the TCP window size is not optimized for WAN links [4]. Several issues such as these have been addressed in a later version of the protocol, SMB2, which is in wide use by all Windows machines since the release of Vista in 2006.

What's good about SMB2?

- Only 19 control commands as opposed to over a hundred in SMB1 increasing performance and resulting in a relatively simpler specification.
- Supports pipelining, i.e. sending multiple requests before responses to previous requests have been received.
- Allows access to devices (albeit a subset of them, such as printers) in addition to file sharing.

However, we still think it is unsuitable for wide scale internet-size deployment because:

- The protocol itself is proprietary (even though the specification for SMB2 has been published).
- A single reference implementation controlled by Microsoft with only one other open implementation that relies on reverse engineering to "catch-up".
- No explicit support for versioning.
- Only works over reliable transport layers such as NetBIOS or TCP/IP.

2.5 HTTP

The Hypertext Transfer Protocol is undoubtedly the most popular and widely used network file protocol in use today. All web content is primarily delivered to users via this protocol, the latest version of which (1.1) was defined in 1999 [10]. HTTP is a request-response protocol standard for client-server computing. A client submits a request for performing an operation on a remote file (most of the time, the operation is to 'GET' the file) and the server returns the request which may either be a static file or some content generated on-the-fly, or other kinds of response indications. All requests and responses are performed in ASCII, though file contents themselves are (sometimes) binary.

HTTP itself makes no assumptions about intermediaries, and thus in between the client and server there may be several proxies, web caches or gateways. In these cases, the client communicates with the server indirectly and only converses directly with the first intermediary in the chain. Resources to be accessed by HTTP are identified using Uniform Resource Identifiers (URIs), which are another IETF specification and are used by other protocols as well. The original HTTP uses a separate connection to the same server for every file it wanted to interact with, however with HTTP/1.1 clients can reuse the same connection to perform multiple file operations, greatly improving performance.

HTTP requests consist of a request line (ASCII text), followed by request headers (also ASCII), an empty line and an optional message body. Responses follow a similar pattern. For this reason, HTTP sessions can be easily monitored and, in fact, easily read by humans. HTTP is a stateless protocol. The advantage of a stateless protocol is that hosts do not need to retain information about users between requests, making the protocol even more simple.

The advantages HTTP brings with it are:

- Relatively simple protocol specification with an abundant number of interoperable client and server implementations.
- Large scale deployment and wide adoption.
- Allows serving "synthetic" files (i.e. content generated on-the-fly based on the request)
- Allows placement of proxies and caches making the system more distributed

However, we also identified a few drawbacks of the protocol:

- No support for versioning. (Must be noted, however, that the WebDAV standard specifies extensions to HTTP to allow for this and distributed authoring of files).
- No support for file metadata or other popular unix file semantics. GET/PUT/POST/DELETE are very limited operations in terms of what one could do with files.
- Lackluster authentication support. Even though the specification defines a method, 99% of web servers serving content use other methods such as cookies to achieve the same.
- Verbose protocol with a high overhead, especially in the event of having to fetch several small-size files.

2.6 9P2000/Styx

9P is a network protocol developed for the Plan 9 from Bell Labs distributed operating system as the means of connecting the components of a Plan 9 system. 9P was revised for the 4th edition of Plan 9 in 2000, under the name 9P2000 that contained various fundamental improvements. The protocol as used in the Inferno operating system is called Styx, which is essentially a variant of 9P2000. [11]

9P2000 is a relatively simple protocol and only contains 13 message types, each indicating a particular type of operation on a file. The messages themselves are binary in nature, and very compact. 9P, unlike all the protocols we have discussed so far support versioning of files. The protocol follows request/reply semantics like most protocols we have seen so far. The server never sends anything unrequested, however the replies do not have to be immediate or in order. The protocol is designed to run over reliable transport protocols such as TCP/IP. Since files are key objects in Plan 9 (they represent windows, network connections, processes, and almost anything else available in the OS) 9P is well suited to serve synthetic files.

In summary, some of 9P's notable features are:

- Relatively simple specification with a clean separation of protocol from client/server tools. Multiple interoperable implementations in a variety of languages are available.
- Support serving files of almost any type (importantly: synthetic and device), as well as versioning for static files.
- Authentication is supported but is cleanly separated from the protocol description itself (unlike other protocols which unify them both – increasing complexity as well as narrowing their options in the future).

However, there is definitely room for improvement. There are a few things 9P2000 does not excel at:

- No support for pipelining requests, reducing performance. (This is similar to the performance problem faced by SMB1 in networks with high latency)
- No support for rich file metadata (in addition to what the unix `stat` system call offers).
- Only works over reliable transport networks such as TCP/IP.

Chapter 3

Goals

In the previous chapter we looked at a few popular and interesting network file protocols. In this chapter, we shall put what we learned to good use and extract all the useful features of each protocol, discard what we did not like, and formulate a concrete set of goals to guide the design of πp. Each section in this chapter will address a single goal that we felt many of the protocols analyzed earlier did not address to the full extent.

3.1 Simplicity

One fact that stands out from the pros and cons of all the protocols we studied earlier is this: protocols that are relatively easy to understand and implement are the ones that gain the most adoption (protocols not considered simple: Coda, NFS, SMB1). The reason for this is also simple: having more than one interoperable implementation of a protocol is a key factor for its success – having a single reference implementation in a particular programming language controlled by a single party does not make a compelling case for wide use of a protocol. How does one measure the simplicity of a protocol? We think a good measure is: how much time does an average programmer take to understand your protocol enough to write a correct implementation of it?

The trade-off here is the feature set of your protocol. To balance this out, we need a set of very clear goals that our protocol tries to achieve and then reach for them in the simplest possible way. We attempt to lay out these goals in this chapter, but we also start out by saying that we should optimize for 90% of use-cases (backed by data) and keep the protocol simple as opposed to optimizing for all possible use-cases and risk adding complexity.

A side-benefit that usually comes along with keeping things simple is low overhead which translates to better performance. This is not always the case, but history is abundant with examples of computer programs that were optimized for space complexity first, and reduction in time complexity came automatically after.

To summarize this into a somewhat concrete goal: we must build a protocol that is as simple as possible (given the feature set we define here) and provide at-least two interoperable implementations built in different programming languages.

3.2 Flexibility

Several protocols we studied in chapter 2 are not able to serve synthetic files or devices over the network. We think this is one of the important applications of a file protocol and thus we must design our protocol to be as generic and flexible as possible.

Flexibility of a protocol also implies how easy it is to map to an existing on-disk filesystem. In order for adoption of a file protocol to be maximal it must not make any assumptions about how files are stored or presented, and it must be generic enough that implementing it for different operating systems must be feasible. We must try and avoid simply creating a network representation of an existing operating system's filesystem interface, and allow for a wide range of applications: from sharing printers, to interprocess communication, and video caching, to name a few. Let us take two examples of the kind of flexibility we would like our protocol to incorporate:

- File open modes: most protocols restrict themselves to the modes that the UNIX system call `open` has offered for decades (read, write, append, ...). However, in πp, we would like to make these modes as generic as possible so that file servers and clients are free to implement additional modes on a per-application basis – this is especially useful in the case of synthetic files, as we will demonstrate later.

- Client endpoint portability: In this age of virtualization and process migration, it is common for clients to shift between physical network endpoints. Most of the stateful protocols in use today, because they rely on TCP/IP, also rely on the physical address of a client to remain the same throughout a session (HTTP is an exception because it is stateless). In πp, we want to make the concept of a session between a server and client as generic as possible without tying it down to specific network addresses.

3.3 Reliability & Isochrony

When judging the reliability of a protocol we usually consider two aspects: reliability of message delivery itself, and the ability of the protocol to recover from crashed/failed states (or malicious clients/servers).

All the protocols we studied earlier deal with the former by simply relying on a transport protocol that guarantees in-order sequential delivery of data (such as TCP/IP). However, in the interest of flexibility (as we discussed in the earlier point), we must question if such a protocol is the best means of achieving this for *all* applications of a file protocol? We mentioned earlier that by 2014, 91% of all internet traffic will be video, and perhaps it is the case that protocols like TCP/IP are *too* reliable for streaming applications. TCP/IP is rarely used for video, and current video streaming solutions prefer utilizing more realtime protocols such as RTSP.

However, we believe it is possible to represent isochronous data (such as video) as regular files and use a generic file protocol to access them while still achieving the realtime performance required. The key is to not rely on a transport protocol such as TCP/IP and build in our own reliability within the protocol. This allows us to control exactly *how much reliability* we need, depending on the use-case. For video, there is no point in receiving frames after the deadline has passed, yet TCP/IP's retransmit mechanism will ensure they do arrive eventually, at the expense of overhead that can simply be avoided.

This is not to say that TCP/IP is not useful. It is very useful in non-streaming scenarios, where we do care that all the bytes of a file reach their destination, in-order. We just do not require TCP/IP as the only option of a transport protocol, and application designers are free to make intelligent decisions on which underlying protocol is the best for them by making full use of the tools that the protocol provides. We do believe that for particular applications using TCP/IP is the appropriate choice, but is certainly not for all possible use-cases.

As for the latter aspect of protocol reliability, all the protocols we have discussed so far have done a good job of ensuring their correctness and we plan on doing the same. While verifying protocol correctness is more of a theoretical problem and is beyond the scope of this particular project, we will take it into account during system design to ensure we are able to recover from faulty states and deal with the possibility of malicious clients and servers.

3.4 Metadata

A glaring drawback of all protocols (with the exception of NFS via `xattr`) we analyzed earlier was the lack of rich file metadata support. We think the days of associating only the `uid`, `gid`, `mtime` and `atime` with a file are behind us. File metadata must be allowed to grow beyond what the aging `stat` system call provides, and we must do so in a future-compatible extensible manner.

Associating extra metadata beyond the bare minimum of what is needed by the operating system is useful to several applications. Search engines may use this to index data, document authoring systems may use this to store the author's name, music files can store artist information here instead of in the file itself, and so on. The only reasonable way to allow this to happen is to implement a flexible key-value pair system that lets application associate arbitrary keys with data. The linux `xattr` system achieves this to a large extent.

This brings us to another aspect that we found lacking in all the earlier protocols: no version information was associated with files. This is, in our opinion, one of the most important pieces of a file's metadata and deserves special attention. By associating a version number with every (non-dynamic) file, and updating it when a file changes brings with it many advantages:

- The most obvious of these is that of backup: the filesystem inherently becomes a 'version control system' that can help in alleviating problems of data loss and allowing access to archived snapshots.
- It allows us to maintain an 'audit trail' of changes to files: who changed the file and when? This is important when files are owned by groups of users and is modified often.
- Versioning greatly simplifies caching content. When specific versions are associated with files, caches can be fully aware of what they store and can guarantee consistency. The problem is then reduced to knowing what the 'latest' version of a file is.

To this end, we wish to design a protocol that not only allows arbitrary metadata to be associated with files, but also maintain their versions as first class objects.

3.5 Distributed-ness

The internet is inherently a distributed system, yet we see that the services offered on the web as of today are very centralized. People rarely use their own computers to store and share their data, but instead rely on single third-party providers (for e.g. Google Docs for documents and Flickr for photos). We believe we can build the right balance of user data security and portability by building certain primitives into the protocol that allow files to be fundamentally distributed across several nodes.

Protocols like NFS, FTP and SMB are designed specifically as client-server protocols and do no expect any 'middle-men', and cease to operate in such scenarios. HTTP, on the other hand, is stateless and thus, it is much easier to build proxies and caches in between. The challenge that πp faces is to build a stateful protocol that not only enables proxies, caches and gateways to operate in between a client and server, but in fact also help caches make intelligent decisions about what data needs to be cached by providing it with the information needed. As we discussed earlier, one of the ways in which we can address the internet scalability issues that are imminent is by ensuring that data is distributed and cached as much as possible. HTTP allows this to some extent, but πp can leverage years of research that has gone into distributed filesystems and make caching even better.

Thus, instead of designing a protocol to transport data from A to B, we approach the problem from the aspect of X, Y, and Z all wanting data stored on A, and how introducing a cache C in between them can make this more efficient, yet secure enough for each client.

3.6 Performance & Latency

This attribute is usually a given - every protocol wants to be the fastest. The end result we wish for is to provide data to whoever wants it in the least amount of time possible. Every protocol we have discussed so far takes this into account, but somewhere along the way certain design decisions hamper performance: in the case of NFS or Coda it is protocol complexity, and in the case of HTTP it is simply its verbosity and overhead.

We plan on employing as many tricks as we can in order to make the protocol as fast as possible:

- πp should run equally efficiently on all types of networks with varying latency (unlike SMB/9P2000). Pipelining is a must-have and the ability for a client to send out parallel requests without waiting for earlier responses is a first step as this reduces the number of round trips.

- HTTP is a text-based protocol and while it has its advantages (human readable requests/responses) we do not feel it is good enough of a trade-off when compared to the compactness of binary messages.
- We can optimize several operations based on the frequency of their occurrence. For instance, it has been observed that a large majority of the files, when opened, are read from start to finish. By providing the size of a file on open, we can potentially save an extra operation of having to read the metadata of a file to obtain its size so that it may be read in its entirety.

Summary

We wish to build a **fast**, **simple**, **distributed**, **reliable**, **versioned**, **caching** network file protocol, where the definitions of each term are as described in this chapter, and each of which are lacking to some extent in existing protocols.

Chapter 4

Design

We will now proceed to describe the design of πp that addresses the goals laid out in the previous chapter. We will not attempt to make any formal description but instead outline some of the protocol's major features and design decisions (appropriately defended). A low-level, more comprehensive description of the protocol messages themselves along with extensions we have defined so far can be found in the appendix.

4.1 Basics

πp is based on a request-response model, and consists of a client and server exchanging binary messages. These messages are exchanged over a single transport channel initiated by the client which may or may not be reliable. Each message consists of one or more operations (a 'group'). Request operations (usually transmitted from a client to a server, but not always) begin with a 'T', while responses begin with an 'R'. In practice, a message is a discrete packet of data.

4.1.1 Terminology

We will use the following terms throughout the rest of this document:

- **fid**: A 4-byte integer chosen by the client to denote a particular (version of a) file on the server. All operations on the file once the fid has been established is done via use of the number.
- **Operation**: A single operation that modifies state of the client-server connection. Operations are the building blocks of the protocol. Examples of operations are `Tread` and `Rclose`.
- **Message**: A group of one or more operations sent as a discrete packet of data over the underlying transport layer.

4.1.2 Versioning

It is an explicit goal for πp to be a versioned file protocol. Thus, we define the following properties:

- All files (with the exception of 'special' files like synthetic or device) are versioned.
- Versions are immutable and are committed upon file close.
- Updates to a file start with a particular immutable version and produce a new one.
- Versions are identified by a signed 64-bit timestamp that represents the number of nanoseconds elapsed since the start of the third millennium at UTC (negative numbers represent timestamps from the second millennium).

Updates to a file can occur in two ways:

- **Public**: When the file is first opened, an *unnamed archival* copy is created. The original version can then be modified by the client that opened it, *as well as* other clients who choose to open it. Changes made by any client can also be observed by other clients. When the last (updating) client closes the file, the version of the file is updated with the current timestamp, while the *unnamed archival* copy is stored with the timestamp of the earlier version.
- **Private**: When a file is opened in private mode, the current version is cloned into an unnamed and invisible new version that can only be read and written to by the client that opened it. When the file is closed, this cloned version will be timestamped (and thus become *current* as it is the latest) while the previous version is archived.

We would like to note that these are suggestions on how behavior with regards to versioning files must occur. The protocol only requires that each file possesses a version history associated with it, as well as a way to query the underlying filesystem for any version, and in particular, retrieving the latest (*current*) version of a given file. They may be implemented in ways other than what has been suggested, but must adhere to the properties of file versions that was listed.

We have also defined a leasing extension to the protocol that enables caches to keep track of which version of a file is *current* using a few more primitive operation types. The exact behavior of the operations are beyond the scope of this project (and has, in fact, been dealt with in a separate thesis), but we do include a brief description of the involved operations in the appendix for completeness.

4.1.3 Pipelining

A message contains all the data necessary for either the client or server to parse and execute the operations contained within. Thus, each message is self-containing and may not depend on any other messages. However, within a given message, the order of operations is very important as they may depend on results of previous operations. Whenever a server receives a given message, it executes each operation contained within *in-order*. If any operation does not execute successfully, the server halts execution of any further operations and returns an error (denoted by an `Rerror`) instead of the response it would have sent it would have sent if the operation had succeeded. The response message sent by the server contain individual operation responses in the same order as the corresponding requests were received.

Let's clarify this with an example. Suppose a client had sent a request to read file 'a', and also write to a file 'b' if the read succeeded the message would look like this:

```
-> Topen(a) Tread() Topen(b) Twrite()
```

If the server succeeded in performing all the operations the response message would look like:

```
<- Ropen() Rread() Ropen() Rread()
```

However, had the first read operation failed, the response would be:

```
<- Ropen() Rerror()
```

The brackets in the messages denote the arguments to each operation, which, since we haven't defined them yet, is simply for convenience sake.

Each message is identified by a 4-byte integer called a 'tag'. A client may send out multiple messages (with different tags) to the server as long as each message is self-contained and they do not have any dependencies on each other (if the client wishes to perform operations dependent on other operations, it must include them in the same message). All outstanding messages must have different tags to differentiate them (and classify the response messages, as they may not be received in-order – note this – only responses to operations within the same message are guaranteed to be in the order in which the requests were received, as the underlying transport is unreliable the messages themselves may be sent and received in any order). However, once a response to a message has been received, the client may choose to reuse the tag that was used for it.

In this manner, πp allows pipelining of operations, as well as allowing multiple outstanding messages. We think this design gives clients maximum flexibility in terms of how they wish to optimize requests to the server. For instance, on high latency networks, the client will try and group as many operations as possible into a single message.

4.1.4 Message Layout

Each message is prefixed with the total length of the message as a 4-byte integer (including the 4 bytes for encoding the length itself), followed by a session ID (described in the next section), the 4-byte 'tag' and finally, a 2-byte integer describing the number of operations included. Each operation has a fixed set of arguments, but since arguments themselves may be variable-length strings or data, operations in their entirety are not of fixed-length but determinate. Parsers should have no trouble decoding operations once it know the full message length and the number of operations within (the latter is not even strictly required and is only provided for convenience).

There are only 5 data types defined in the protocol: u16int, u32int, and u64int are integers in network-endian byte order of their respective sizes; string is a regular UTF-8 encoded string with its length (in bytes) prefixed as a u32int; data is an arbitrary set of bytes with its length also prefixed as a u32int. Given this, the generic form of a πp message is:

`{hdr:data}{len:u32int}{id:u32int}{tag:u32int}K{O1,O2...On}`

where '`hdr`' is the transport level header information, '`len`' is the total length of the message, '`id`' is the session ID, '`tag`' is the unique identifier for the message, and '`K{O1,O2...On}`' is the optionally encrypted concatenation of `n` operations. The operations themselves are formatted as a 4-byte integer specifying the operation itself, followed by arguments specific to the operation – which can only be a combination of any of the 5 data types we defined. The operation codes and their arguments are formally specified in the appendix, but we will also discuss a few important ones in this chapter.

4.2 Sessions

πp is a stateful protocol, and conventionally servers maintain state for a particular client based on the incoming network address it receives requests from the client on. However, one of the goals we mentioned earlier was to allow for portability of clients. For this reason, we introduce an explicit message exchange to establish a session ID to identify client-server state. All subsequent messages that are sent by the client or server must include the corresponding session ID so the other end may correctly classify that message. Decoupling state from network endpoints has a few advantages:

- It allows clients to change network locations but simply pick up where they left off at their new location.
- It allows multiple clients to talk to a single server over the same network interface but on different, independent sessions.

- It allows two or more co-operating processes present at different locations to share a single session with a server.
- It allows two or more different (co-operating) clients to establish independent sessions with the same server over the same transport channel.

4.2.1 Session ID Exchange

Session establishment is the first step, and thus in a πp connection, once a client has established a transport channel to the server, the first operation it sends is a '`Tsession`':

{`csid:u32int`}{`afid:u32int`}{`msize:u32int`}{`options:string`}

Here, '`csid`' is a session ID that the client wishes to identify itself with. All subsequent messages sent from the server to client will be prefixed with it. '`afid`' is a fid that the client wishes to associate with an 'authentication file' (the purpose of which we shall describe shortly). '`msize`' is the maximum size of a message that the client is able to process (this may be determined in conjunction with the underlying transport protocol used). '`options`' is a string describing any protocol extensions the client wishes to use (only two such extensions have been defined so far and have been described in the appendix, but this is completely extensible). The server responds with an `Rsession` operation if the `Tsession` operation it just received is acceptable to it (otherwise an `Rerror` is sent back):

{`ssid:u32int`}{`afid:u32int`}{`msize:u32int`}{`options:string`}

The meanings of each field are very similar to those found in the `Tsession` operation. '`ssid`' is a session ID that the server wishes to identify itself with. All subsequent messages from the client to the server must be prefixed with this number. The server will set '`afid`' to the same value as the one it received if it is able to provide an authentication file. If the server does not require or support authentication/encryption, this value will be set to a special value (`NOFID`, currently defined as `~0`). '`msize`' and '`options`' are the maximum message size and protocol extensions that are acceptable to the server. The values sent back by the server are the final values that both parties must honor from this point forward. The server must always send back a value of '`msize`' less than or equal to the one it received, and similarly, a subset of protocol extensions that the client requested.

It must be noted that the client is free to append more operations after a `Tsession` in the first message it sends, within reasonable limits of '`msize`'. However, until it receives an `Rsession` back, it must not send any more

messages, simply because it does not yet know what session ID to prefix them with. The message that contains the initial `Tsession` is prefixed with a special value `NOSID` (currently defined to be `~0`) and only 1 such message is allowed to be on the channel between a client and server at any given time.

Since all subsequent messages between the client and server after the ID exchange has occurred is prefixed with the appropriate session IDs, state is now decoupled from network endpoints and we are able to enable the four use-cases laid out in the earlier section. Whenever a client or server receives a message, the prefixed ID is what is looked at first in order to decide which process gets to handle the message, or where to pull out state for that connection from. Migrating processes take with them knowledge of session IDs and so the session is not 'broken' simply by moving between two network endpoints – both for clients and servers.

4.2.2 Authentication & Encryption

We will now discuss the use of the '`afid`' that was used in the session ID exchange. The concept of an authentication fid is a very versatile one and was present in the original 9P2000 protocol as well. The basic idea is to avoid including any specific authentication mechanism in the protocol itself, while taking advantage of the fact that almost any authentication method can be represented as reads and writes to a particular file.

The '`afid`' represents an open file handle to a special file that is controlled by the authentication provider. If the client receives an '`afid`' in an `Rsession`, it must then follow-up with a series of read and write operations on that fid that execute the chosen authentication protocol. For instance, let us take the example of the client wanting to write a username-password pair to authenticate and then perform a public key exchange to encrypt subsequent messages with. Assuming the '`afid`' chosen by the client was `1` (and was accepted by the server in the `Rsession`) the following message exchange would occur:

```
-> Twrite(1, 'bob:passwd') Tread(1,2)
   Twrite(1, bobs_pub_key[m]) Tread(1,n)

<- Rwrite(10) Rread('OK') Rwrite(m) Rread(srv_pub_key[n])
```

The result is that the client has now authenticated on behalf of user 'bob' and a public key exchange has occurred.

What we present is simply an example of how an authentication and key exchange using an '`afid`' may occur. The process of reading and writing to a fid is generic enough to represent almost any means of authentication, and

the client-server may take as many roundtrips as needed in order to finish executing the chosen protocol. In fact, if the authentication mechanism has not been agreed on by the server and client beforehand, the first pair of reads and writes may be used only to determine which common mechanism is acceptable to both the client and server.

If authentication fails for any reason, it is the duty of the mechanism itself to notify all parties involved (in our example, reading from the '`afid`' after writing a username-password pair returns 'OK' to indicate success). Any subsequent operations sent to the server after authentication fails will simply return `Rerror`s.

We must emphasize how important authentication and encryption are, especially given the portability of session IDs. If a client and server choose not to encrypt their messages, it is trivial for a malicious party to sniff packets from the client after a session ID exchange has occurred and use it to send messages to the server pretending to be the client. If a server chooses to use only the session ID as the means of identifying state of a client, we strongly recommend it to enforce authentication and encryption requirements on all incoming connections. If a client sends messages containing regular operations without operating on the '`afid`' first, the server must simply return an `Rerror` indicating that the client has not authenticated yet. Similarly, if it receives unencrypted packets and policy dictates otherwise, `Rerror`s may be generated.

4.2.3 Proxying & Caching

As we have discussed in the previous chapters, the ability to proxy requests and cache responses is critical to the scalability of any large distributed filesystem. It is an explicit goal for πp to allow caches and proxies to act on behalf of clients.

We introduce our next operation type: `Tattach` and `Rattach`. A `Tattach` serves two primary purposes:

- To introduce and authenticate a particular user to the server
- To indicate a client's interest in a particular file tree that is offered by the server

We already did perform authentication during the session exchange, but we make the clear distinction between the user that initiated the *connection* and the user who wishes to access a particular *file tree*.
Here's what a `Tattach` looks like:

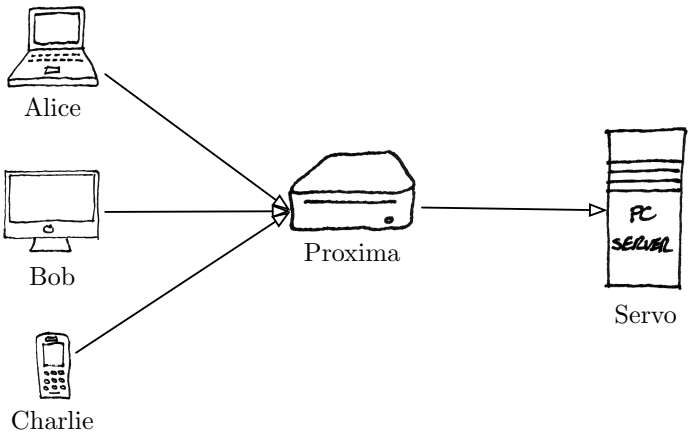
{`fid:u32int`}{`afid:u32int`}{`uname:string`}{`aname:string`}

'`fid`' indicates the fid that the client would like to associate with the root of the file tree it wishes to access, '`afid`' is the fid it would like to associate with the authentication file where it will prove that it is user '`uname`' (or is acting on behalf of), and '`aname`' is the name of the file tree that the client wishes to attach to. The server then responds with an `Rattach`:

{`afid:u32int`}

'`afid`' is set to the same value as was received in the `Tattach` is the server is willing to execute an authentication protocol to prove that the client is, or that the client is acting on behalf of, '`uname`'. The server must set '`afid`' to `NOFID` if it does not support or require authentication. As mentioned earlier, we do not believe we must specify the means by which this is done, and it may be chosen arbitrarily according to the needs of the application by both the client and server. This '`afid`' behaves exactly like the authentication fid used in the session exchange and the server must respond to read and write requests on it.

Let us take an example:



Alice, Bob, and Charlie all initiate session exchanges with Proxima and each of those links are encrypted using their individual public keys. Proxima, may have, at an earlier stage (or upon an initial session exchange from either Alice, Bob or Charlie) initiated its own session exchange with Servo. The link between Proxima and Servo is encrypted using only their public keys.

When Charlie needs a specific file on Servo, he will initiate a `Tattach` with Proxima, prove that he is indeed Charlie and additionally provide a signature over the same '`afid`' that he authorizes to Proxima to access files

on behalf of him. Subsequently, Proxima will initiate a `Tattach` with Servo and provide this signature over that '`afid`'. If Servo is convinced that Proxima is indeed acting on behalf of Charlie (an independent channel to verify this may or may not be needed) it will allow access to files under the fid Proxima chose to associate Charlie's root file share with. If Alice or Bob also need files on Servo, the same procedure will be repeated, Proxima will send one `Tattach` per user it wishes to act on behalf of.

Since Proxima is a shared trusted party between the three users, it can also optimize and cache responses from Servo. Not only can it cache files specific to, say, Charlie to speed up his requests, but if a common file that all three clients need is found in its cache, Proxima can serve requests directly to each of them without ever having to talk to Servo. πp also provides additional versioning information with respect to files (as was discussed earlier) that can make Proxima's job as a cache easier.

4.2.4 File & Server Identifiers

In order to ease the job of caches even further, πp expects files and server to posses unique identifiers for themselves. This information can be queried with the `Tread` request for file *metadata*, and is discussed in the following chapter.

However we must note here that every server possesses a unique 64-bit identifier called an `sref`. Every file on a server also possesses a unique 64-bit identifier called an `fref`. An `fref` refers to a particular file, and not to any specific version of the file, and it must remain the same across renaming. Together, the `sref` and `fref` are able to globally and uniquely identify any file.

It is left to the server to implement the creation and maintenance of these identifiers, though they may find it easy to create a file identifier upon creation of the file itself and persist it throughout its life-cycle. The creation and maintenance of server identifiers is also left unspecified, though it is certainly viable to create such identifiers based on the MAC address of the network interface the server will listen on. The availability of this information, in addition to file versioning information discussed earlier, will greatly simplify the operation and implementation of caches.

4.2.5 Closing a Session

A session may only be closed by a client with the use of an `Tclunk` operation. The server may not initiate the close of a session, if a server stops responding to requests by the client, it is an error. A `Tclunk` operation simply contains the server session ID of the session the client wishes to terminate:

{ssid:u32int}

It is an error for the client to include any operations after a `Tclunk`, as they will not be processed by the server. The server responds with an `Rclunk`, this operation has no arguments. The server may choose to close the transport channel after it has ensured that the client has received the message containing the `Rclunk` - which must always be the last operation in that message.

Sometimes it is also necessary for a client to withdraw a message it had already sent to the server (for e.g. if a read request is no longer valid because the deadline for the data has already passed - this is common while streaming video). This is made possible by use of the `Tflush` message:

{tag:u32int}

which indicates to the server that the client does not wish the message with '`tag`' to be processed anymore. If the server has already processed the entire message at the time of receipt of the flush request, it must return an `Rerror` in response to the `Tflush` - and send back the results of the message that was requested to be flushed as normal. If the message has not been processed yet or has been processed partially (in which case the server must stop after the operation it is currently executing has finished) and must send back an `Rflush` (which contains no arguments).

If the message that was requested to be flushed had been partially processed, the server must also send back the results of the operations that were processed. The server never interrupts the execution of a single operation itself, and may only stop in-between operations of a message. Individual operations themselves can only either succeed or fail, never partially so.

4.3 File Operations

Once a session has been initiated, authenticated and an attach performed to a given file tree (again, with the proper authentication) the client may begin performing operations on the files & directories exported. We will now take a look at all the πp operations that let the client deal with remote files.

4.3.1 Open & Close

The `Topen` operation is one of the fundamental tools which clients use to manipulate fids. The operation performs all or some of the following (in the order they will be executed on the server depending on arguments):

- **Clone**: This associates a new fid with a file that is pointed to by a given fid. It requires two arguments: an fid which already points to something and a new, unused fid.
- **Walk**: A walk descends into a directory and associates one of its children with another fid. It requires two arguments: an fid to walk from (this must be a directory, walks on fids pointing to regular files are undefined), and a path which represents the child to look for under the directory. The provided fid is then associated with the requested child after the operation succeeds. Hence, if a reference to the parent is to be retained, a the fid must be cloned by providing a new, unused fid.
- **Open**: This behaves like a regular file open. It requires two arguments: an fid and a mode to open the file in.

The exact operation that is performed when a `Topen` operation is requested depends on the arguments that were passed along:

`{fid:u32int}{nfid:u32int}{path:string}{mode:string}`

If only '`fid`' and '`nfid`' were provided, a clone is performed. If '`fid`', and a '`path`' are provided, a walk is also executed. Finally, if '`fid`' and '`mode`' are present, the operation concludes with a regular open. Once a fid has been opened, it's state changes on the server such that it may no longer be walked. Similarly, a file cannot be opened without being walked to first. If the operation succeeded, the server responds with an `Ropen`:

`{ftype:u32int}{version:u64int}`

where '`ftype`' is the type (such as regular, directory, append-only, synthetic etc.) and '`version`' is the timestamp of the file that was just opened, cloned, or walked to.

It must be noted that the '`mode`' argument is a string. This string is of arbitrary length and is free to be interpreted by the server with the exception of the first three bytes, each of which denote 'r', 'w' and 'a', and a fourth acceptable value of '-'. The server must support the following values of '`mode`':

Value of `mode`	What it means
`r--`	Read Only
`-w-`	Write Only
`rw-`	Read & Write
`--a`	Append Only

Any bytes subsequent to the first three are free to be interpreted by the server and are specific to the application. The client must know about the extra modes that the server implements beforehand, although file metadata may be used by the server to communicate what extra modes are available. We will discuss file metadata in a following section.

Closing a file is a simple affair. A `Tclose` contains:

{`fid:u32int`}{`commit:u16int`}

where '`fid`' denotes the file to close, and '`commit`' describes if the changes made to the file (if any) since it was opened must be written - thus resulting in the creation of a new version. If the close succeeded, the server responds with an `Rclose`:

{`version:u64int`}

where '`version`' is the timestamp, and therefore the file's current version that was created as a result of the close. If the close did not succeed, for example, if the changes to the file were already committed and the client requested them not be, or if a new version could not be created due a conflict (another user had opened the file and closed it in the meantime); the server responds with an `Rerror` instead. When a file is closed, the fid is forgotten.

4.3.2 Read & Write

The contents of a file (or directory) as well as their metadata are both read using the `Tread` operation:

{`fid:u32int`}{`offset:u64int`}{`count:u32int`}{`attrs:string`}

The '`attrs`' argument determines if the read has been requested for the contents or metadata of the file pointed to by '`fid`'. If '`attrs`' is not present (i.e. an empty string of zero length) then the read is for the contents of the file, beginning at byte '`offset`' for a length of '`count`' bytes. We will discuss what happens if '`attrs`' is non-empty or if the contents of a directory is read in the section on file metadata. The response for the request to read the contents of a file is either an `Rerror` in the case that the file could not be read (for instance, if the permissions did not allow it), or an `Rread` of the form:

{`dat:data`}

The length of '`dat`' is always equal to or lesser than '`count`'.

Writing to a file is similarly achieved using the `Twrite` operation:

{`fid:u32int`}{`offset:u64int`}{`dat:data`}{`attrs:string`}

Just like in `Tread`, if '`attrs`' is set the operation implies a modification of the file's metadata. If the contents of '`fid`' are to be written on the other hand, '`dat`' is written to the file at '`offset`'. The server responds with an `Rwrite` in case the write succeeded:

{`count:u32int`}

where '`count`' is the number of bytes written, which obviously can never be greater than the length of '`dat`' of the corresponding `Twrite`.

A `Twrite` operation to the contents of a directory is undefined, though it's metadata may be modified. A directory's contents can only be indirectly modified with the use of create and remove, which we discuss next.

4.3.3 Create & Remove

A file is created with use of the `Tcreate` operation:

{`fid:u32int`}{`name:string`}{`perm:u32int`}
{`mode:string`}{`ftype:u32int`}

This indicates a request by the client to create a file of type '`ftype`' under '`fid`' (which must point to a directory as creating a file inside another is not defined) named '`name`'. The file (or directory, as indicated by '`ftype`') will be initialized to '`mode`' which is interpreted in the same way as in `Topen`. The permissions of the newly created file are set to '`perm`' which is also free to be interpreted by the server, but we recommend a convention that is presented in a following section.

If the file was successfully created, the server responds with an `Rcreate`:

{`version:u64int`}

where '`version`' is the timestamp at which the file was actually created on the server. Removing a file only requires it's fid:

{`fid:u32int`}

the successful response to which is an `Rremove` operation which consists of no arguments. Attempting to remove a non-empty directory is an error.

4.3.4 File Metadata

In the section on reading and writing to files, we mentioned that those operations may also be used to read and modify the metadata associated with a file. As we discussed in the chapter on goals, we wish for metadata associated with a file to be a flexible key-value pair based system. Applications are free to associate as many key-value pairs with the metadata of a file.

In order to read a file's metadata, the client must set the '`attrs`' argument in an `Tread` operation to the list of keys - one per line (hence, newline separated). Two special characters, however, are reserved to mean specific sets of keys:

- '$\star$' implies that the client wishes to read *all* the metadata associated with the file.
- '`#`' implies that the client wishes to read the default set of metadata associated with the file. This default set of keys always have values, and *cannot* be modified by use of the `Twrite` operation (but they may be indirectly changed with the use of other operations).

The default set of metadata keys are listed below:

Key	Value
`sref:u64int`	Unique server identifier
`fref:u64int`	Unique file identifier
`ftype:u32int`	File type
`perm:u32int`	File permissions
`name:string`	File name
`length:u64int`	File length
`atime:u64int`	Last access time

All of these values are self-explanatory, or have been discussed already. The response to a metadata `Tread` operation is an `Rread` whose '`dat`' will be a set of newline separated key-value pairs, appropriately quoted. If a particular key that was requested by the client was not found in the file metadata, the key-value pair will simply be omitted in the response.

If the contents of a directory is read, the '`dat`' attribute of the response will start with a 32-bit integer specifying the number of children the directory contains, followed by that many records each of which are simply the default metadata values concatenated one after the other. Each record, therefore, will be (44 bytes + length of the file name) long.

The '`attrs`' attribute for a `Twrite` should be set to a list of key-value pairs, one per line, appropriately quoted; just like the response to a metadata `Tread`. If a key-value pair is included and the key cannot be found in the file's existing metadata the key is created and set to the provided value.

4.3.5 Permissions

We have defined the '`perm`' argument so far as a 32-bit integer and said that the server is free to interpret it as needed by the application. It is our intention to leave the specification open to interpretation and not include any particular access control mechanism in the definition of the protocol itself.

An easy, backwards compatible way is to interpret the permission field is to treat it as a regular UNIX file permission bitmap. However, for a more sophisticated access control list based system, this may be insufficient. We propose that such servers export an extra file tree, that by convention we can call `perm`. This file tree may be accessed as any other file tree, through use of the '`aname`' argument of the `Tattach` operation. Once this permission file tree has been attached to, the client may inspect its contents to retrieve what we call *permission objects*.

Each object, represented as a single file, represents a group of users with a certain set of access control bits. An example of such an object would be one that states: "Alice, Bob and Charlie have read-only access" and another such object may state: "Bob has read-write access" to the files that choose to use this permission object as its `perm` identifier. The name of the file may be a hex version of a 32-bit integer that represent this particular object and serves as a reference. This value may be used as the '`perm`' argument in the `Tcreate` call, as well as the value of a file metadata's '`perm`' key. If the client wishes to create a new group of users with a fixed set of access control bits, it must simply create a new file with a unique name in the file tree.

Once again, we omit the description of the permission objects themselves, but we believe that this is a flexible way of representing an access control list. We may revisit this subject at a later date in order to refine the idea further.

Chapter 5

Implementation

As we stated in the chapter on goals earlier, it is our intent to produce at least two interoperable implementations of the protocol in different programming languages. In this chapter we will describe the libraries that were written for the C and Go programming languages. We will then proceed to discuss three example applications of the protocol, each of which use a particular feature. Finally, we conclude with an evaluation of the implementations and present notes on how they may be improved.

5.1 Code Generator

During development of the protocol, as is expected, changes to operation types and messages were very frequent. An implementation of a protocol primarily consists of an 'API' to pack/send as well as unpack/receive binary messages into data structures offered by the language. We decided to write a library to do this parsing, but because the protocol specification itself was subject to rapid change writing the libraries by hand would mean that we would have to go back and continuously keep them updated with the specification.

To counter this problem, we instead wrote a program that would generate libraries implementing the message parsing in both the C and Go programming languages by taking a JSON file representing the operation types and their arguments. Whenever we decided to tweak the protocol specification, we only had to change the JSON input file and re-run the code generator to get new libraries. The code generator itself was written in the Go programming language.

5.1.1 C

The C implementation of the parsing code isn't a regular ANSI compliant C library, but was built to be run by the Plan 9 C compiler set. However, the library may also be used on most modern UNIX-based systems (this includes Linux and Mac OS X) by use of the plan9port suite (a port of Plan 9 userspace tools to UNIX).

The basics of the C library lie in 4 functions that are used to convert a block of memory into a group of operations, and then to extract individual operations from that group; and vice-versa. A single operation is denoted by a union $\pi call$. The function signatures are:

```
void B2G(Group*, Block*);
int  G2C(Group*, pcall*);
void C2G(Group*, pcall*);
void G2B(Group*, Block*);
```

A `Block` is simply a chunk of memory with read and write pointer locations (is already used extensively in Plan 9). Typical usage of the API would be to copy a message from the transport layer, perform decryption if necessary, and place the bytes in a Block. The developer then calls `B2G` to create a `Group` from the message and then repeatedly call `G2C` to extract individual operations from the `Group` until than function returns `0`. If a message is to be sent, the developer uses the methods `C2G` and then `G2C` in the reverse direction.

5.1.2 Go

The Go library presents a few more features than the C version (which is pretty bare-bones). Just like the C version, we make low-level parsing routines available:

```
type Packet struct {
    Id   uint32
    Tag  uint32
    Rmsg uint32
    Roff uint32
    Wmsg uint32

    Rbuf *io.Reader
    Wbuf *bytes.Buffer
}

func (pkt *Packet) Put(op interface{}) (err os.Error)
func (pkt *Packet) Get() (op interface{}, err os.Error)
```

In addition to providing the ability to adding and retrieving individual operations from a `Packet` (which plays the role of both a `Block` and `Group`

in the C version), the Go library also provides transport-layer routines to send and receive packets over TCP or UDP:

```
type Connection struct {
    Type          string
    Address       net.Addr
    StreamHandle  *net.TCPConn
    PacketHandle  *net.UDPConn
}

func (conn *Connection) RecvPacket(pkt *Packet) (err os.Error)
func (conn *Connection) SendPacket(pkt *Packet) (err os.Error)
```

That's not all. The Go library also makes writing πp servers easier by providing a convenient interface:

```
type Session struct {
    // list of sessions
    Next *Session

    // session IDs
    Ssid uint32
    Csid uint32

    // use at will
    Aux interface{}
}

type Operations interface {
    Session(sess *Session, arg *p.Tsession) (ret p.Rsession, err os.Error)
    Attach(sess *Session, arg *p.Tattach)   (ret p.Rattach, err os.Error)
    Flush(sess *Session, arg *p.Tflush)     (ret p.Rflush, err os.Error)
    Open(sess *Session, arg *p.Topen)       (ret p.Ropen, err os.Error)
    Create(sess *Session, arg *p.Tcreate)   (ret p.Rcreate, err os.Error)
    Read(sess *Session, arg *p.Tread)       (ret p.Rread, err os.Error)
    Write(sess *Session, arg *p.Twrite)     (ret p.Rwrite, err os.Error)
    Remove(sess *Session, arg *p.Tremove)   (ret p.Rremove, err os.Error)
    Close(sess *Session, arg *p.Tclose)     (ret p.Rclose, err os.Error)
    Clunk(sess *Session, arg *p.Tclunk)     (ret p.Rclunk, err os.Error)
}

func New(ops Operations, addr string) (*Srv, os.Error)
```

The developer simply implements the 10 functions (and uses the `Aux` property of the `Session` object to store state across transactions), passes a pointer of the object to the `New` method to run a πp server.

5.2 Applications

The most apparent practical application of πp is to run a file server for several clients, with perhaps a cache in between. However, in this section we will take a look at some non-obvious applications of πp. Note that we did not actually implement all of these ideas, but are rather presented in order to highlight a particular feature of the protocol that helps in building the application being discussed.

5.2.1 RPC

Synthetic file systems are a very flexible way of implementing remote procedure calls in a language-agnostic manner. πp may be used to implement such a synthetic file system to provide remote access to a set of computations. In fact, the provided service need not even be remote, the Plan 9 operating system exports a lot of 'local' functionality through a file-based interface.

However, in such file-based interfaces, we often encounter the need to not only transmit the data required for the computation, but also control messages to dictate what operations to perform on the data. For example, in the `/net` interface offered in Plan 9, there are two files for each connection: `data` and `ctl`. When we need to send a packet, we write the destination address in the `ctl` file and the actual data to be sent in the `data` file. πp can simplify this to be a one-step operation with use of file metadata. Arbitrary file metadata in combination with a dynamic files allows us to eliminate the need for two separate channels of communication (like `ctl` and `data`) and perform the entire operation in a single step.

5.2.2 Wikifs

The HTTP protocol has explicit support to present a single 'resource' in multiple formats. We encounter this need in many applications, where the data in a file is essentially the same but may need to be presented in different formats depending on the program that needs it. A common example, is a wiki page which essentially has two forms of representation: the actual content of the file in plain ('wikified') text and the HTML version that is rendered by web browsers.

The flexibility of arbitrary open modes in πp can be used to represent the different forms of the same file. For example, if a text editor requested to read a wiki file the `Topen` request could contain the mode '`r--t`'. If a browser would need to read the file instead the mode could be '`r--h`'. The file server interprets the last byte of the mode as text ('t') or html ('h') and returns the appropriate form of data on subsequent `Tread` requests. This flexibility in open modes can be extended to any application in which files represent essentially the same data, just represented in different formats:

```
$ mount -t piep user:password@en.wikipedia.org /mnt/wiki
$ vim /mnt/wiki/Vrije_Universiteit
# read as plain text
$ firefox /mnt/wiki/Vrije_Universiteit
# read as HTML
```

5.2.3 Video Server

Since πp does not enforce the requirement of running it over a reliable transport layer, we can use this to our advantage to serve streaming content. For example, if we use UDP, we have no way of knowing if a particular message (containing, say, a `Tread`) ever reached the server, or if the response was sent and lost in transit.

This is not necessarily a bad thing. For an application such as a video stream server, where all operations are idempotent (since we only ever read video files and never write to them) we actually do not care if a particular request reached or if a response was lost if that frame has already passed the deadline. By carefully constructing a set of timers that dictate which frames of the video need to be delivered at which times, it is possible to only selectively retransmit important messages that affect the realtime deadlines. This is a lot more work for the client, since the developer will essentially be implementing a subset of TCP, however it ensures maximum control and utilization of bandwidth. As we mentioned earlier, we do not want to be *too reliable* in cases like this.

On a related note, sometimes it may be required to run πp over UDP, yet expect reliability because the file operations involved are not idempotent. For these cases, we have built a set of protocol primitives that help in ensuring reliable operation and have been described in the appendix.

5.3 Evaluation

We present the number of lines of code that were written for the C and Go implementations of the protocol:

Component	**Lines of Code**
Generator	758
JSON Description	126
C Parser	794
C Server Helper	92
Go Parser	748
Go Server Helper	253

Using the library generated, we implemented a server in Go called '*exportfs*', which takes any regular directory and provides its contents over a πp connection. We also implemented two corresponding clients 'πget' and '$\pi getmul$' that fetches single or multiple files from a server. All experiments were performed over TCP.

Our first experiment was with downloading a single file of size 600MB. We repeated this 10 times for each protocol tested, to obtain average download times:

Protocol	**Time**
πp	46.970s
FTP	47.195s
HTTP	51.464s
NFS	44. 945s

NFS was the winner is this case, but πp was not far behind. We ran a similar experiment, this time downloading 600 files of 1MB each:

Protocol	**Time**
πp	32.432s
FTP	1m18.619s
HTTP	1m26.156s
NFS	44. 945s

Here, we start to see the πp leading. As expected, HTTP performed the poorest because of having to re-establish a new connection for every file (we used `wget`) with FTP suffering similar times. NFS shows pretty good performance, only behind πp by less than 10 seconds.

We suspect that πp will perform better than other protocols in cases such as creating a large number of files simultaneously or streaming video content. Unfortunately, due to lack of time we could not implement servers and clients to test these use cases and the actual numbers are yet to be determined. We do look forward to evaluating these cases, however, it will not be a part of this thesis.

Chapter 6

Conclusion

We have described in this thesis the design, implementation and a few applications of a new network file protocol: πp. We believe πp will live up to our original expectations of a simple, distributed, reliable, versioned, and caching protocol that makes up for a few disadvantages that we identified in existing protocols.

However, our work is far from complete. We still have more work to do in order to prove that πp indeed performs better than other network file protocols when deployed in real world applications. In particular, the following tasks still remain un-tackled:

- A clearer and more robust mechanism to implement ACLs and permissions in the protocol.
- A set of client-side and server-side tools that utilize the protocol to implement real world applications for testing and performance evaluation purposes.
- An on-disk versioning filesystem that maps closely to the expectations laid out by the protocol.
- A clearer understanding of the role that naming services (such as DNS) play in a πp based network.
- Algorithms and data-structures that can be used by πp based file servers and caches to manage a large number of files and users concurrently.

However, it is our belief that the πp protocol as described in this document provides for a solid base to work on these tough challenges and is hopefully a significant contribution towards our primary goal of providing a better, faster networking service to end users.

Appendix A

Protocol Operations

First we define constants:

```
enum {
    Port  = 564    // default port for file servers
    Msize = 32768, // default message size
};

// Special values
enum {
    Notag = ~0,
    Nofid = ~0,
    Nouid = ~0
};

// Ftype flags
enum {
    Freg       = 0x0,
    Fdir       = 0x1,
    Fappend    = 0x2,
    Fversioned = 0x4
};
```

Now, we proceed to describe each operation: it's binary code as well as arguments.

```
"Tsession": [
    {"code": "100"},
    {"Csid": "u32int"},
    {"Afid": "u32int"},
    {"Msize": "u32int"},
    {"Options": "string"}
],

"Rsession": [
    {"code": "101"},
    {"Ssid": "u32int"},
    {"Afid": "u32int"},
    {"Msize": "u32int"},
    {"Options": "string"}
],
```

```
"Tattach": [
    {"code": "102"},
    {"Fid": "u32int"},
    {"Afid": "u32int"},
    {"Uname": "string"},
    {"Aname": "string"}
],

"Rattach": [
    {"code": "103"},
    {"Afid": "u32int"}
],

"Rerror": [
    {"code": "105"},
    {"Ename": "string"}
],

"Tflush": [
    {"code": "106"},
    {"Tag": "u32int"}
],

"Rflush": [
    {"code": "107"}
],

"Topen": [
    {"code": "108"},
    {"Fid": "u32int"},
    {"Nfid": "u32int"},
    {"Path": "string"},
    {"Mode": "string"}
],

"Ropen": [
    {"code": "109"},
    {"Ftype": "u32int"},
    {"Version": "u64int"}
],

"Tcreate": [
    {"code": "110"},
    {"Fid": "u32int"},
    {"Name": "string"},
    {"Perm": "u32int"},
    {"Mode": "string"},
    {"Ftype": "u32int"}
],

"Rcreate": [
    {"code": "111"},
    {"Version": "u64int"}
],

 "Tread": [
    {"code": "112"},
    {"Fid": "u32int"},
    {"Offset": "u64int"},
    {"Count": "u32int"},
    {"Attrs": "string"}
],
```

```
"Rread": [
    {"code": "113"},
    {"Dat": "data"}
],

"Twrite": [
    {"code": "114"},
    {"Fid": "u32int"},
    {"Offset": "u64int"},
    {"Dat": "data"},
    {"Attrs": "string"}
],

"Rwrite": [
     {"code": "115"},
     {"Count": "u32int"}
],

"Tremove": [
    {"code": "116"},
    {"Fid": "u32int"}
],

"Rremove": [
    {"code": "117"}
],

"Tclose": [
    {"code": "118"},
    {"Fid": "u32int"},
    {"Commit": "u16int"}
],

"Rclose": [
    {"code": "119"},
    {"Version": "u64int"}
],

"Tclunk": [
    {"code": "120"},
    {"Ssid": "u32int"}
],

"Rclunk": [
    {"code": "121"}
]
```

The operation code themselves are 4-bytes long (u32int). All integers are represented in network-endian order. `strings`s are prefixed with a u32int representing their length, and `data` objects are prefixed similarly. The argument names are provided for convenience only and play no role in the binary protocol itself. The arguments are expected to be present in the exact same order as presented here (top-down).

Appendix B

Protocol Extensions

B.1 Leasing

We introduce the following new protocol operations in order to implement a leasing system on file versions. This extension is requested by including the string "lease" in the '`options`' argument of a `Tsession`.

```
"Tlease": [
    {"code": "124"},
    {"Fid": "u32int"}
],

"Rlease": [
    {"code": "125"}
    {"Expires": "u64int"}
],

"Trenew": [
    {"code": "126"}
],

"Rrenew": [
    {"code": "127"}
],

"Trevoke": [
    {"code": "128"},
    {"Fid": "u32int"}
],

"Rrevoke": [
    {"code": "129"}
]
```

The goal of this protocol extension is to make it easier for caches to keep their local copies of files up to date by obtaining 'leases' on particular versions of files. A client (or in this case, a cache) may request a lease on a version of the file by using the `Tlease` operation. The server responds with an `Rlease` indicating that the lease is valid until '`expires`'.

Shortly before the lease expires the client may request a renew for *all* leases issued to it with a `Trenew` operation. The server responds with an `Rrenew` to indicate that the leases were renewed successfully for the same time period for which the first lease was granted. If an individual file's lease has expired, the server additionally sends a `Trevoke` message indicating to the client that the lease for that fid is no longer valid. The client must acknowledge with an `Rrevoke` message.

This scheme is explained in more detail in a separate thesis dealing with the reliability and consistency of πp caches.

B.2 Retransmission

We noted earlier that it is possible to run πp over an unreliable transport network while still ensuring consistency with respect to operations that are no idempotent. For this purpose it is necessary to implement a special server that buffers responses to messages until an acknowledgement that the client has received them is received. We introduce three new operation types to enable this:

```
"Tack": [
    {"code": "130"},
    {"Tag": "u32int"}
],

"Tenq": [
    {"code": "132"},
    {"Tag": "u32int"}
],

"Renq": [
    {"code": "133"},
    {"Tag": "u32int"}
]
```

Clients use timeout and retransmission for ensuring that their requests reach the server. The server must execute messages with the same tag only once, irrespective of the number of times it has received the message. The client may send the `Tenq` message to a server in order to monitor progress with respect to a particular message. The response to a `Tenq` is either an `Renq` indicating that the server has received the message but is still processing it or the response to the message itself. A server may also preemptively send a client an `Renq` without having received a `Tenq` if it anticipates that processing a message may take a long time.

A server must hold on to all messages it transmits to clients until it receives a `Tack` for that message subsequently (this `Tack` may be part of the next message sent by the client. If no `Tack` is received the server must retransmit the response periodically until it receives the corresponding `Tack` from the client.

Bibliography

[1] A. Bhushan. A file transfer protocol. *IETF RFC 114*, 2010.

[2] Peter J. Braam. The coda distributed file system. *http://www.coda.cs.cmu.edu/ljpaper/lj.html*, 1998.

[3] Peter J. Braam. Why don't we just reimplement coda from scratch? *http://www.coda.cs.cmu.edu/misc/sloccount.html*, 2002.

[4] Neil Carpenter. Smb/cifs performance over wan links. *http://blogs.technet.com/b/neilcar/archive/2004/10/26/247903.aspx*, 2004.

[5] Microsoft Corp. Common internet file system protcol (cifs/1.0). *http://tools.ietf.org/html/draft-heizer-cifs-v1-spec-00*, 1996.

[6] J. Reynolds J. Postel. File transfer protocol (ftp). *IETF RFC 959*, 2010.

[7] Shane Kerr. Use of nfs considered harmful. *http://www.time-travellers.org/shane/papers/NFS_considered_harmful.html*, 2000.

[8] Ratul Mahajan. How akamai works. *http://research.microsoft.com/en-us/um/people/ratul/akamai.html*, 2001.

[9] Sun Microsystems. Nfs: Network file system protocol specification. *IETF RFC 1094*, 1989.

[10] W3C/MIT et. al. R. Fielding, T. Berners-Lee. Hypetext transfer protocol – http/1.1. *IETF RFC 2616*, 1999.

[11] Dennis M. Ritchie Rob Pike. The styx architecture for distributed systems. *http://www.vitanuova.com/inferno/papers/styx.html*, 1999.

[12] D. Noveck et. al. S. Shepler, M. Eisler. Network file system (nfs) version 4 minor version 1 protocol. *IETF RFC 5661*, 2010.

[13] Opera Software. Opera unite. *http://unite.opera.com/*, 2009.

[14] Cisco Systems. Vni forecast. *http://ciscovni.com/vni_forecast/*, 2010.