

Network Operating Systems

Rahul Murmura, George Mason University

July 1, 2009

Abstract

The objective of this project is to identify what comprises a network operating system, understand the `/net` filesystem architecture of Plan 9 from Bell Labs and implement it in the Linux Kernel as a synthetic filesystem. This project will empower the kernel with certain network related functions which, if previously present, were handled by the sockets and other POSIX-compliant libc libraries in userspace.

Contents

1	Network Operating System	2
1.1	Components of Network Operating Systems	2
2	Plan 9 from Bell Labs	3
2.1	Plan 9's network internals	3
2.2	Advantages of <code>/net</code> filesystem's design	4
2.3	Arguments against Network Transparency	5
3	Glendix	6
3.1	Compile once execute everywhere	6
3.2	Primary channels of implementation	6
3.3	Per-process namespaces and Union Mounts	7
4	Implementation: <code>/net</code> in Linux	8
4.1	Basic design	8
4.1.1	Synthetic Filesystems	8
4.1.2	Current socket architecture Vs <code>/net</code>	9
4.1.3	The proposed file based alternative	10
4.2	Status report on completed work	11
4.2.1	DNS: a tricky nut to crack	12
4.2.2	socknetlib Vs fsnetlib	12
4.3	Proposed offshoot project: better routing support	13
5	Conclusion	13

6 Acknowledgements	14
References	14
A Appendix	15
A.1 Project Log: Timeline	15
A.2 Quagga: Routing support on GNU/Linux	16
A.3 NOX: Network Operating System	17

1 Network Operating System

A Network Operating System (NOS) is software which controls and manages a network. Usually, it refers to whatever software is running the routers. This automatically reflects that router hardware companies invest to create their own operating systems which help a user configure those routers, e.g. CiscoIOS, JUNOS. There are also some general purpose operating systems which have specialized distributed computing features, and on customization, can make very good routers, e.g. Plan 9 from Bell Labs.

While general purpose operating systems like Mac OS X, Windows and GNU/Linux have the many of the above features, they do not qualify as a NOS in our sense of the term. A NOS is a system that is specifically written to implement and maintain computer networks.

1.1 Components of Network Operating Systems

Generally speaking, a Network Operating System is a system that provides some of the following features[14]:

- Basic support for hardware ports (interfaces) and capability for per-port network configuration
- Security features such as authentication, authorization, login restrictions, and access control
- Packet filtering, network address translation (NAT) and firewall
- Name services and directory services
- File, print, data storage, backup and replication services
- Remote access
- System management
- Network administration and auditing tools with graphic interfaces
- Clustering capabilities
- Fault tolerance and high availability
- Routing and switching

2 Plan 9 from Bell Labs

Plan 9 from Bell Labs[9] was originally led by Rob Pike, Ken Thompson, Dave Presotto and Phil Winterbottom with support from Dennis Richie as head of the Computing Techniques Research Department. This is more or less the same team that originally built the UNIX decades before. They started this project with a goal to fix every mistake made in the design of UNIX. An extract from Wikipedia says:

“Plan 9 from Bell Labs is a distributed operating system, primarily used for research. It was developed as the research successor to Unix by the Computing Sciences Research Center at Bell Labs between the mid-1980s and 2002. Plan 9 is most notable for representing all system interfaces, including those required for networking and the user-interface, through the filesystem rather than specialized interfaces. Plan 9 aims to provide users with a workstation-independent working environment through the use of the 9P protocols. Plan 9 continues to be used and developed in some circles as a research operating system and by hobbyists. The name Plan 9 from Bell Labs is a reference to the 1959 cult science fiction B-movie Plan 9 from Outer Space.”

Plan 9 from Bell Labs has been designed from ground up to work with hardware spread all across a network. This native or in-kernel support for distributed computing is a salient feature of Plan 9.

Plan 9 has already found its way to the compute nodes and I/O nodes on Blue Gene/L, a computer architecture project for supercomputers, cooperatively undertaken by IBM, the Lawrence Livermore National Laboratory, United States Department of Energy and academia[4].

2.1 Plan 9’s network internals

Plan 9’s network internals has been discussed on their documentation website[7]. There is a `/net/cs` file which is basically the Connection Server. This file helps in DNS translation. When we write “`net!www.google.com!http`”, the operating system queries a DNS server which need not be a Plan 9 system. The file `/net/cs` then returns – “`/net/tcp/clone 74.125.67.100!80`”. There are many other files and folders inside the `/net` directory. The `/net` in my virtual machine looks like this:

```
> term% ls /net
/net/arp
/net/bootp
/net/cs
/net/dns
/net/ether0
/net/icmp
/net/icmpv6
```

```
/net/ipifc
/net/iproute
/net/ipselftab
/net/log
/net/ndb
/net/tcp
/net/udp
```

After getting back the IP address of the server we are trying to connect to, we need to read from the file `/net/tcp/clone`. Doing this will return a folder under `/net`, say `/net/tcp/6`. Inside that numbered connection folder under the interface, there are files named *data*, *ctl* and *status* among others. In order to make a connection we need to pass the command “connect 74.125.67.100!80” to the *ctl* file. The *status* file will reflect the status of the connection. When connected, writing to that *data* file will send the data to the server, and reading from it will return information received from the other end.

NOTE:- The above drill of writing to `/net/cs` and subsequently writing to the clone file can be either done using simple `cat` and `echo`, as discussed below in section 4.1.3. It can also be done using the “dial” function from the `libc` in C programs, or using the program `ndb/csquery`.

2.2 Advantages of `/net` filesystem’s design

When UNIX was originally designed, there were no computer networks. Yes, BSD Sockets were designed first on a UNIX, but that does not imply that UNIX is the best possible design for a networking enabled operating system. Today, all operating systems – including Mac OS X, GNU/Linux, Solaris and Windows – are direct derivatives of the original UNIX first designed over 40 years ago!

So folks at Bell Labs (read Rob Pike, Ken Thompson, Dave Presotto, Phil Winterbottom, Dennis Ritchie, Brian Kernighan, et. al.) designed a new network structure, that exposes the kernel TCP/IP stack in form of filesystems to the user space, instead of the traditional Socket library approach. Plan 9 has been designed from ground up, with networking in mind, and is inherently a distributed operating system. Here’s an extract from Wikipedia:

“Plan 9 does not have system calls for the multitude of communication protocols or device driver interfaces. For example `/net` is the API for all TCP/IP, and it can be used even with scripts or shell tools, writing data to control files to write and read connections. Relevant sub-directories like `/net/tcp` and `/net/udp` are used to interface to respective protocols. You can implement a NAT by mounting a `/net` from a perimeter machine with a public IP, while connecting to it from an internal network of private IP addresses, using the Plan 9 protocol 9P in the internal network. Or you can

implement a VPN by mounting a /net directory from a remote gateway, using secured 9P over the public Internet.”

Features like packet filtering are currently in process of being added to Plan 9. These modern day networking features were originally eliminated from the design to keep the core networking module simple. You can mount the /net of any node onto any other node having access priviledges to do so. If the node serving the /net is a perimeter node having access to both the inside and the outside network, it inherently becomes the gateway for that node which remote-mount its /net filesystem. This design provides network transparency i.e. the same tools and applications can be used to work with both local and network resources; section 2.3 discusses some of the arguments against this design and the corresponding clarifications from the Plan 9 team.

When a connection is made using the /net filesystem, all reads and writes are just byte streams. All communication is independent of byte ordering, CPU type or text/binary mode and completely depends on what the application needs. What you send, is exactly what you get on the other end of the stream.

Another advantage would be network management. On most UNIX-like systems, several files such as /etc/hosts, /etc/networks, /etc/services, /etc/hosts.equiv, /etc/bootptab, and /etc/named.d hold network related information. Much time and effort is spent administering these files and keeping them mutually consistent. Tools attempt to automatically derive one or more of the files from information in other files but maintenance continues to be difficult and error prone. The task specially becomes humongous when dealing with a network of computers. One of the interesting solutions proposed in the area is NOX, an operating system for network management. In the world of /net however, there is something called Network Database, which is a set of two simple ASCII files, which hold all relevent information including domain names, IP addresses, Ethernet addresses, boot file location and supported protocols.

2.3 Arguments against Network Transparency

Joel Spolsky from Microsoft argues in his article “Three Wrong Ideas From Computer Science” against network transparency[12]. He suggests that it is a bad idea to have technologies like remote procedure call (RPC) and Microsoft’s Distributed COM (DCOM) where the user is oblivious of whether a particular resource exists locally or on the network. Specially in cases when the network is not perfectly reliable. Leaving network failiures apart, there is also the problem of applications remaining frozen until the given task is completed. If the files we are working with are larger and over the network, this time could stretch long enough for the user to think it was a malfunction. As a compromize, he suggests that this problem could be reduced if some exception handling was implemented, and the application quit the task instead of waiting on it.

In a discussion on this article on the Plan9 mailing list, Dan Cross[3] explains that in Plan 9, the idea of network transparency is slightly different. The technique was implemented to provide a common interface to users to deal with

network and local resources, in this case file read/write. However, the important difference is that programmers still interact with the network resources under the `/net/tcp` knowing that they are interacting with the network. The tools used are the same as they would use for local resources, and an application does not need to access complex POSIX libraries to talk to the network, thereby simplifying the design of basic applications greatly. The difference is only psychological and the method to access remote files is no longer the focal point of the application design, and rightfully so. When handled correctly, network transparency only makes things convenient for the user.

3 Glendix

While people have already ported some of the ideas of the Plan 9 Operating system to Linux, like `utf-8` and `/proc` filesystem, bulk of the ideas, like per-process namespaces, and union mounts are still only existant in the Plan 9 world. This is why the Glendix project began. The Glendix project is about bringing the beauty of Plan 9 from Bell Labs to the world of Linux. The primary motivating factor is to promote the Plan 9 style of application development to the large base of developers that Linux already has. The secondary factor is to reduce Linux kernel's dependancy on the GNU based userspace software.

3.1 Compile once execute everywhere

This is the mantra described in the Glendix IWP9 2008 paper[1]. As described in the paper, there are various ways to provide compatibility between different systems. "Plan 9 from User Space" (also known as `plan9port`) is an existing software package for POSIX compliant operating systems that consists of ports of several Plan 9 applications. While most of Plan 9's libraries have also been ported, this solution is not completely perfect. A more appealing solution was to achieve binary-level compatibility of all Plan 9 applications. The initiative was to ensure that it would not matter where the program was compiled, it should run as expected on both Plan 9 and Linux.

In order for this approach to work, we have to make Linux behave exactly as Plan 9 kernel would, as far as applications were concerned.

3.2 Primary channels of implementation

There are two primary channels for an application to access functionality provided by the Plan 9 kernel: system calls and file servers. If both were to be implemented in Linux Kernel, userspace applications should be oblivious to the fact that the underlying kernel is Linux and not Plan 9.

The Glendix team has already completed a loader which understands the Plan 9's executable format (`a.out`). Implementations of the system calls that the Plan 9 binaries make is underway. Along with the system call handler, 15 of the 39 system calls have been completed. Suprisingly, a number of applications like

8c (the Plan 9 C Compiler), sed, grep, echo, cat, tar, cb, cal and dc already work in Glendix. These Plan 9 binaries get native compatibility in the Linux kernel. There are however, a number of applications that assume the presence Plan 9 filesystems like /net and /dev/draw which provide networking and drawing (Linux framebuffer equivalent for Plan 9). In order to support these binaries, the Glendix project requires kernel-level implementations/extensions providing these synthetic filesystems along the same designs as /proc and /dev in Linux, which were also originally Plan 9 ideas later ported to Linux.

3.3 Per-process namespaces and Union Mounts

The document *Use of Name Spaces in Plan 9* [8] on the Plan 9 website discusses how file systems along with per-process namespaces can help solve many issues often left to more exotic mechanisms. Wherever possible, Plan 9 has been designed to represent resources, both local and remote, as heirarchical file systems; and a user or a process aquires a private view of this filesystem by constructing a name space depending on the priviledges given to it.

Although Linux has had this feature for a long time now, you need to be a root user to do this. The problem lies in un-priviledged users gaining system-wide access by confusing the setuid programs if given rights to name spaces. Combined with bind mounts, people have created some useful setups under the current Linux. For example:

```
$ mkdir ~/mytmp
$ touch ~/mytmp/test
$ mount --bind ~/mytmp /tmp
$ ls /tmp
test
$
```

Now, without per-process namespaces, the new /tmp would be visible to the entire system. However, if a process did the above in its own namespace, then all its subprocesses will see ~/mytmp mount at the mount point /tmp, and all external processes will see the main /tmp. This design could solve many problems very trivially, like having the libraries of 32-bit as well as 64-bit libraries under /usr/lib so that applications do not have to worry about checking for /usr/lib32 everytime.

For all of this to work, we need to eliminate root user from the designs of the Linux system and use Plan 9 like security[10]. Once that is done, name spaces can be opened to every user, and a user will only get a view of the system based on the priviledges the user has. Under this new design, there is no 'system owner', but there is a owner for every filesystem. Many non-Plan 9 technologists have proposed similar ideas under the name 'multi-root user system'[13], independent of the Plan 9-way of dealing with users and security. Also, union mounts are one step better than bind mounts. As the name suggests, unin mounts do not hide the contents of the destination folder, where the

filesystem is being mount. There is a way to specify priority in case of name clashes among the contents of the two folders, however. There has been multiple projects which are together trying to achieve all of the above, in near future. Glendix aims to incorporate all of those which are based on the Plan 9's designs.

4 Implementation: /net in Linux

This report describes my first implementations of the /net as a virtual filesystem on the Linux kernel that works like the Plan 9 counterpart. Over 1000 lines of code has been written so far, and according to Ohloh[6], 29% of my code is comments and white spaces. The Glendix project is already working on providing binary compatibility on the Linux kernel for Plan 9 applications, and once Linux has a /net, the Plan 9 applications which look for a /net can also feel at home. Advantages of using a /net filesystem in place of sockets has already been discussed in section 2.2. Work on this project is somewhat endless. Once the /net is brought up to completion equivalent to that of Plan 9, development will continue to shape the /net for both kernels together.

4.1 Basic design

4.1.1 Synthetic Filesystems

Linus Torvalds and numerous other kernel developers dislike the ioctl() system call, seeing it as an uncontrolled way of adding new system calls to the kernel. The 2.6 kernel contains a set of routines called "libfs" which is designed to make the task of writing virtual filesystems easier[2]. libfs handles many of the mundane tasks of implementing the Linux filesystem API. Using synthetic filesystems, we can define exactly what we want done when someone writes to a file, or reads from a file. We can also create or delete files on-the-fly.

Before writing our own filesystem, it is recommended to try implementing a sample FS. LWN.net for provides this piece of documentation[2] for those building a virtual file system for the first time. The Makefile should look something like:

```
ifneq ($(KERNELRELEASE),)
obj-m := lwnfs.o
else
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
endif
```

After running Make, a kernel object is generated. We need to mount this filesystem. Before we can do that, this module has to be loaded into the running kernel. This is done using insmod command for one-time only.

```

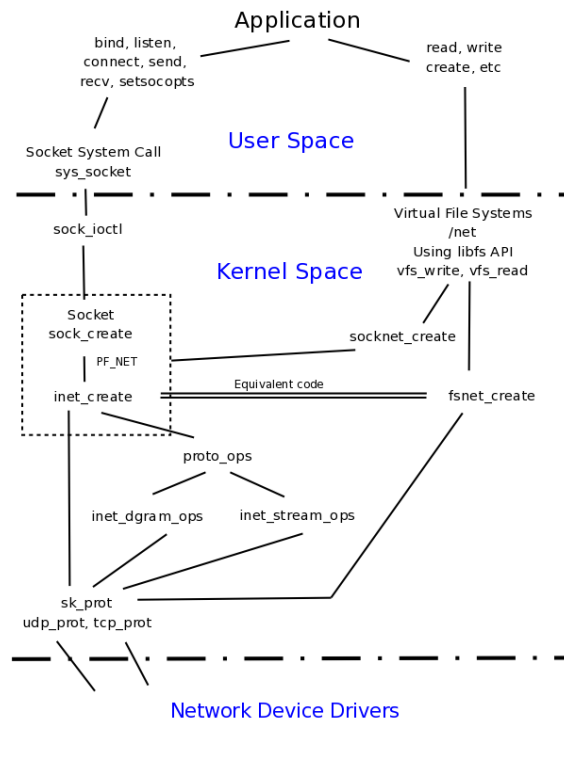
glendix ~/lwnfs $ make
glendix ~/lwnfs $ insmod lwnfs.ko
glendix ~/lwnfs $ mount -t lwnfs none /lwnfs

```

By looking at this `/lwnfs/counter` we can understand the structure of libfs and exactly how to use it. More complex implementations can be found in the Linux kernel like the `/proc` filesystem and the debugfs.

4.1.2 Current socket architecture Vs /net

The current socket design in the Linux kernel is well explained in the book, *TCP/IP Architecture, Design and Implementation in Linux*[11]. The figure below is a broad flow diagram showing the current socket architecture on the left side and the proposed parallel design on the right. The way the network stack is currently designed is that there is a TCP/IP stack, calls to which is made by the `sk_prot` structure of the sock layer. This layer is a common interface to the various transport layer protocols implemented in the kernel. This structure provides objects to the socket layer. The socket layer is responsible to create and maintain kernel sockets that store connection states, and handle system calls via `sock_ioctl` like `sys_socket`, `send`, `recv`, `accept`, `listen`, `bind`, `connect` and others from user-space.



These sockets store information using file descriptors on disk. These file descriptors are not complete files. Also, by using this method, it became compulsory for user-space applications to use the socket library provided by the POSIX compliant libc.

4.1.3 The proposed file based alternative

Instead of using the POSIX socket interface, Plan 9 from bell Labs introduced performing simple read and write calls to virtual files which store the connection states in place of socket structure. The idea is that we should be able to run a shell script like below, and perform network programming.

```
#!/bin/rc
server=$1
port=80
clonefile=/net/tcp/clone
netdir=()
fn showme {
while (~ '{cat $netdir/status} Established*) {
line='{read}
if (! ~ $#line 0)
echo $line
}
exit
}
<[5] $clonefile {
netdir='{basename -d $clonefile} ~ / ~ '{cat /fd/5}
echo connect $server!$port >$netdir/ctl || exit 'Cannot Connect'
echo connected to tcp!$server!$port on $netdir
cat $netdir/data | tr -d '
' | showme &
while (cmd='{read}) {
echo $cmd > $netdir/data
}
}
```

To use this script, run it with the server ip or url as an argument, which will be picked up by \$1. After running this command on rc from plan9ports, enter the request as :

```
GET / HTTP/1.0
Host: <hostname>
```

The requested page will be fetched and displayed on the shell. The way this script works is that we first receive a connection folder number by simply reading /net/tcp/clone. By doing that, the kernel returns the next available folder as \$netdir, which has files like *ctl*, *data* and *status*. If the currently created folders are all being used, the filesystem automatically creates a new folder named $n+1$ where n was the name of the last active folder.

```

> term% cat /net/tcp/clone
> 1
> term% ls /net/tcp/
/net/tcp/0
/net/tcp/1
/net/tcp/clone
/net/tcp/data
/net/tcp/status

```

Next, we write a command `connect <url/ip>!<port>` to the `/net/tcp/1/ctl` file, and that should send a tcp handshake request to the server url mentioned on the port specified. Once connection is established, we can write to the `data` file of our connection folder to send data to the server and read from that file to receive data sent by the server. In the given script, which is to be run on client-side, we send a HTTP Get request to the server on port 80 and get back the content for the page requested. All of this is done without using any socket ioctl. Note that a script was used for simple cat and echo commands instead of directly working in the shell, because when we `echo connect <url/ip>!<port>` to the ctl file, and exit to terminal again for next command, the file and the connection get closed.

4.2 Status report on completed work

The core of the netfs code is in `net.c` where we define the actual inode details. Here, we use the libfs API and give a name to the filesystem, in our case “net”. This name is used while mounting the filesystem. The functions `slashnet_create_dir` and `slashnet_create_file` define what to do when we need to create files in our filesystem. There is another function, `slashnet_create_files` which lists the files and folders that are initially need to be created as soon as the user runs command `mount -t net none /net`. As of this document’s writing, we see the following filestructure when executing this mount command.

```

Glentoo ~ # hg clone http://hg.glendix.org/glendix
Glentoo ~ # cd glendix/netfs/
Glentoo netfs # mount -t net none /net
Glentoo netfs # ls /net/
cs ether0 tcp udp
Glentoo netfs # ls /net/tcp/
0 clone stats
Glentoo netfs # ls /net/tcp/0
ctl data
Glentoo netfs #

```

The revision <http://hg.glendix.org/glendix/rev/6fdffc287daf> demonstrates a way to use `tcp.c`, `cs.c`, `ether.c` etc. to do the file-specific processing, while passing only the file structure `*flp` from a common `write_file` function in `net.c`. The filesystem now occupies some memory as soon as it is mount - `TMPSIZE` *

no_of_files to be precise. I would say this is desirable. I dont know with respect to code, but Plan 9 also seems to behave in the same fashion as far as usability is concerned.

The main files to be implemented were `/net/cs`, `/net/tcp/clone` and the `/net/tcp/n/ctl`. Doing these will be enough to test out a simple connection from command-line. Plan 9 uses “connect 192.168.1.8!80” as the command. Bash does not allow the symbol '!'. I dont know if it will work if I used other shells from the UNIX world, but using `rc` from `plan9ports` solves the problem. There were many roadblocks along the path of our implementation of these special files. Following is the discussion of currently open issues.

4.2.1 DNS: a tricky nut to crack

The `/net/cs` refers to Connection Server. This works as a DNS resolver in Plan 9 kernel. Apparently, Linux kernel does not seem to have a DNS resolver. The `glibc` function in GNU/Linux systems, `gethostbyname()`, first checks `/etc/hosts` and if no entries are found, it makes a direct socket connection with the ip address given in `resolv.conf` and fetches the DNS resolved ip address. It is yet to be evaluated whether we should at this time go ahead with our own in-kernel DNS implementation. There has been arguments regarding the fact that a DNS query is a user-space activity. So did the Linux Kernel Hackers actually chose not to have DNS in-kernel for some real-world reasons? This question is still open and the `/net` filesystem I am currently working on, is meant only to work with ip addresses.

4.2.2 socknetlib Vs fsnetlib

When relevant commands are written to `/net/tcp/n/ctl` we need to process them. This is done by calling a process function from the write function. This process function makes calls to execute the actual network instructions and returns the result to our `/net` filesystem. Slashnet’s repository has two separate programs, `socknetlib.c` and `fsnetlib.c`. Each of these hook onto a different point on the socket/network architecture of current Linux kernel.

While it might be desirable to go the `fsnetlib` way, because it makes calls directly to the tcp stack by setting the `sk_prot` structure, much effort will be needed to complete this library. Hence `socknetlib` is being written to finish some functions to get proof of concept. What my `socknetlib` does is it just makes calls to the `sock` structure in the kernel which maintains socket state in-kernel.

One of the problems that we faced at this juncture was that if I dont have `sock->sk` element, because I dont use a socket to represent a connection, but use a folder `/net/tcp/n`, then how to I associate the `sk` structure with it? *struct file* has no `sk` element like *struct socket* does. The solution was found in the inode definition of connection folders. It stores the pointer to the `sock` structure as the `i_private` variable under the `dentry` structure of the connection directory. Hence, we are still by-passing the socket `ioctl` even though a lot of the socket architecture is still maintained in kernel when following `socknetlib`’s design.

It is believed that there will not be any performance difference in either approach, because both are in kernel-space. It is just a question of least disruptive short-term design Vs ideal, complex and long-term design.

4.3 Proposed offshoot project: better routing support

The subject of this offshoot project is “Improving /net/iproute for advanced routing protocols in Plan 9”. The main idea is to be able to run Plan 9 and Glendix on routers. It seems that /net/iproute is a good place to start. It has a complete interface for editing routes. Currently ip/rip implements a simple routing protocol in user-space and writes to /net/iproute. What we need is a user space script that implements more advanced routing protocols, like <http://www.openbgp.org/> does on OpenBSD and Quagga on Linux.

The deliverable is to implement a routing protocol alongside ip/rip, say either ip/ospf or ip/bgp in user-space. The implementation of these protocols are not as straight forward as rip. These protocols require more information from the network like traffic metrics and distance information. The task will be to improve /net/iproute to incorporate whatever will be necessary and include some ability for interface-specific configurations and decision-making. The advantage will be that we can implement new routing protocols and test them easily on real networks. We should be able to do things like configure one NIC with ip/rip and a second NIC as ip/new_rout_prot if needed.

A good approach will be to make some sample routing test files. These will both illustrate the information that we want iproute to provide us and the manner in which it should be provided. Let our protocol consume these files and give it the ability to dump out its datastructures in such a way that we can do some verification on its reliability and correctness. At the end of this phase, we will have finalized the interface for userspace scripts to define routing, designed the interface for getting the required info from /net/iproute and ironed out all bugs with the routing protocol implementation.

A large part of the project is undecided. The primary objective is to bring Plan 9 / Glendix to routers, and it sounds like a good direction of work in the near future!

5 Conclusion

Originally just a project that was needed because some of Plan 9’s applications, when ported to Linux, looked for a /net filesystem; the scope of this project has been extended to empower Linux kernel with some tools previously not available. I find the concept of TCP/IP stack exposed as a filesystem “/net” to be a really powerful design. Being able to work with ASCII text written to files instead of complex architecture specific code heavily depending on dynamic libraries really intriguing. I plan to explore newer network architectures made possible because of Plan 9’s approach.

While this document mirrors most of Plan 9's ideas extended with some of my own imagination, there is a great deficit between what has really been implemented so far. The project is only in its early stage, but with this document, I attempt to provide a overview of exactly what we have set fourth to achieve. The Glendix team strongly believes in this project and I hope more developers will join now to provide thier technical expertize in order to take this project forward. The readers are welcome to provide suggestions and apprehension against the proposed design.

6 Acknowledgements

This project is a subset of the larger Glendix project. I thank Anant Narayanan and others involved in starting the Glendix project for giving me such a platform to work on something so exciting. I would like to thank the entire Glendix team for being supportive and giving valuable technical suggestions this far.

This report is aimed at being the final deliverable for my independent reading and research at George Mason University under Dr. Brian Mark. I would like to thank Dr. Mark for guiding me and giving me the opportunity I was looking for, to enhance my ideas in Networking and Operating Systems.

References

- [1] Vinay Pamarthi Manoj Gaur Anant Narayanan, Shantanu Choudhary. Glendix: A plan9/linux distribution. In *IWP9 '08: Proceedings of the 3rd International Workshop on Plan 9*, pages 1–8. Bell Labs, 2008.
- [2] Corbet. Creating linux virtual filesystems, 2003.
- [3] Dan Cross. Re: Essay: Is network transparency something bad?, 2002.
- [4] Jim McKie Ron Minnich Eric Van Hensbergen, Charles Forsyth. Petascale plan 9 on blue gene, 2007.
- [5] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, 2008.
- [6] Ohloh.net. Gliendix contributors, 2009. [Online; accessed 01-July-2009].
- [7] Dave Presotto and Phil Winterbottom. The organization of networks in plan 9. Technical report, Bell Laboratories, Murray Hill, New Jersey 07974, USA.
- [8] Ken Thompson Howard Trickey Phil Winterbottom Rob Pike, Dave Presotto. The use of name spaces in plan 9. Technical report, Bell Laboratories, Murray Hill, New Jersey 07974, USA.

- [9] Sean Dorward Bob Flandrena Ken Thompson Howard Trickey Phil Winterbottom Rob Pike, Dave Presotto. Plan 9 from bell labs. Technical report, Bell Laboratories, Murray Hill, New Jersey 07974, USA.
- [10] Rob Pike Dave Presotto Sean Quinlan Russ Cox, Eric Grosse. Security in plan 9. Technical report, Bell Laboratories, Murray Hill, New Jersey 07974, USA.
- [11] M. Ajaykumar Venkatesulu Sameer Seth. *TCP/IP Architecture, Design and Implementation in Linux*. Wiley-IEEE Computer Society Press, 2008.
- [12] Joel Spolsky. Three wrong ideas from computer science, 2000.
- [13] Will Varfar. Network transparency and the multi-root filesystem, 2004.
- [14] Wikipedia. Network operating system — wikipedia, the free encyclopedia, 2009. [Online; accessed 27-June-2009].

A Appendix

A.1 Project Log: Timeline

```

changeset: 87:31c5adbe25aa
branch: slashnet
user: Rahul Murmuria <rahul@murmuria.in>
date: Sun May 03 16:57:36 2009 -0500
summary: Using sock->ops->connect causes kernel to crash...
        WARNING: Non-functional code being commit
.
changeset: 86:1ee17ef3d2ba
branch: slashnet
user: Rahul Murmuria <rahul@murmuria.in>
date: Sun May 03 00:09:10 2009 -0500
summary: Added processing code for /net/tcp/n/ctl
        starting with "connect <ip>!  
<port>"
.
changeset: 85:ef351368303b
branch: slashnet
user: Rahul Murmuria <rahul@murmuria.in>
date: Tue Apr 07 06:31:41 2009 -0500
summary: Initial check-in of socklib and fsnetlib
.
changeset: 84:9d1ec2b3b2ac
branch: slashnet
user: Rahul Murmuria <rahul@murmuria.in>
date: Sat Apr 04 21:19:27 2009 -0500
summary: Added README for trying out netfs code

```

```

.
changeset: 83:6fdffc287daf
branch: slashnet
user: Rahul Murmuria <rahul@murmuria.in>
date: Sat Apr 04 21:17:51 2009 -0500
summary: Tutorial: How to process the written
        instructions in netfs
.
changeset: 82:5fe1f801a6c6
branch: slashnet
parent: 74:18b8b6d0f876
user: Rahul Murmuria <rahul@murmuria.in>
date: Sat Apr 04 19:25:43 2009 -0500
summary: Seperate buffers for every file.
        filp->private_data works now!
.
changeset: 74:18b8b6d0f876
branch: slashnet
user: Rahul Murmuria <rahul@murmuria.in>
date: Tue Mar 31 08:00:58 2009 -0500
summary: Added ether.c and tcp.c
.
changeset: 70:5147a06b8d67
branch: slashnet
user: Rahul Murmuria <rahul@murmuria.in>
date: Mon Mar 30 03:08:06 2009 -0500
summary: Fixed sizeof to strlen in read_file,
        distributed the code to different c files
.
changeset: 59:092b922b74ce
user: Rahul Murmuria <rahul@murmuria.in>
date: Sat Mar 28 17:59:04 2009 -0500
summary: Fixed a memory leak
.
changeset: 53:92dc24ff2dce
user: Rahul Murmuria <rahul@murmuria.in>
date: Mon Feb 23 17:52:27 2009 -0500
summary: Initial check-in of netfs -
        currently working on /net/cs

```

A.2 Quagga: Routing support on GNU/Linux

Quagga is a network routing suite providing implementations of OSPF (v2 & v3), RIP (v1, v2 & v3) and BGP (v4) and IS-IS for Unix-like platforms, particularly FreeBSD, Linux, Solaris and NetBSD. The Quagga architecture consists of a core daemon (zebra) which acts as an abstraction layer to the underlying

Unix kernel and presents the Zserv API over a Unix or TCP stream to Quagga clients. It is these Zserv clients which typically implement a routing protocol and communicate routing updates to the zebra daemon. Existing Zserv clients are: ospfd (implementing OSPFv2); ripd (implementing RIP v1 and V2); ospf6d (implementing OSPFv3 - (IPv6)); ripngd (implementing RIP ng (IPv6)); bgpd (implementing BGPv4+ (including address family support for multicast and IPv6))

Additionally, the Quagga architecture has a rich development library to facilitate the implementation of protocol/client daemons, coherent in configuration and administrative behaviour. The problem with Quagga is that it is severely complex to do this.

A.3 NOX: Network Operating System

The NOX[5] is a project that began at Stanford University. They mainly attempt to improve network management and audit tools. Their concept of a 'network Operating System' is a little different. They use the term to denote systems that provide an execution environment for programmatic control over an entire network. Today, network is managed using very low-level tasks which require the network administrator to work with IP addresses and MAC addresses of every node, and be aware of what services is being provided by which node. NOX is an operating system which promises to provide an abstraction layer, much like traditional operating systems help provide control over a single system.

Their design requires centralization. They have developed applications that help deal with packet classification, policy-based filtering, routing and other network services like DNS. To achieve this, they use a central database having all the network information, and a single network view. Despite needing a centralized programming model, they vouch for network scalability and have spent considerable amount of time in improving performance and scalability. As a result, they have achieved creating management applications that work with high-level names as against low-level IP and MAC addresses.

While they may have achieved what they promised, it is a complex design, and the same can be achieved with much more ease and flexibility using our /net filesystem. Because we can mount the /net of any system onto our own, the same management applications can be used in a distributed fashion. In /net design, there is no requirement of a central database. Moreover, the /net filesystem is ideal for ISPs as you can grant temporary access to your /net, and the network administrator can mount the /net onto any other system in the network to carry out management tasks. Network transparency results in regular user-space tools and applications be useful for complex network management tasks and you dont need a central Network Management server in this design. This gives a sense of fine-grained control balanced with sufficient level of abstraction. You can compare NOX Vs /net to be something like monarchy Vs parliamentary form of government.