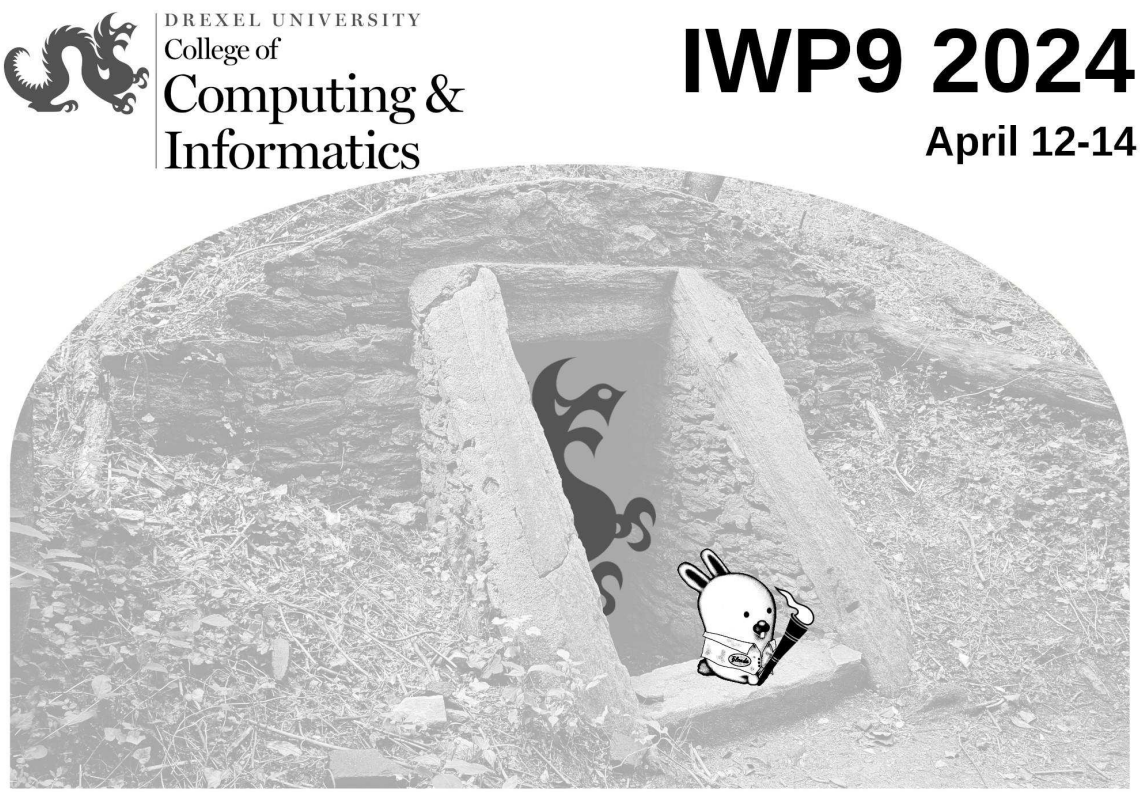
Proceedings of the 10th International Workshop on Plan 9



Sponsored by the Plan 9 Foundation

Proceedings of the 10th International Workshop on Plan 9

April 12–14, 2024

Drexel University

College of Computing and Informatics

Philadelphia, PA, USA

10th International Workshop on Plan 9

April 12–14, 2024

Hosted by: Drexel University, College of Computing and Informatics, Philadelphia, PA, USA

Organizing Committee:

Ori Bernstein, 9front
Michael Engel, University of Bamberg
Daniel Maslowski, Oreboot
Ron Minnich, Samsung System Architecture Lab
Brian L. Stuart, Drexel University
Fariborz Tavakkolian, 9Netics

Program Committee:

Brian L. Stuart, Drexel University, Chair
Ori Bernstein, 9front
Michael Engel, University of Bamberg
John Floren, Gravwell
Daniel Maslowski, Oreboot
Ron Minnich, Samsung System Architecture Lab

Table of Contents

Papers:

Works in Progress:

A Git for 9

Ori Bernstein
ori@eigenstate.org

ABSTRACT

Plan 9 is a non-posix system. Upstream git has been ported, but feels distinctly mismatched in a plan 9 environment. Even in its native environment, the user experience leaves much to be desired. Here, we describe our own take on the Git version control system: An implementation of git that is sufficiently compatible with the rest of the Unix world to interoperate, but with a typical Plan 9 view. This software provides both a client and server, and has been used to host 9front development on 9front for the last 2.5 years.

1. What is Git?

Git is a version control system. It is used for tracking changes to a set of files. It manages commits, file hierarchies, and file contents, and the forking and merging that happens when people work in parallel on a single project.

2. A History of Plan 9 Source Control

Plan 9 was one of the early systems that provided pervasive versioning. At Bell Labs, the file systems used were among the first where daily dumps, or snapshots, of file system contents were taken. In addition to While no explicit commits were made, daily snapshots of the entire state of the source tree were available for diffing against, distributing between users, bisecting bugs, auditing, and even replicating. While primitive to modern eyes, this situation got the team at Bell Labs much of the benefits of modern version control, at almost none of the cost.

In the meanwhile, the rest of the world developed tools like RCS, SCCS, CVS, SVN, Bzr, Monotone, Mercurial, and Git. These tools are relatively large, complex, and to a large degree were ignored by the team at Bell Labs.

When 9front forked Plan 9, people stopped being in the same room, same network, or same timezone. Tools for managing disparate source trees became more useful, and more important. Accordingly, 9front imported a copy of mercurial, along with Python. This served us well for the better part of a decade. However, the world moved on, and our version of both python and mercurial gradually drifted more and more out of date. Updating them became a good deal of work.

Outside of the Plan 9 universe, the world had also moved on. Git won the VCS wars, so any code outside of the Plan 9 universe, and even a significant portion of the code within it, was hosted on git.

When Git for Plan 9 was initially started the goal was not to replace mercurial, but to allow access to code hosted on popular hosting providers such as github. The author of this paper was maintaining code that primarily ran on Unix, but which he wished to develop and support on Plan 9. The lack of a git implementation made this painful, and solving this was therefore the initial goal.

3. What's In a Repo?

There are many tutorials and reams of documentation on the nitty gritty details of how git stores files in repositories. This document will not go over the storage in full. Instead, it will give a rough overview, in sufficient detail to put color on how the rest of git is implemented.

Git is a distributed, offline capable version control system. Therefore, every git clone contains a full copy of the repository in the *.git* directory. The repository is composed of several components: An object store, the index, and the branch references, or *refs* for short. In our implementation of git, the index format is changed. Additionally, the *.git* repository also contains an empty directory named *fs*. This directory is used as a mount point for *git/fs*.

To make this more concrete, a git repository may look something like this:

```
+-- objects
|   +-- 3f5d...c9283
|   +-- ae7d...3073f
|   +-- pack/
|       +--1fd30...3cfb3.pack
|       +--1fd30...3cfb3.idx
+-- refs/
|   +-- heads
|   |   +-- front
|   +-- remotes
|       +-- remotes
|           +-- front
+-- HEAD
+-- INDEX9
```

The objects/ directory is where the contents of the repository are stored. Each file, directory, commit, and tag gets its own object within the *objects* directory. The file name within the *objects* directory is the hash of the compressed data.

The *objects/pack/* directory contain pack files, which increase the efficiency of the object store. Pack files aggregate a collection of objects up into a single file, compressing them and deduplicating them against each other. They are paired with an index file, which allows rapidly seeking to objects within the pack.

The *refs/* directory contains a set of files containing commit pointers. These commit pointers are files containing a single commit hash on a single line.

The HEAD file is the final layer of indirection for commits. It contains the name of the current checked out branch in the *refs/* directory.

Finally, the repository contains the *INDEX9* file. This file is the only place that the Plan 9 git repository format diverges from the Torvalds git repository format. This file contains a list of tracked, newly added, and newly removed files. The index is used by *git/commit* to decide what files to add and remove from the subsequent commit. Unlike Torvalds git, the checked out files are used to update the commit.

Git9 is built around six core binaries. All other commands are present to wrap these, and provide a convenient user interface. These are the core binaries:

git/get: Negotiates a and downloads minimal set of data to sync a local respository from a remote one.

git/send: Negotiates and transfers a minimal set of data to a sync a local repository to a local one.
git/serve: Negotiates and transfers a minimal set of data to serve a git client.

git/save: Updates the contents of a repository, creating a new commit object and updated file tree.

git/query: Implements a small query language to inspect the commit graph.

git/fs: Serves repository history as a file system

There first three are designed as one would design them on Unix, and indeed have equivalent commands in Torvalds git. Most of the remaining commands are written as rc scripts. They lean heavily on git/fs to make them possible. Some of the remaining ones, like *git/walk*, are in C, because the performance overhead of exec.

Git/get is the core of commands such as git/clone and git/pull. For a provided set of branches, it downloads the data that is not currently in the repository. It prints the branch names and hashes, allowing a wrapper such as git/pull to update the branches.

Git/send is the inverse of git/fetch. It finds the commits in the local repository which are not present on the remote repository, and sends the minimal amount of information to update that remote repository.

git/save creates a new commit from a given list of files
a message, and a parent commit. The commit info is prepared by *git/commit*, *git/import*, or other similar scripts..

Git/query allows asking questions about the commit graph. These queries are provided via a separate binary rather than a file system, The query language is vaguely inspired by mercurial revsets, but stripped down and converted to a mostly postfix form for ease of parsing.

Where git9 differs most strongly from Torvalds git is the addition of git/fs.

Git/fs serves a file system interface to a git repository in the current directory. This file system provides a read-only view of the repository contents. By default, it is mounted on /mnt/git. It does not cache mutable data, so any changes to the git repository will immediately be reflected in git/fs.

The existence of a file system interface is extremely powerful, allowing scripts that operate on historical data to be written with ease. For example, git diff is not strictly necessary. The same effect could be achieved with plain old diff(1), though with clunky file paths. The git/diff shipped with git9 exists simply for the purpose of shortening the paths typed.

The file system looks something like:

```
/mnt/git
  +-- ctl
  +-- HEAD
  |  +-- tree
  |  |  +--files
  |  |  +--in
  |  |  +--head
  |  |
  |  +-- hash
  |  +-- msg
  |  +-- parent
  |
  +-- branch
  |    |
  |    +-- heads
  |    |    +-- master
  |    |      +-- [commit files, see HEAD]
  |    +-- remotes
  |       +-- origin
  |           +-- master
  |              +-- [commit files, see HEAD]
  +-- object
  | # blob contents
  +-- 00051fd3f066e8c05ae7d3cf61ee363073b9535f
  +-- 00051fd3f066e8c05ae7d3cf61ee363073b9535c
  +-- [tree contents, see HEAD/tree]
  +-- 3f5dbc97ae6caba9928843ec65fb3089b96c9283
  +-- [commit files, see HEAD]
```

4. Git from a File System Point of View

In Plan 9, everything is done as a file system. This is because it eables a tool based approach. Scripts and reusable utilities that can access the exposed names replace data-structure and file-format specific code. A good example of the benefits of this approach can be seen in git/merge. The core of git/merge is included below:

```
fn merge{
        ourbr=$gitfs/object/$1/tree
        basebr=$gitfs/object/$2/tree
        theirbr=$gitfs/object/$3/tree

        all='$nl{{git/query -c $1 $2; git/query -c $2 $3} | sed 's/^..//' | sort | uniq}
        for(f in $all){
                ours=$ourbr/$f
                base=$basebr/$f
                theirs=$theirbr/$f
                merge3 ./$f $ours $base $theirs
        }
}

gitup

flagfmt=''; args='theirs'
eval '''{aux/getflags $*} || exec aux/usage

if(! ~ $#* 1)
        exec aux/usage

theirs='{git/query $1}
ours='{git/query HEAD}
base='{git/query $theirs ^ ' ' ^ $ours ^ '@'}

if(~ $base $theirs)
        die 'nothing to merge, doofus'
if(! git/walk -q)
        die 'dirty work tree, refusing to merge'
if(~ $base $ours){
        >[1=2] echo 'fast forwarding...'
        echo $theirs > .git/refs/'{git/branch}
        git/revert .
        exit ''
}
echo $ours >> .git/merge-parents
echo $theirs >> .git/merge-parents
```

Note that the *git/merge* command is largely a wrapper around the *merge3* command.

There's more code involved in deciding what to merge than there is in the merging itself. Walking through it in sections, we begin with initializing the binary.

The script begins with some relatively uninteresting code. It loads a few utility functions, and parses flags. One of these utility functions is *gitup*. Gitup finds the base of the repository, and changes into it, so that commands which expect a *.git* directory to be present work.

Next, we need to figure out what the commits we're merging are. For a 3-way merge, we need the last common commit, and the two heads we're bringing together. A few *git/query*s make short work of that.

```
theirs='{git/query $1}
ours='{git/query HEAD}
base='{git/query $theirs ^ ' ' ^ $ours ^ '@'}
```

The only interesting thing to note is the @ operator in the *git/query* command line. It finds the least common ancestor of two commits. For example, if you had this commit graph:

```
       o---o---T <-- theirs
      /
-----L     o---o
        /
        o---o---O <--ours
```

Then the @ operator would walk back to the point at which the two branches diverged, marked with *L*

The next chunk of code ensures that we're in good shape to merge. If the least common ancestor of our curernt commit is the same as their commit, that means that the branches never diverged. Moreover, it means that we already have their commit.

```
--o--o--T--o--o--O <--ours
        ^
        |
      theirs
```

If we are in this situation, there's no need to merge.

It also checks that we don't have unmerged work in the tree, so we don't make a mess of things that are in progress.

```
if(~ $base $theirs)
        die 'nothing to merge, doofus'
if(! git/walk -q)
        die 'dirty work tree, refusing to merge'
```

If, on the other hand, the base commit is the same as our commit, we can simply fast forward. The commit graph for that looks like:

```
--o--o--O--o--o--T <--theirs
        ^
        |
      ours
```

And we can simply move the *ours*pointer forward to point at their branch.

After the setup and checks are complete, we're ready to merge. For this, we simply invoke *ape/diff3* on the files, pairwise, to do the merge. We first figure out which files we want to merge:

```
fn merge{
        ourbr=$1/tree
        basebr=$2/tree
        theirbr=$3/tree
        all='{walk -f $ourbr $basebr $theirbr |                 subst -g '^('$ourbr'|'$basebr'|'$
```

In this snippet of code, we walk down the file trees to get the list of all files in all three branches. This is the list of files we want to invoke merge on. Since some of the files may exist in one branch but not another, it's necessary to substitute the ones that don't exist with happens*/dev/null*.That snippet:

```
for(f in $all){
        ours=$ourbr/$f
        base=$basebr/$f
        theirs=$theirbr/$f
        if(! test -f $ourbr/$f)
                ours=/dev/null
        if(! test -f $basebr/$f)
                base=/dev/null
        if(! test -f $theirbr/$f)
                theirs=/dev/null
```

Finally, the important bit happens: the merge:

```
        if(! ape/diff3 -m $ours $base $theirs > $f)
                echo merge needed: $f
```

We ensure the files are tracked or removed, as needed, and then we're done.

```
        if(test -f $f)
                git/add $f
        if not
                git/rm $f
}
```

Most of the other tools in git9 are written as shell scripts, following similar principles. Examples include <code>git/clone</code>, *git/log*, *git/commit*, and *git/revert*.

5. Production Experience

This implementation of git has been used to manage the 9front sources since June 2021. The transition uncovered some bugs around permission handling in a multi-user system. Other than that, the migration to git was largely uneventful.

When serving git for plan 9, we integrate deeply into the Plan 9 environment. The git/serve binary is designed to run under *listen*(1). When running unauthenticated, the git binary is launched with a script such as:

```
#!/bin/rc
exec git/serve -r/usr/git
```

A new protocol name, *hjgit://* has been added to the Plan 9 git for authenticated communication. Authentication is done using factotum, using tlsclient and tlssrv to establish a tunnel. This is served using the following script:

```
#!/bin/rc
exec tlssrv -a /bin/git/serve -wr/usr/git
```

This setup is used on both Shithub and the 9front git host. It has been in use since June 2021, and is currently serving between 3 and 4 gigabytes of traffic across the web interface as well as the git service. Service has been more or less problem free, though it exposed one issue in the mount interface. The mount id would wrap to negative, causing Plan 9 to detect a successful mount as an error.

6. Comparison to Torvalds Git

Because of the tool based approach, the Plan 9 implementation of git is far simpler, shorter, and more amenable to changes. However, it misses several features, such as *git/blame* due to the difficulty of accessing internal state for tools such as *diff*.

The Plan 9 implementation of Git sits, therefore, at 9662 lines of code at the time of this writing, excluding tests. The Torvalds implementation of Git sits at 315,306 lines of code, excluding tests, guis, and git web viewers.

7. References

[1] Git authors ‘‘Git documentation,’’ *https://github.com/git/git/tree/f86de088f8c32377fbd681cc481e7128af83ce2f/Documentation,* Updated Wed, May 31 2023

[2] John Wiegley, ‘‘Git from the Bottom Up’’ *https://jwiegley.github.io/git-from-the-bottom-up/* Updated April 1 2023

[3] Konstantin Serebryany, Timur Iskhodzhanov ‘‘ThreadSanitizer - data race detection in practice’’ *Workshop on Binary Instrumentation and Applications*, 2009

Θfs: An Experimental Generalized Storage Server

Brian L. Stuart

ABSTRACT

As part of an ongoing research program studying techniques and architectures for data storage, we have investigated the possibility of unifying file, object, and block access to data stores. To date, the result of this investigation is a generalized storage server that provides all three methods of access. Using full path name hashing and extensible metadata, this server provides a highly flexible platform that can form the basis of a production server or of further storage research. Here, we discuss the background behind the server, the basic concepts of its design, and the directions for continued R&D planned for it.

1. Introduction

In the very earliest days of computing, data storage was primarily in the form of punched cards and magnetic tapes. A particular set of data was identified as much by what tape contained it, or what box contained the cards as any other identification. It was the advent of the disk drive that shaped our modern view of data storage. In particular, the disk had two characteristics that weren't found on tapes or cards: fixed relatively large blocks and random access. These two characteristics led to our modern picture of a particular set of data occupying one or more blocks of storage space and being labeled with a file name.

Very quickly, a model of data access developed where an application speaks to a file system component in terms of file names and offsets within the file. The file system code then talks to a block device driver to read and write individual blocks. In [1], we note that the job of the file system, then, is two-fold. First, it must manage the set of names for the files, and second, it must mange the allocation of blocks to files. These relationships are illustrated in Figure 1.

Beginning with research at Xerox PARC and widely deployed by workstation manufacturers like Sun and Apollo, the application often ran on a different machine than the file system. So by means of some network file access protocol, the application makes a request of the file system (server) which accesses disks in terms of blocks just as before. Over the years, numerous such protocols were developed with the most relevant to Plan 9 being NFS, CIFS (née SMB), and 9P. This approach has gained the name Network Attached Storage (NAS) in the trade.

Later work explored the possibilities of putting the network connection between the file system and the block devices, rather than between the application and the file system. Several protocols also grew up around this type of storage organization. Among them are FiberChannel, iSCSI, and AoE. A common trade term for a network system structured in this way is Storage Area Network (SAN). Figure 2 illustrates both file-oriented and block-oriented network accessible storage.

Recently, a third alternative for where to put the network connection has emerged [2]. If we put the connection in the middle of the traditional file system structure between the name management and the block management, we get what is often called an object

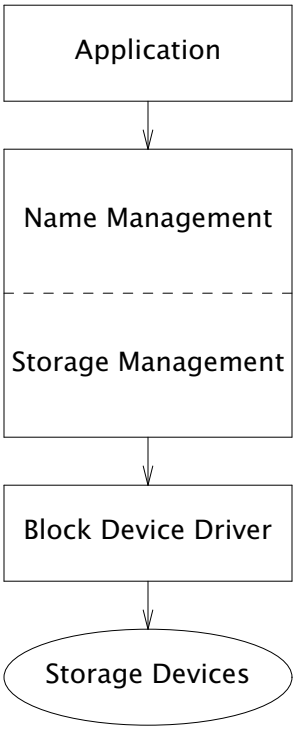


Figure 1: Storage Management Component Relationships

storage system. This structure is illustrated in Figure 3. For completeness, we should note that the concept of an object store does not necessarily have to be implemented over a network. However, it is the use of networked storage appliances as object storage devices that interests us here.

As with any set of successful technological alternatives, each of these storage architectures has strengths and weaknesses relative to the others. This means that some applications are better suited to communicating in terms of files, some in terms of blocks, and some in terms of objects. Therefore, we do not attempt to argue for one approach as universal in this paper. Instead, we examine the design of a storage server that provides all three means of access in such a way that they all coexist and share the same raw storage space.

2. Access Characteristics

To get a better sense of how a unified storage system should be structured, we need to look at how each type of storage access is specified.

2.1. Blocks

We begin by looking at how block I/O operations are specified. Normally, there are one or more collections of blocks identified by logical unit numbers (LUNs). (While semantically problematic, we often also refer to the actual storage space as a LUN.) The number of LUNs is relatively small, and the size of each is relatively large. Conceptually, each LUN is like a single disk drive. Once established, it generally does not change size, and therefore cannot grow as a result of writes. In practice, the set of blocks that make up a LUN may be spread across several drives, or sometimes a subset of a drive. From the perspective of a client using a block store, though, the whole of a LUN has characteristics much like those of a disk drive.

I/O operations on blocks are carried out in units of blocks, typically the size of historical

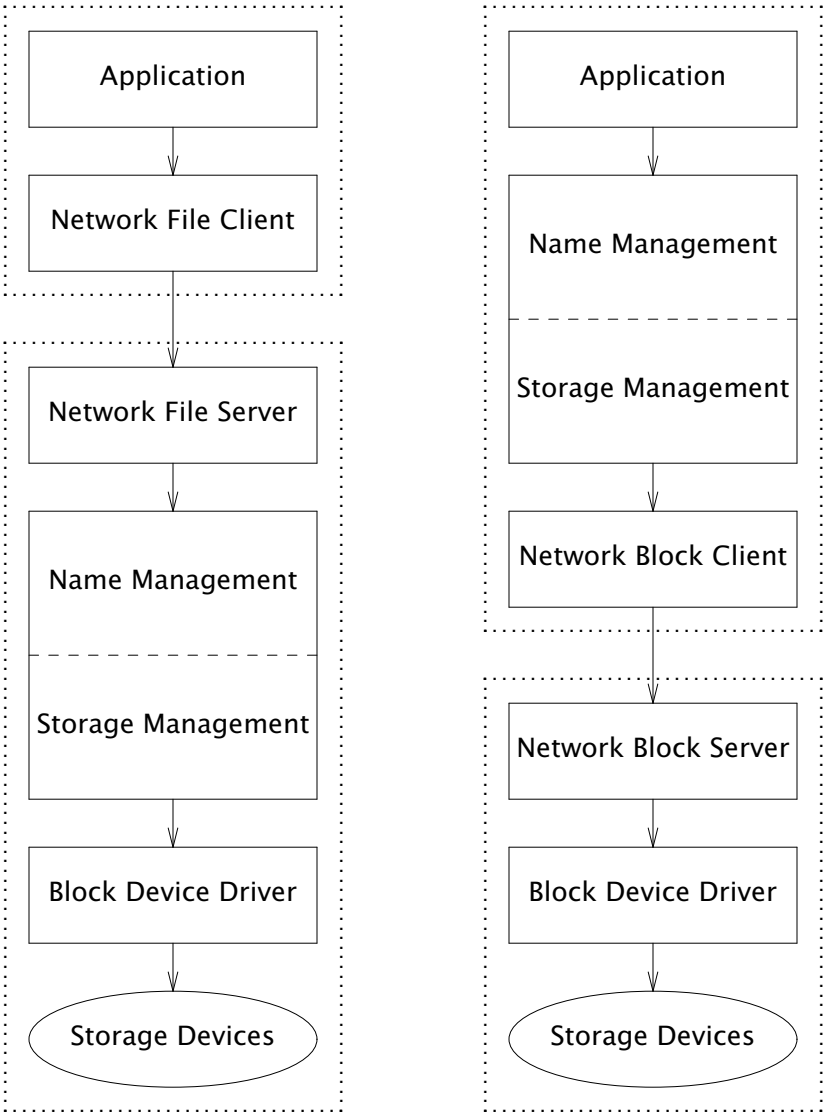


Figure 2: Network Storage Architectures: Network Attached Storage (left) and Storage Area Network (right)

disk sectors, 512 bytes. Often reads and writes will involve several blocks, but incomplete blocks are never transferred. Thus to specify a read or a write in a block store, we need a LUN, a starting block number, and a count of the number of blocks to read or write. Taking AoE as an example, LUNs are identified by a 24-bit target number, block numbers by a 48-bit LBA, and block counts by an 8-bit sector count.

One last issue of block storage is the allocation policy. Are blocks allocated administratively before placing the block store in use, or are they allocated on demand as they are written? Following the historical connection between LUNs and physical storage devices, block storage systems originally operated according to the administrative allocation policy. As the option of demand allocation became available, it was usually described with the trade term thin provisioning. By contrast, the traditional administrative allocation came to be called thick provisioning. In most cases, dynamic allocation is implemented much like sparse files in a traditional UNIX file system where only the written blocks are allocated and not intervening unwritten blocks.

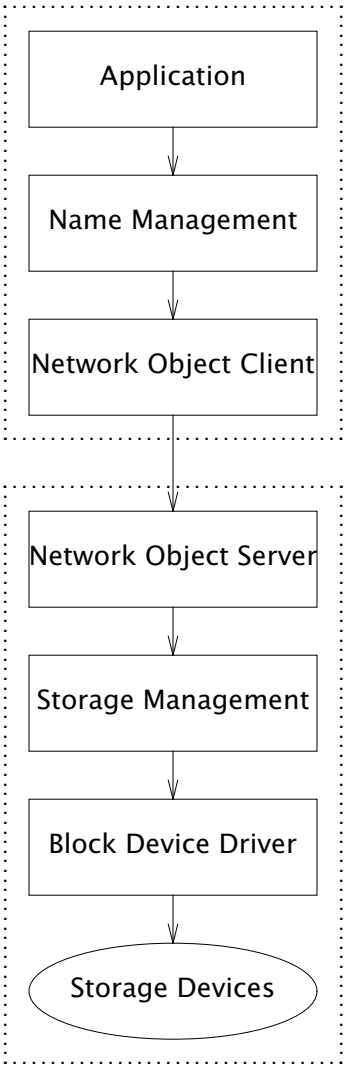


Figure 3: Network Object Storage Architecture

2.2. Objects

Similar to the LUNs in block storage, objects are generally identified by a numerical designation. In the case of the SCSI object storage device specifications, this designation is a pair of 64-bit numbers described as the partition ID (PID) and the object ID (OID). As suggested by the large number of identification bits, there are generally many more objects in an object store than LUNs in a block store. Unlike block stores, I/O in object stores is specified in terms of the byte offset and byte count, rather than block number and block count. Finally, the allocation policy of objects is usually dynamic, possibly with administratively established upper limits.

2.3. Files

Files are, of course, the most familiar of the three storage paradigms we are considering. They share much in common with objects. Accesses are specified in terms of byte offsets and byte counts. Allocation is typically on-demand allowing files to grow. The primary difference between files and either objects or blocks is the means of identifying a file. Rather than a fixed numeric identification, files are identified with strings that have few (if any) constraints. The names are also usually organized hierarchically.

Although object IDs can be thought of as degenerate case of file names, the SCSI identification scheme only provides for two levels of hierarchy (partition and object).

3. The Naming/Allocation Management Boundary

At this point, we must admit to some degree of hand-waving and over simplification. In particular, the boundary between the naming functions and the space management functions of the file system implementation is a fuzzy one. We've described the hierarchical relationship among files as being a property of the naming. In practice, that hierarchy is also usually represented in the on-disk data structures. As discussed later here, however, that coupling between what are otherwise two independent functionalities is unnecessary, and removing it opens up a number of interesting possibilities.

4. Design Motivation

We can see from the similarities between files and objects that they practically beg for a unified implementation. In fact, in one of the early presentations on object storage, the question was asked whether objects should be implemented on top of files, or files on top of objects.

The first approach borders on trivially easy. All one need do is create files whose names are the object numbers expressed in a human-readable notation. For example, if we create one directory for each of the partitions in the SCSI object storage device specification, and one file in each of those directories for each object in that partition, we can name both with a hexadecimal representation of the 64-bit numbers. As long as the file system allows names of at least 16 characters, then this kind of organization can be used without any modification of the file system design. The reason this approach is not often used is that most file systems are designed around the idea that a typical directory contains tens to hundreds of files. However, we can easily have millions of objects in a partition.

More commonly developed have been approaches where file systems are implemented on top of object stores. In these implementations, each file and directory is contained in an object. Because SCSI OSD defines a rich set of object metadata, the traditional UNIX i-node metadata need not be managed directly by the file system implementation. Exofs is an example of such a file system in Linux.

Regardless of what approach we take to connecting files and objects, it is clear that the key mechanism is a mapping between names and numeric IDs. With that perspective, it is clear that the infrastructure for a unified file and object system has been around for a long time. For example, the UNIX file systems of the 1970s mapped names to i-numbers that uniquely identified files. A little more recently, the 9P protocol identifies files by an identifier called a QID of which a 64-bit path number uniquely identifies a file.

One might suggest that the best approach to this unification is to replace the 64-bit path in the QID with the pair of 64-bit IDs in the SCSI OSD specification. However, our results show that translating both name and object IDs to an independent numeric identifier (such as a QID path) is more conducive to coexistence of files and objects and also provides a mechanism for integrating block access. To understand why, we first observe that the unique IDs for LUNs are usually administratively assigned, unique IDs for files are usually assigned by the file system, and unique IDs for objects are usually assigned by the client. So the benefits arise because mapping all client and administratively assigned identifications to an internally managed space of IDs prevents collisions between ID assignments.

For the remainder of this discussion of the design and implementation of our prototype generalized data store, Figure 4 illustrates its major components and overall structure.

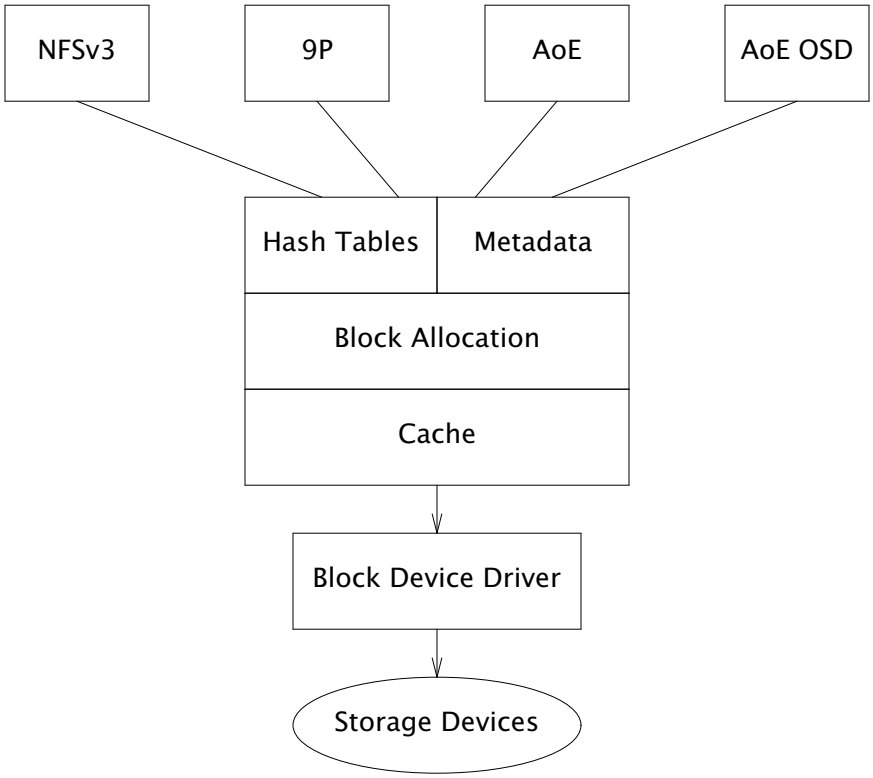


Figure 4: θfs Architecture

5. Name and ID Management Design

For the prototype we report here, we have opted to use the 9P QID path for the internally generated unique IDs. The high-order four bits of the QID path are a type specifier. This is used to allow multiple allocation strategies to coexist. The next 12 bits are reserved. Their planned use is as a node ID for future research in distributed storage. Finally, the remaining 48 bits of the QID path are assigned according to strategies suitable for different access mechanisms.

For mapping to these IDs, we use a hash table. Unlike other file systems using hash tables, we hash the full path name of each file to find the bucket where we find the file's ID. Similarly, for objects, we have a textual representation of the partition and object IDs. Collisions between file names and textual object IDs are prevented by constructing all path names beginning with a slash (/). The QID paths of both files and objects use a type field of 0. Block accesses, on the other hand, use a type field of 2. For the LUNs accessed using a block paradigm, the 48-bit field contains the AoE target number in the lower 24 bits.

With this arrangement, we can manage a storage space shared by files, objects, and block LUNs. Although not implemented in this prototype, this arrangement also allows the file server interface to provide access to both the objects and the block LUNs through special name spaces. For simplicity, we use the same allocation mechanism for all three storage types, resulting in naturally thinly provisioned LUNs.

6. Metadata Design

Most 9P servers are implemented in such a way that the names map directly to metadata structures that contain the QID. However, because we want to support access directly via the QID path to storage that is potentially unnamed, we map names only to QID

paths, and separately map QID paths to metadata. Unlike the i-numbers in a traditional UNIX file system, the QID path numbers come from a space too large to use as simple indices. Therefore, we use a second hash table that maps QID paths to metadata lists.

Each of the different types of storage entities requires a different set of metadata. For example, an AoE LUN needs to include the major and minor AoE target numbers, but doesn't need any information representing any kind of hierarchical arrangement among the LUNs. An object needs to include the PID and OID. Regular objects also need a mapping to blocks that comprise the object, but partition objects need a way of identifying those objects that are contained in them. Similarly, files served by 9P require owner and group identifications, a mode, access and modification times, and pointers that describe the hierarchical structure. Regular files need block mappings, but directories don't. Furthermore a full implementation of the SCSI OSD specification includes a substantial amount of additional metadata.

Even among different remote file access protocols the set of metadata varies. For example, for files accessed through 9P, we need textual owner and group names, and QIDs that contain a path, a version, and a type, none of which are found in NFSv3. Conversely, for NFS, we need to support numeric user and group IDs, symbolic links, device nodes, named pipes, etc, all unknown in 9P.

For these reasons, we have opted for an extensible metadata design. The basic idea is similar to that found in the VMS file system [1,3] and later in NTFS [1,4]. Each metadatum consists of a next pointer, a type, a name, and a value as illustrated in Figure 5. In the present prototype, the value field is a fixed eight-byte value. In each metadatum, this value is used in one of three ways. For integer valued metadata, the integers are stored as 64-bit numbers. For short strings (≦7 bytes), the string is stored directly in the eight-byte value. If the datum value is a longer string or an arbitrary byte sequence, the eight-byte value is an offset to where in the storage space the real value is stored. All of the metadata entries for a given storage entity are arranged in a linked list. Free metadata entries are also organized in a linked list.

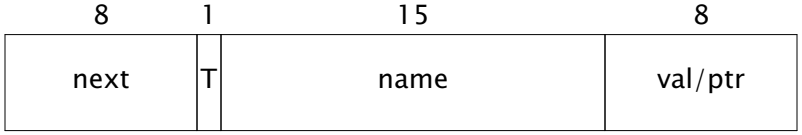


Figure 5: θfs Metadata Structure

Block mapping is similar to that found in the i-nodes of traditional UNIX file systems with a few simplifying differences. Each metadata block that identifies real storage space contains one datum that specifies the root block of a tree of block pointers that is either zero, one, two, or three levels deep. For the zero depth case, a single metadatum specifies one data block for small files. Entities created with a known size, such as LUNs, are created pointing to a tree of the appropriate depth. Entities created with an initial size of zero are created with a tree of depth zero, but with no data block allocated. If such a file or object grows too large to be represented by a single block of block pointers, a new block is allocated to be the root of a tree of depth two, and it's first element points to the existing block of block pointers. The metadatum that had pointed to the index block is replaced by one pointing to the newly allocated root. Similarly, if an entity grows too large for a two-level tree, a third level root is allocated and the metadata updated to point to it.

7. Design Implications

Managing names in terms of full paths has turned out to naturally break the coupling between name management and space management. There is no longer any need for a dedicated directory data structure on the storage media. In particular, we no longer have a unique root directory structure that is typically located by a pointer in a superblock. At first glance, this would simply be a mildly interesting point, hardly worth mentioning. However, it turns out there are a number of useful properties that fall out essentially for free from this simple design element.

First, the coupling created by on-disk directory structures made the integration of objects conceptually awkward. In fact, most presentations of object storage redefine the function of the subsystems on either side of the point at which an object protocol is injected. However, by making the hierarchical relationships a function of name management alone, the integration of files and objects can be quite straightforward as illustrated in the preceding sections.

The second interesting capability that falls out of this simplification arises from the lack of a unique root identified in a superblock. It becomes almost trivial to support multiple independent file systems, each with its own root directory. One simple way to make use of this capability is to let each file tree be named by a ''root'' which begins with the slash (/) character, and as we construct path names, we precede each path component by a slash as a separator. Thus, the sys directory of an unnamed file tree would be internally known by the string ''//sys'' and by ''/fs1/sys'' for a file tree named fs1. The attach message of the 9P protocol makes provision for clients to specify a named file tree when establishing an initial connection to the server. Similarly, the mount RPC protocol associated with NFS has provision for a string identified as a directory path. Normally, it is used to mount a subtree of a single file system served by a file server. However, if we generalize the string to simply specify the root of some file tree, then we can use it in the same way as the file tree specifier of the 9P protocol and select among independent file systems.

Finally, in reading this discussion, one can reasonably conclude that this is just a complicated way of saying there's a higher-level root that contains all the object, all the LUNs, and all the independent file systems. Indeed, this is an excellent way to view the overall structure. Viewing the structure from the perspective of an implicit root opens the possibility of crossing access method boundaries. For example, if we define a file system called ''/object'' then we can make all object accessible via 9P or NFS.

8. Experimental Object Extension to AoE

To provide a basis for development and testing of network object storage support, we have chosen to eschew implementing iSCSI in favor of adding experimental object storage features to AoE. These features are modeled on (but considerably simplified from) the SCSI OSD specification. Several elements of the SCSI OSD model have been omitted in their entirety for this research. First, there is no quota mechanism implemented. However, implementing one has not in any way been precluded by the prototype design. Second, no provision has been made for either the optional security manager or the optional capability-based policy manager. Third, setting or querying attributes (metadata) are not allowed to be piggy-backed on other operations as is allowed in the SCSI OSD specification. Fourth, the optional collection object type has not been implemented.

With those caveats established, we now describe the protocol extension implemented in the present prototype. The AoE command value 5 is used to issue an OSD command. For OSD commands, the AoE Arg field includes:

1. A one-byte command field.
2. A one-byte flags field.

3. A two-byte length field.
4. An eight-byte partition ID field.
5. An eight-byte object ID field (omitted for some OSD commands).
6. An eight-byte address field (omitted for some OSD commands).
7. A variable length data field for those commands or responses that require it.

Both requests and responses follow this format, as illustrated in Figure 6, except that for responses, the PID, OID, and address are omitted.

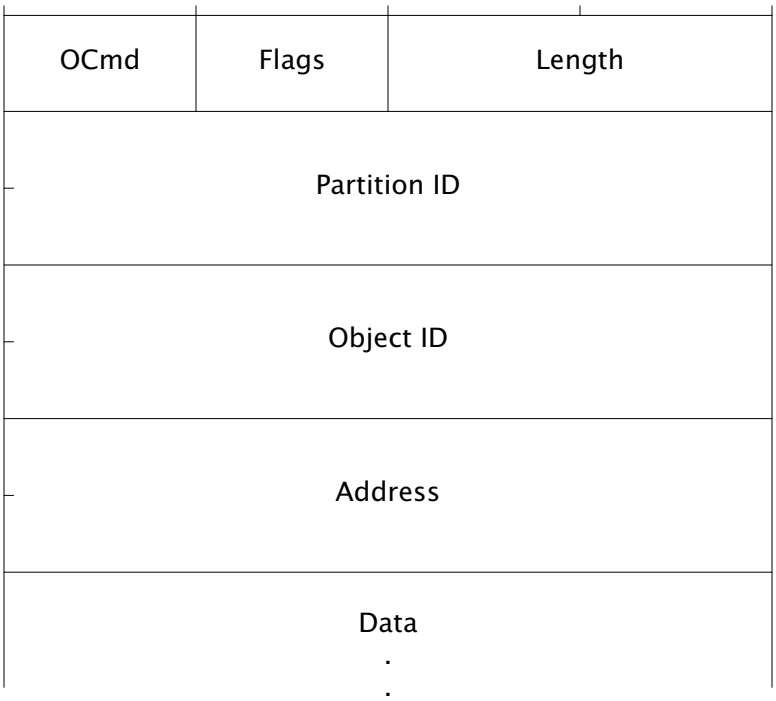


Figure 6: AoE Object Store Extension

The OSD commands supported are a subset of the SCSI OSD commands, and the value of the command field is the low-order byte of the service action value in the SCSI OSD protocol for specifying the same operation.

0x01: **Format OSD.** This command creates the root partition object as defined for SCSI OSD. All fields after the length are omitted for this command. (However, because of the minimum Ethernet frame size, some fields for this and other commands are sent anyway.)

0x02: **Create.** The create command is used to create a new object within a partition. A metadata list is allocated for the object and populated with a default set of metadata, but no data blocks are allocated on create. The PID and OID are specified, but the address and data fields are omitted.

0x03: **List.** Only the PID is specified as part of the list command. If the PID is zero (i.e. the root partition), the target sends back a list of the IDs of all extant partitions. Non-zero values of PID will elicit a list of the objects in the referenced partition. The length field of the response gives the number of bytes in the list of IDs returned.

0x05: **Read.** For reading from an object, the PID/OID specify the object to be read, the address field gives the offset from the beginning of the object where the read should start and the length field gives the number of bytes requested. In the

response, the length field gives the number of bytes successfully read and the number in the data field.

0x06: **Write.** Similarly the write command requests that the number of bytes specified in the length from the data field be written starting at the offset in the address field to the object in the PID/OID fields. On response, the length field gives the number of bytes successfully written.

0x07: **Append.** The append command works like the write command, except that the address field is omitted and the offset is implicitly the current size of the object.

0x08: **Flush.** Upon receipt of a flush command, the target writes all dirty in-memory buffers to the disk.

0x0a: **Remove.** The object specified the the OID/PID fields is removed by this command. Its data blocks and metadata are freed.

0x0b: **Create Partition.** This command is similar to the object creation command except that a partition object is created. Thus only the PID is specified in the request.

0x0c: **Remove Partition.** This command removes an empty partition freeing its metadata list. If there are objects in the partition, it is an error.

0x0e: **Get Attributes.** Metadata for the object specified by the PID/OID (partitions being specified by on OID of zero) are queried by this command. The data field contains a list of the null-terminated names of the items being requested. Data in the response message is a list of the query results where each item is composed of its null-terminated name, a single character type specifier and a value. The types are h: short, l: long, v: vlong, s: string, and b: blob. The integral types are two, four, and eight bytes, respectively. However, in the current prototype, all integer types are handled as 64-bit vlongs. Strings are null-terminated and blobs are prefixed by a two-byte length field. All numeric values are given in network byte order.

0x0f: **Set Attributes.** Setting metadata values is handled as the converse of getting attributes. The data field contains the requested values to set in the same format as returned by the get attributes request.

Two flags are currently defined for the flag field. The high-order bit (0x80) is set to indicate a response message, and the next bit (0x40) is set to indicate an error. As suggested above, with the exception of the read request and the write and append responses, the length field always gives the number of bytes in the data field.

9. Implementation Details

Up to this point, we have described this work in pretty general and abstract terms. In this section, we begin filling in details as they apply to the present prototype. The prototype file/object/block store is called θfs. (Earlier iterations of this design were named according to earlier letters of the Greek alphabet. Ultimately, the intention is to name the final version ωfs, or perhaps Ωfs.) It is implemented in C on Plan 9 using the 9P and thread libraries. File services are provided by a 9P server and an NFS server. Block and object services are provided by an AoE target.

9.1. Sizes

Blocks in θfs are 2^{15} bytes (32KB), and block addresses are specified as 64-bit values. Thus a block containing block addresses will contain 2^{12} (4K) such addresses. A single superblock occupies the first block on the storage space. The next group of blocks are used for the two hash tables. Following the hash tables there is a variable sized free bit map. Variable sized regions for a metadata structure pool and a string/byte sequence pool follow the free bit map. The remainder of the storage space contains allocatable blocks for data and block indices.

As mentioned above, the blocks comprising a file, object, or LUN are mapped by a one, two, or three level tree of block pointers. For a single level of block pointers, the maximum size is $2^{12} \times 2^{15} = 2^{27}$ bytes (128MB). For two levels of block pointers, the maximum is $(2^{12})^2 \times 2^{15} = 2^{39}$ bytes (512GB). For three levels, the entity can be as large as $(2^{12})^3 \times 2^{15} = 2^{51}$ bytes (2PB). If a fourth level is added, the maximum size will be enough to support up to 63-bit offsets for object and file access as well as the full ATA 48-bit block numbering in AoE.

9.2. Hash Tables

Each hash table contains a prime number of buckets in the range of 1048573 to 268435399. The prime number chosen is one greater than or equal to the number of blocks in the store and close to a power of two.

Both hash tables are open implementations. For the path name to QID path table, non-zero bucket entries point to blocks containing all elements that hash to that same bucket. If a single block is not large enough to contain all of them, then the blocks form a linked list with the last eight bytes of the block containing the block number of the next in the list. Each element in the table is composed of an eight byte QID path, followed by a two byte character count, followed by a variable length path name. For the QID path to metadata block number mapping, bucket entries give the index of the first metadatum for the first entity hashing to that bucket. As mentioned earlier, the set of metadata for a given entity is organized as a linked list. One special metadataum in each list is called qhnext and gives the list head for the next entity that hashes to the same bucket.

9.3. 9P Service

The 9P server in θfs is generally pretty conventional. In a departure from the libthread and lib9p design, each instance of the srv loop runs not in an independent process, but in another thread with all being in the same process. This helps to eliminate a substantial amount of mutex locking that would otherwise be required.

9.4. AoE Support

The AoE target functionality was adapted from the implementation of vblade (an AoE implementation from Coraid). First, the mainline startup code for vblade was replaced with calls from the θfs startup code. This also involved minor changes to allow LUNs to be created and removed on the fly. Second, the parts of vblade that interacted with the files backing it were changed to make calls into the θfs code to make use of its cacheing and block allocation infrastructure. Third, the configuration information was moved from being stored in a reserved area at the beginning of the LUN file to the θfs extensible metadata. Finally, support was added for the experimental OSD additions to AoE including support for all the commands listed above.

9.5. NFS Support

The θfs prototype includes support for NFSv3. Version 3 of the protocol was chosen because it is nearly universally supported and because significant potential clients did not support Version 4 at the time of the prototype development.

Normally, we would implement NFS as a service layered on top of an existing file system or as a protocol translation to 9P. However, because one of the objectives of the current work is to demonstrate coexistence among disparate file systems, we wish to support symbolic links, named pipes, UNIX-domain sockets, and device nodes, which are concepts not present in 9P. The extensible metadata design makes this easy, but it requires that the NFS server have direct access to the metadata. In the present prototype, therefore, NFS is implemented as a server integrated directly into the file system implementation.

Because both the NFS and 9P servers use the same path name and QID hash tables, they provides two different protocol views of the exact same file system. File types supported in NFS but not 9P simply appear as empty files to clients viewing the file system through 9P.

There are two primary features of most NFS servers that are not provided in the current prototype. First, we do not provide support for hard links. The NFSv3 standard provides a mechanism for a server to indicate whether it supports each type of link, but clients that expect hard links to be supported are likely to be disappointed. Second, NFS implementations generally include a lock daemon to support distributed locking of files and regions of files. Lock daemon functionality has not yet been included in θfs.

9.6. Cacheing

The current cacheing mechanism is a pretty typical buffer cache with reference counting. Writes are handled in a write-back fashion with a process writing dirty blocks every 15 seconds. The cache keeps a nominal maximum of 4000 blocks in memory and blocks are reclaimed following an LRU policy. However, blocks that are dirty or that have non-zero reference counts are not reclaimed until the next opportunity. Therefore, it is possible for the number of blocks in the cache to sometimes exceed the nominal limit of 4000.

9.7. Snapshots

We support snapshots in θfs when it runs on storage served by devsnap, as reported in [5]. The multiple root mechanism supports a dump file system that is compatible with the traditional Plan 9 dump file system. The presence of the snap device is not necessary, however, and θfs can run directly on a disk partition.

9.8. Console

In addition to listening for TCP 9P connections, for NFS requests, and for AoE messages, θfs posts a file descriptor to /srv to provide a textual console interface. Connecting to it with con -C allows one to interact with θfs to get current status and to examine metadata. The current set of commands supported include:

allow: Effectively turn off all permissions checking. This also allows 9P wstat messages to change file ownership.

blockuse: Look up a block by number and print out how it is used.

checkalloc: Scan the file system to compare the free map to the actual block usage.

cstat: Print a summary of the current state of the cache. This includes the number of blocks in the cache, the number of dirty blocks, the number of cache misses, the recent hit rate, and a reference count histogram.

disallow: Turn on normal permissions checking and prohibit ownership changes by 9P wstat messages.

dumpusers: Print out all known UID information, both as known to 9P and to NFS.

fixfamilies: Clear out references to children or siblings where the QID doesn't map to any metadata.

fixpaths: Clears path hash table entries with no corresponding metadata.

halt: Shut down the file system cleanly.

help: Print a list of the valid commands.

hstat: Print a summary of hash table statistics including number of lookups, number of collisions, and maximum search depth.

inituid: Initialize the NFS UID table from the file //adm/nfs.

lcreate: Create a LUN given its AoE major and minor numbers and its size in sectors.

lls: Print a list of all LUNs giving their AoE target numbers and sizes.

lmeta: Given a LUN's AoE target number, print out the LUN's metadata.

lrm: Remove a LUN given its target number.

mpred: Locate the predecessor to a metadata item in the linked list.

mprint: Print the details of a given metadata item.

mstat: Print a summary of metadata management statistics.

newroot: Create a new file system root.

nfsdebug: Control the level of debugging of the NFS file server. An argument value of 0 turns off debugging. Values 1 and 2 select increasingly detailed debugging information.

p2q: Show the QID path for a given path name.

p9debug: Turn on or off 9P debugging. The argument is the value assigned to chatty9p.

phash: Print the hash bucket for the specified path.

pmeta: Print the metadata for the file whose path name is given.

q2m: Show the metadata entry number for a given QID path.

qmeta: Print the metadata for the entity identified by the given QID path.

recovermeta: Check the consistency of and garbage collect the metadata and rebuild the free list.

revert: Roll back to an earlier snapshot as managed by devsnap.

rmp: Remove a hash table path entry.

rootallow: In the NFS server, give UID 0 the same super user privileges that it traditionally has in UNIX.

rootdisallow: In the NFS server, treat UID 0 as an ordinary user for permissions checking.

setmeta: Set the integer or string value of a metadata item.

setmstruct: Manually overwrite a metadata structure.

setqhash: Set a QID to metadata mapping in the hash table.

snap: Take a snapshot through devsnap.

super: Print the contents of the superblock. This includes the magic number, the next sequential QID path to assign, the number of blocks, the number of free map blocks, the location of the free map, the status, the location of the first data block, the number of free blocks, the QID path of the first LUN and the locations and sizes of the metadata and string pool regions.

sync: Trigger a pass of the write-back process to flush dirty blocks to the disk.

Most of the same console functionality is available by way of the file /mnt/θfsctl. Reads from this file combine the results of the super command and all stat commands. Writes to the control file are treated the same as writes to the console with the commands allow, disallow, lcreate, lrm, rootallow, rootdisallow, and sync being supported.

10. Results

The θfs prototype has been self-hosting in the sense that editing and compilation of the code currently take place on copies of the files stored on a θfs file system accessed with 9P. Similarly, the editing and formatting of versions of this paper were carried out on a copy of the troff source file stored on θfs.

The implementation of θfs is about 10000 lines of code, of which approximately 1000 were taken from vblade, and about 2500 are the integrated NFS server implementation. A slightly earlier version of the θfs code can be found in contrib/blstuart/θfs.

As indicated, the most extensive testing and use of this storage system has been via the 9P interface. The AoE block interface has been exercised using both the Plan 9 and the Linux software initiators to the extent of creating a thinly provisioned LUN, building a file system on that LUN, and carrying out file system operations on that file system. It should be noted that the file system so-created is completely independent of that served by 9P and NFS, but coexists with and allocates blocks from the same pool as the one they serve. As there currently exist no AoE OSD clients, testing of the object interface has been limited to a small test client written specifically for this project. The file system served via NFS has been mounted on both MacOSX and Linux and used concurrently with 9P clients of the same file system.

Although θfs should be viewed as still in the prototype stage, there are a few deployments that suggest that it is at least usable. First, it was integrated into a research version of the Coraid SRX product. In that setting, θfs served a file system over 9P and NFS while the existing SRX AoE functionality provided block storage. The result was an integrated NAS and SAN appliance which was used internally (though far from stress tested) at Coraid research. Second, a Plan 9 file server has been configured with θfs in /boot and corresponding changes in /sys/src/9/boot/local.c. This system boots taking its root from θfs and has spent considerable time serving a 9vx terminal.

11. Future Work

The prototype as described in this paper is a snapshot of a work in progress. As such, there are a number of developments in the planning for this system. This section attempts to give a brief summary of some of them.

11.1. Permanent AoE Object Support

The object extension to AoE described here is strictly one designed for research purposes. The needs of a real extension that can be standardized and productized depend greatly on the needs of the initiator side. In other words, the exactly subset of SCSI OSD functionality that is needed in AoE depends more on how client OSs use OSD than on the specifics of our server-side research. Therefore, we leave this issue open for the time being, with the expectation that those involved with initiator development will play a critical role in establishing a permanent AoE extension for handling objects.

11.2. Log Structuring

As file system design has developed over the past couple of decades, the benefits of log structuring/journaling have become evident. These techniques are particularly valuable in their resilience to power failures and other unexpected events. Because they are presently well-understood and not necessary for our research objectives, we have not designed this prototype as either a journaled or a log-structured file system. However, neither have we made particular design decisions to preclude the use of these techniques. It is expected that a final version of this system (an ωfs) would be built around a log-structured design. However, the stage at which that happens will likely depend on experience we gain in using the prototype.

11.3. Performance

Up to this point, only a cursory pass has been made at analyzing and enhancing performance. In particular, prior to θfs, some fairly deep changes were made in metadata handling to bring the overall performance in some small tests to the same general ballpark as other Plan 9 user-land disk file systems such as kfs and fossil. No attempt has

been made thus far to measure the NFS performance of θfs or to compare it to other NFS implementations. It is anticipated that substantial effort will be made in performance improvement as the basic functionality stabilizes and in preparation for other research directions. Because little analysis or measurement has been done to date, we cannot say where significant improvement is likely to be found.

11.3.1. Hashing PID/OID Directly to QID Path

The first thing that appears to be a glaring inefficiency is the way we handle the object PID and OIDs. For each request, we have a pair of 64-bit integers that needs to map to a single 64-bit integer. On the surface, at least, it seems inefficient to make a string out of the two 64-bit integers, hash the string as a path name, and get the 64-bit QID path from the hash table, possibly incurring one or two extra disk accesses. So it would seem there could be benefit from a mechanism that translates directly from the PID/OID pair to the QID path, particularly if that translation could be done with fewer disk accesses. On the other hand, whatever mechanism is used, the net effect will still be to hash 128 bits to 64 bits. It's not clear whether the effects of cacheing are such that the existing translation is just as good as any other. Whether this inefficiency is significant and can be improved remains an open question.

11.3.2. Including Full Path Name in Metadata

The design of NFS expects a fairly conventional approach to directory organization and representation. In particular for the lookup RPC, given a unique ID for a directory and the final element of a path name, a unique ID for the specified file should be quickly obtainable. However, in the present design, the concept of a directory is primarily one of name management and not reflected in data structures or data management. To get a full path name for hashing, we build the string by traversing up the name tree to the root, prepending the name of each node found along the way. Although conceptually such an approach can be made pretty elegant, if the cache contents are unfavorable, it could require a number of disk accesses. Therefore, the NFS lookup RPC might benefit from including a file's full path name in its metadata. Doing so would reduce the path name construction to a concatenation of the directory's path name and the final name element given in the lookup request.

11.3.3. Including Ancestor QID List in Metadata

Because the 9P walk message allows up to 16 path elements to be specified, in principle, hashing the full path name should allow for faster walks by not processing each intermediate directory. However, there are a couple of ways in which the semantics of the walk expect step-by-step directory traversal. First, the response message to a walk request is specified to include the QIDs of each intermediate directory traversed. Second, it is normally expected that the permissions and ownership of each intermediate directory will determine how far the traversal can be allowed to proceed. In the prototype described here, we deal with these issues by traversing upward from the endpoint of the walk building the QID list along the way. (We currently punt on the permissions issue.) Of course, such a traversal can require additional disk accesses, depending on the contents of the cache. Including ancestor QID and permission information in a file's metadata would reduce the number of disk accesses if we find the existing approach to be problematic. This potential improvement is expected to be considered soon in conjunction with methods of providing correct permissions semantics.

11.4. Distributed Services

The last future direction we discuss here is the one that sparked the work described in this paper. Experience with a number of storage products has led us to a general dissatisfaction with architectures incorporating storage ''heads.'' Such devices tend to be

bottlenecks with numerous clients on one side and numerous low-level storage elements on the other. This kind of architecture is nearly guaranteed to lead to expending more resources to get lower performance. Instead, we expect that it is possible to develop storage components that collaborate to present to clients a single, large data store and at the same time spread the data and communications out among themselves. The storage system developed here is intended to be the per-node system on which a headless, distributed storage architecture is built.

12. Conclusion

As with most systems research, few if any of the individual components here lack precedent in earlier file or other data storage designs. However, in our experience, the particular combination of mechanisms is at the least, unusual. Taken together, these mechanisms lead to a number of uncommon and interesting large-scale characteristics. Experience with θfs suggests that it has been a successful endeavor in three ways. First, it has demonstrated workable techniques for integrating disparate methods of data access. At one level, the techniques allow us to make available files, blocks, and objects simultaneously sharing the same pools of storage. Some of those same techniques also allow us to make files available over multiple protocols that may expect different underlying storage features. Second, this prototype has established a clear path for further research into techniques relevant to stand-alone storage service devices. Finally, indications are that this work will serve well in its initial goal of providing a platform on which to carry out distributed data storage service research.

13. Acknowledgements

We would like to express our appreciation to Coraid for its support during this research. In particular, Brantley Coile established the research environment in which the work was conducted.

14. References

[1] Stuart, B.L., *Principles of Operating Systems: Design & Applications,* Boston, MA, Course Technology, 2009.

[2] Factor, M., et al, ''Object Storage: The Future Building Block for Storage Systems: A Position Paper,'' *Proceedings of the Second International IEEE Symposium on Emergence of Globally Distributed Data,''* http://www.research.ibm.com/haifa/projects/storage/objectstore/papers/PositionOSD.pdf, 2005.

[3] McCoy, K., *VMS File Systems Internals,* Boston, MA, Digital Press, 1990.

[4] Solomon, D.A. and Russinovich, M.E., *Inside Microsoft Windows 2000,* Redmond, WA, Microsoft Press, 2000.

[5] Stuart, B.L., ''An *O*(1) Method for Storage Snapshots,'' *Proceedings of the 9th International Workshop on Plan 9,* 2023.

Distributed Industrial Protocols: Serving Modbus on Plan 9

Thaddeus Woskowiak
tswoskowiak@gmail.com

ABSTRACT

In my previous WiP paper "Building Distributed Industrial Systems Using Plan 9" I discussed why building distributed industrial control systems using plan 9 is desirable. In this paper I present a practical application for using 9P to multiplex access to devices which speak the ubiquitous Modbus protocol.

1. Introduction

Modicon designed and built the first purpose built industrial control computer which we now call a programmable logic controller (PLC). Modicon also developed a protocol to facilitate communication between their controllers called Modbus, a portmanteau of Modicon bus. Modbus allows the control computer to not only communicate with other computers but with remote IO devices. These IO devices can multiplex many physical signals using interface electronics to transport this data digitally across a few wires. This greatly reduces wiring and the machine can be reprogrammed at will.

Modbus is one of the earliest examples of what is termed a "fieldbus". Fieldbuses, which is analogous to a computer bus, allow a computer to read and write remote process data from other devices. Modbus has seen widespread adoption leading to the standard being offered by most automation manufacturers. This makes Modbus a very attractive protocol for serving over 9P.

2. Modbus Internals

Modbus [1] is a client-server RPC protocol with a defined set of function codes which a client may call on a server. These function codes are single cycle request-response which carry out a number of operations such as reading or writing data. Other function codes deal with device settings, status, file records, and the encapsulation of other protocols. In addition, the specification allows a vendor to add their own custom function codes.

Modbus was originally designed to facilitate data transfer between Modicon machines which were big endian 16-bit word machines. They addressed 64k words of memory commonly referred to as registers. Register values can represent variables, packed bits, addresses, packed ASCII strings, or be part of a larger 32 or 64 bit object. Registers are also used for addressing physical IO such as digital and analog IO points. Bits are used to represent physical digital IO point states as well as boolean program variables. Bits are also called coils to represent the coil of a "virtual relay" in a ladder logic program. Due to this legacy, Modbus process data is transferred as either 16 bit big endian words or bits.

The bus portion of Modbus was originally based on a multi-drop serial bus where a single client broadcasts an 8 bit addressed Modbus frame to every server on the bus. Valid server addresses are in the range 1-247, with 0 being the broadcast address. The

address is also called the Unit ID and only one server may be addressed at a time unless it's a broadcast frame.

Each destination server checks the frame using an error checking code, then processes the request if the address matches. If the request can not be satisfied an error code is returned. A successful request results in a return code along with any requested data. Since only one server may speak at a time, servers can not respond to broadcast requests, they can only attempt to process them. All versions of the protocol feature this bus design.

Although Modicon machines were word based the wire protocol is byte based. A Modbus frame consists of a Modbus Protocol Data Unit (PDU) encapsulated in the framing data. The framing depends on which of the three protocol versions is used: Modbus RTU[9], which stands for remote terminal unit, is the original binary protocol. Modbus ASCII[9] is similar to RTU but encodes the binary values as base-16 ASCII with frame delimiting characters. Finally, Modbus TCP is similar to RTU but features a larger header designed for multiplexed networking of the server. RTU and ASCII are designed to operate over a multi-drop serial bus such as RS-485 or a single device via RS-232. Modbus TCP is designed to operate over Ethernet networks using the TCP protocol. A secure version [2] of the Modbus TCP protocol [3] is specified which encrypts TCP frames using TLS and X.509 certificates for authentication.

The Modbus PDU is the core of the protocol and contains the RPC request. It begins with a 1-byte function code followed by one or more 1 - 2 byte parameters and optional data bytes. The PDU structure for all three variants of the protocol is the same. The function codes themselves are grouped together by category: data access, diagnostics, and other. Data access consists of function codes for reading and writing bits, 16 bit words, and file records. Diagnostic function codes are only present in serial bus servers and are used to read status as well as read or write settings. The "other" category only specifies two codes for dealing with other fieldbus protocols, which is beyond the scope of this paper. An example Modbus PDU for reading multiple holding registers is as follows:

```
0x03 0x00 0x010 0x00 0x02
```

The first byte contains the function code followed by two 16 bit integer parameters: the starting address, 16, and the quantity of registers to be read, 2. The response starts with the function code copied from the request, a one byte byte-count of the data, 4, followed by the data consisting of two 16 bit integers, each containing the value 9:

```
0x03 0x04 0x00 0x09 0x00 0x09
```

If the above request generates an error the client will return an exception code. The exception code is communicated by adding 0x80 to the function code number followed by the exception code:

```
0x83 0x02
```

In this case exception code 2 indicates the starting address and quantity of registers requested is unable to be satisfied.

Data access exposes two kinds of data: read only (RO) bit and word data and read/write (RW) bit and word data. RO bits are called "input bits" which consist mainly of status bits or physical digital input points which can not be written. RO word data is stored in "input registers" which are used to represent variables which are RO such as analog values from analog to digital converters (ADC). RW bits are "coil bits" and are used to access both program boolean variables and physical binary outputs. RW registers are called "holding registers" and are used for operations such as storing program variables, configuration registers, and data exchange. These registers may overlap or be part of a completely different memory map.

3. Hardware Used

To build the library and accompanying programs it was necessary to test against hardware which implements the Modbus protocol. Four devices were used:

- Automation Direct Protos-X modular fieldbus system, a Modbus TCP server.
- Automation Direct EA6 touch screen HMI that speaks Modbus RTU, ASCII, and TCP.
- Automation Direct Click PLC. Its three serial ports speak Modbus RTU and ASCII.
- Morning Group electric meter which speaks Modbus RTU over RS-485.

4. Library Design

The first step for communicating with Modbus is to build or use an existing Modbus library. The library would have to be responsible for the transmission, reception and processing of Modbus frames as well as error reporting. I decided to write my own as an exercise and build something native to plan 9. The library design is intended to be as simple as possible while conforming to the Modbus specification as closely as possible. It is designed to be media agnostic and performs IO using file descriptors. It does not involve itself with process data and only sends and receives packed bits and word primitives in order to comply with the protocol specification.

While initially studying the protocol I realized the bus architecture is always present in each of the Modbus frame types - we always communicate the UID, even when using TCP. This means that Modbus can be thought of as an 8bit addressed bus attached to a port, conventionally a UART or TCP port.

Bus IO is performed over two file descriptors, one for transmitting and the other for receiving. The use of two descriptors makes it possible to use stdin and stdout for IO enabling mod bus programs to operate over pipes. This is especially useful for establishing TLS encrypted Modbus TCP sessions using tlssrv(8) [4] which eliminates the need for writing security code.

The client code begins by calling one of the Modbus function codes to generate a request. The request is serialized into a buffer in network order which represents the full Modbus PDU. The buffer is then framed according to the Modbus frame type and written to the transmit fd. The client code then listens for a response and returns any received data. A time out mechanism implemented using alarm(2) and notify(2). At the time of writing the client portion of the code, the library implements only the most commonly used word and bit function codes. This is more than enough functionality to read and write useful data and handle errors.

The library components used to setup a client and call functions on a server are described below:

The Modbus session is stored in a Modbus struct:

```
struct Modbus {
        inttfd;// transmit fd
        intrfd;// receive fd
        intmbtype;// MBTCP or MBRTU or MBASCII
        uchar*msg;// modbus messge buffer
        /* internal variables omitted for simplicity */
}
```

The `mbinit()` function is called with the transmit and receive file descriptors as well as the frame type. It allocates memory, copies the parameters and returns a Modbus struct:

```
Modbus* mbinit(int txfd, int rxfd, int mbtype);
```

Calling a function on a Modbus server is achieved using one of the provided function

calls. These calls take the Modbus struct along with a uid (unit id - the bus address of the Modbus server) and the necessary parameters. The following calls handle reading and writing register data:

```
/* RW holding registers */
int mbRdhreg(Modbus *m, uint uid, u16int *mbreg, uint sa, uint n);
/* RW holding registers */
int mbWrhreg(Modbus *m, uint uid, u16int *mbreg, uint sa, uint n);
/* RO input registers */
int mbRdireg(Modbus *m, uint uid, u16int *mbreg, uint sa, uint n);
```

The following calls handle reading and writing bit data:

```
/* RW coil bits */
int mbRdcbit(Modbus *m, uint uid, uchar *mbbit, uint sa, uint n);
/* RW coil bits */
int mbWrcbit(Modbus *m, uint uid, uchar *mbbit, uint sa, uint n);
/* RO input bits */
int mbRdibit(Modbus *m, uint uid, uchar *mbbit, uint sa, uint n);
```

The server code, as is the case for 9P requires the user to attach callback functions for each function code. The server code starts by listening for Modbus frames according to the frame type. The listening code is specific to each frame type and as of writing, only Modbus TCP is supported. We first start by listening for a frame, then processing it to see if it is valid and if so, send the received UID and PDU to the PDU processing routine which is responsible for calling the attached callbacks. The UID can be used for de-multiplexing, allowing a single machine to respond to multiple UIDs.

The library components for building a modbus server uses the same Modbus data structure with the addition of a data structure which holds the callbacks:

```
struct Callbacks {
        int (*fc01)(Mbreq*);
        int (*fc03)(Mbreq*);
        int (*fc06)(Mbreq*);
        int (*fc16)(Mbreq*);
};
```

The callbacks are named using the interger value of the function code. The Modbus struct is then passed to the `mblisten()` function which reads Modbus frames using the framing defined by mbtype in the Modbus struct. Integer errors returned are mapped to Modbus exception codes.

5. Client Design

The initial client design used the Modbus library directly, and so the program spoke Modbus directly to one or more servers. This approach was first used to test the library during its development. One of the first test cases for the library was a data logging project that polled the Click PLC and recorded variable values in a text file. I quickly abandoned this approach as using the library directly is very limiting as the process speaking to the bus cannot share the bus without extra code. In my opinion such functionality is beyond the scope of a control or supervisory process, and is needlessly complex. This lead in to thinking of a way to serve the bus via 9P that would allow multiple clients to concurrently share the devices on the bus. This abstraction can allow a bus to be shared across a network or even the internet by multiple clients concurrently.

5.1. Mapping Modbus Data To 9P

Since the Modbus protocol mainly deals with transferring bit and word values between machines we can think of those functions as read and write mechanisms. For example the function code "read multiple holding registers", sends two parameters to the server in the request: starting address and the number of registers to read. This corresponds to a 9P read request where in addition to requesting a specific number of bytes, we also specify a seek offset relative to the beginning of the file.

Conversely, the function code "write multiple registers", corresponds to a 9P write call. For bit operations the function codes for reading and writing use the same offset and address parameter fields.

The first issue encountered, was figuring how to map bits and words to bytes. The function call parameters map 1:1, but the data sizes do not. 9P is a byte aligned protocol whereas Modbus transfers words or packed bits. I started by working out how to map words. The obvious approach was to map the 64k word address space to a 128k byte address space. Though this leaves us with the glaring issue of how to handle requests which are not aligned on 2 bytes? Technically, unaligned reads could be handled by calculating the offset to find the required number of registers to read, and only return the necessary bytes. Though, after thinking about unaligned writes I realized there is no reason to read half a register. The simple response is to return an error on unaligned operations.

In the case of bits, alignment was no longer an issue but a different design challenge was presented. Bits are aligned by 1-bit so there is no alignment issue, but in a Modbus frame the bits are packed into bytes. Read(2) and write(2) calls return an integer value representing the number of successfully written or read bytes. Initially I thought of returning the raw byte stream from the frame keeping the efficient bit packing, but this does not map the number of bits successfully read to the number of bytes read from the file. If we want the return value of `pread(2)/pwrite(2)` to match the number of bits returned, then we can return each bit in a byte. At first this may seem wasteful and inefficient, but this is how boolean types are stored, one bit per byte, allowing us to treat the data as an array of bools.

5.2. Modbus(4): A 9P File Server Design

Now that we have a way to map sequential Modbus data to files, the file server function needs to be designed. Since Modbus presents a bus-like hierarchy similar to that of usb(3), we can implement a similar approach. The usb(3) file server is two-level, with the root representing the bus, and decimal addressed subdirectories representing devices on the bus each containing files representing control and data accesses. The same hierarchy can be applied to Modbus, with the root representing the bus, and UID numbered subdirectories containing the control, status, register and bit data.

Unlike USB, Modbus does not have an auto discovery or configuration mechanism. To set the server's UID or address, a programming device or physical switch are used to set the Modbus address. Since there is no way to scan for Modbus devices, we will have to add and remove servers manually. A `ctl` file in the root will accept control messages to add and remove server directories. A useful option is to name the directories to have sane naming in the file system (e.g. `/n/modbus/powermeter/` versus `/n/modbus/12/`). This applies to each of the supported frame types and allows us to treat them all as buses. This lets us use the same file server for RTU, ASCII and TCP servers with the added bonus of addressing UIDs on a TCP server in order to talk to devices behind a Modbus TCP bridge.

6. Serving Modbus From Plan 9

Serving the Modbus protocol from a plan 9 machine is useful for letting a plan 9 machine serve data to a Modbus client such as a PLC. This can be facilitated for both TCP and ASCII framing but not RTU. The frame delimiting mechanism for RTU uses an inter-frame quiet period not less than 3.5 times the baud rate of the bus. Since RTU does not communicate the frame length, nor does the uart(3)[5] driver have a way to delimit frames based on time, we cannot reliably read RTU frames. Because of this, ASCII framing was specifically developed for reception by general purpose computers, thus eliminating the quiet time frame delimiting problem.

My initial approach for designing a server is the opposite of the Modbus file server - to serve a file via Modbus. This approach treats a byte addressable file as memory representing a series of registers and packed bits. The server is started manually for serial ports or automatically by listen(8)[6] for a network listener.

Initially I had wanted the server file mapping to match the client file server using four files for input bits, coil bits, input registers, and holding registers. Though after some though a more pragmatic design was chosen which only serves a single file. The file layout is big endian so Modbus reads and writes are 1:1 with no data marshaling. Since bits are packed into bytes, the Modbus library makes it easy to pass a byte array between the read and write calls. The same applies for registers where each register is stored as 2 bytes in a buffer. The file to be served is passed to the server program as an argument along with the offsets for each of the four main data access functions: input bits, coil bits, input and holding registers. This allows the function code address space to be overlapped or separated over a larger file to map all 256k bytes of register space and 16k bytes of bit space.

Example command for running a Modbus listener:

```
% listen1 -t tcp!*!502 mbsrv -r 8 -h 16 /mnt/segment/mbdata
```

No argument is given for bit offsets are given so bit reads start at the beginning of the file. Word data for input registers starts 8 bytes in while holding registers begin 16 bytes in. The data stored in `/mnt/segment/mbdata` should be stored big endian otherwise the client will have to byte swap the data during marshaling.

On-disk files can be used or stored in a ram disk and treated as memory; though the real intent for this server implementation is to share synthetic files from a segment(3)[7] device that is attached to a control process. User space 9p servers such as a control process can export variable memory as a file or a variable database that is also a file server, and could be used to share variables among multiple control programs. The interface between all the components is file IO via 9P, allowing the programs to live anywhere on a network.

7. Modbus Client Program: Putting It All Together

A simple PLC program was written to demonstrate both the Modbus file server client and listener server. The control logic is implemented as a function pointer state machine where each access to state returns the next state. A loop mimicking a PLC runs a series of functions which creates a cyclic control process: read data, run state machine, write data, check for errors, repeat. We can also optionally insert a sleep() to limit the execution time. This allows one or more fieldbus calls to read process data into local variables, run the code using this data, and write that output data back to the fieldbus(es).

A few portable data marshaling functions facilitate copying of a big endian ordered byte buffer to `u16int` and vice versa. These are part of pread/pwrite() wrappers who's signatures are `u16intrdmb(intfd`, and `u16intwrmb(intfd`, functions read and write a Modbus file server's register file and properly marshal the data independent of machine

architecture.

Mapping Modbus to variables in the control code is accomplished using a shared array of u16ints. To assign these variables a name an enum is used as an index. The array is given a short name such as mb[] and an enum holds the index name, e.g.

```
enum mbvar {
   TempSet = 1
};
mb[TempSet] = 70;
```

The array is first passed to the `rdmb()` function which fills it with Modbus data, the state machine runs, and `wrmb()` functions then write the data to the Modbus servers.

Serving data to a Modbus TCP client, specifically the EA6 HMI, was implemented using a segment(3) device and the listen server. The segment memory is attached to the control process before the PLC loop is entered. The same data marshaling functions are used in the PLC loop to encode the machine dependent u16int type to and from a big endian ordered buffer. Since segment(3) memory is also served via a file the Modbus listen server uses this file to access the data stored within.

A PLC example loop using two states to implement a simple thermostat fan controller. The Modbus variable TempAmb stores the ambient temperature value while TempSet is the desired set point.

```
State s;
u16int *mb;
int fd;
s = stop;
while(pv[loopcnt]) {
            mb[MBrxb] = mbrd(fd, mb + 8, 8, 0);
            s = s(mb);
            mb[MBtxb] = mbwr(fd, mb, 8, 0);
            mb[loopcnt]--;
            sleep(slptm);
         }
void*
run(void *pm)
{
            u16int *mb = pm;
            fprint(2, "run0);
            mb[fan] = 1;
            if(mb[TempAmb] < mb[TempSet])
            return stop;
            return run;
}

void*
stop(void *pm)
{
            u16int *mb = pm;
            fprint(2, "stop0);
            mb[fan] = 0;
            if( mb[TempAmb] > mb[TempSet] )
            return run;
            return go;
}
```

8. Conclusion And Further Thoughts

The simple design of the Plan 9 operating system facilitated the rapid development of test and prototyping programs. The dial(2) library is very simple and intuitive to use which allowed for a simple fixed function TCP client to be written within a day. The 9P abstraction provides a very simple OS abstraction allowing me to open a file and read data from a device without libraries.

My overall goal is to continue working on the Modbus software until it is stable enough for daily use. From there I want to work on the control code and put more thought into the PLC process. I want a clean way to share discoverable variables from a control process using as little code as possible using both files and shared segments. Eventually other protocols will be investigated such as MQTT and the high performance EtherCAT protocol. Eventually I hope this software is stable enough to replace commercial control software in certain applications. There is more work to be done and plenty more ideas to explore.

9. References

[1] Modbus Protocol Specification: https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf

[2] MODBUS Security Protocol: https://modbus.org/docs/MB-TCP-Security-v36_2021-07-30.pdf

[3] MODBUS Messaging Implementation Guide version 1.0b: https://modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf

[4] tlssrv(8): http://man.9front.org/8/tlssrv

[5] uart(3): http://man.9front.org/3/uart

[6] listen(8): http://man.9front.org/8/listen

[7] segment(3): http://man.9front.org/3/segment

[9] Modbus Serial Line Protocol and Implementation Guide V1.02: https://modbus.org/docs/Modbus_over_serial_line_V1_02.pdf

5e: Memory Corruption Detection for the Rest of Us

Ori Bernstein
ori@eigenstate.org

ABSTRACT

Outside of the Plan 9 world, there have been a large number of tools written to detect misuse of memory. Electric Fence, Valgrind, ASAN, MSAN, and many other tools are available. Plan 9 isn't entirely left out in the cold, but we can do better. This paper sketches out one potential tool that could help us.

1. A Partial Taxonomy of Memory Bugs

There are a wide variety of memory safety bugs that people will tend to run into. Some of these include:

This happens when a variable is read from before it is written to. Our compilers will warn for this in simple cases, but can do nothing when variables are passed by reference, or when a malloced buffer is used. An example of an uninitialized memory bug may look something like this:

```
int v, x;
x = 0;
x += v;
```

Here, *v* contains uninitialized memory. Frustratingly, adding debug prints will often lead to the uninitialized memory contents changing, which can make this sort of issue frustrating to debug.

This is a relative of uninitialized access, but it can be slightly more insidious. This occurs when a variable has been malloced and then freed, but is used after it's freed. After it's freed, the memory may be reused for other data structures, which can lead to strange corruption at a distance.

```
int *m = malloc(sizeof(int));
free(m);
*m = 42;
```

This occurs when accessing outside of an allocated buffer, whether the buffer is on the heap or the stack. This kind of bug can lead to corrupting the

2. The Core Idea: Shadow Maps

Many of the errors listed above can be detected, at least heuristically, by tracking the state of memory as the data is accessed. At a cost of two bits per byte of program memory, we can track whether a byte of memory has been allocated, and whether it has been initialized. If a byte is within the or *TEXT* segment at program startup, then it can be assumed valid. A direct call to , must return zeroed memory, so it is also treated as both allocated and initialized. Memory that comes from or from a stack allocation is be treated as allocated, but uninitialized.

As the program progresses, a correct program will write to any memory before reading from it later. Thus, every access to a memory location needs to be instrumented with a shadow map check for the appropriate bits.

3. An Emulator for Debugging

There are several ways to implement a shadow map, and to hook it in to the program executed. ASAN[1] hooks deeply into the compiler, emitting function calls and inline checks. This approach performs relatively well, typically halving performance. This is a high cost, but low enough that it can be turned on for debugging real workloads. However, the change to implement it on Plan 9 would be intrusive. Instead, it seems simpler to implement the memory verification in an emulator. The arm emulator, '5e', was chosen for this purpose, primarily because in the code, all memory accesses were already passing through the *vaddr* function.

Because of this high level of centralization around memory access, the bulk of the code to hook in the shadow map is below:

```
bits = MARKALLOC;
if(op & ARD)
        bits |= MARKINIT;
checkaccess(s, off, len, bits);
if(op & AWR)
        markvalid(s, off, len, MARKINIT);
```

All accesses must touch allocated memory. Instructions that read must additionally check that the memory has been initialized. If an instruction writes to memory, it sets the initialized bits on the memory, so future reads are aware that this memory is valid.

This leaves the question of how to adjust the shadow map. The verifying emuator must be aware of allocations, frees, and similar. Therefore, the emulator uses in order to find the addresses of and other memory related functions. When the emulator PC hits these functions we pull their arguments out of the stack and registers, and update the shadow map with the addresses. We record the PC that we are returning to, and disable tracing within these functions, assuming them to be correct. When the PC returns to the recorded location, we update the memory map to match the change in memory state.

4. Deficiencies

Currently, the verifying emulator, is very rudimentary. There is a great deal of low hanging fruit for improvement.

It doesn't know about the *APE* allocator. APE programs will not have their memory issues correctly diagnosed. The best fix for this would likely be removing the separate allocator APE uses, and unifying it iwth libc.

The emulator doesn't know about how much stack a function uses. As a result, there is no mechanism for marking the top of stack uninitialized after a function returns, leading to Therefore, the stack will be marked as initialized when it should be marked uninitialized. Bugs may be missed.

It would be useful to add red zones around global and stack objects in the compiler, in order to allow easier detection of out of bounds accesses.

5. Possible Extensions

There are a number of directions to explore. In addition to hardening the tool and smoothing out the basic rough edges, it may be interesting to look into other dynamic verifications that could be done.

For example, *TSAN* that detects data races and accesses to memory without a held mutex would be incredibly useful.

Also, now that there is an emulated proof of concept, it may be worth looking into a version that hooks into the compiler or linker, and produces instrumented native binaries.

6. Availability

The code described is available at *https://only9fans.com/ori/5v/HEAD/info.html*

7. References

[1] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov ‘‘ AddressSanitizer: A Fast Address Sanity Checker,’’ *Proceedings of the USENIX Annual Technical Conference,* 2012

[2] Nicholas Nethercote and Julian Seward, ‘‘Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.’’ *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 2007.

[3] Konstantin Serebryany, Timur Iskhodzhanov ‘‘ThreadSanitizer - data race detection in practice’’ *Workshop on Binary Instrumentation and Applications,* 2009

Three More Cortex-M Inferno Ports

David Boddie
david@boddie.org.uk

ABSTRACT

Starting from a port of Inferno to a single microcontroller, work is ongoing to extend support to a wider group of microcontrollers and small systems based on ARM Cortex-M4 and Cortex-M7 cores.

Introduction

A port of Inferno to the STM32F405 [1] microcontroller has been available for over a year. Based on an ARM Cortex-M4 core, this microcontroller provides a basic level of features that makes code written for it applicable to other Cortex-M4 microcontrollers, and also to more powerful systems based on the Cortex-M7 core, without too much adjustment.

It is unclear how much code could be reused to support other related cores, such as the Cortex-M0+, which is used by the RP2040 microcontroller. However, since that microcontroller is already supported by the Raspberry Pi Pico port [2], no effort has been made to investigate this.

The descriptions that follow briefly cover some implementation issues that affected ports to systems based on Cortex-M4 and Cortex-M7 cores made during the last year. It may be useful to consult the Wikipedia article [3] about ARM Cortex-M cores to learn more about the similarities and differences between them.

Toolchain

The existing ports of Inferno to ARM systems relied on the availability of a regular 32-bit instruction set. In contrast, Cortex-M4 cores only provide support for Thumb and Thumb-2 instructions, making the use of a different toolchain necessary: a combination of `tc`, `5a` and `tl`.

Of these tools, `tc` and `tl` were modified to add support for the floating point instruction set used by Cortex-M4 cores with the appropriate extension. Additionally, `tl` was adjusted to re-enable existing support for compiling string constants into the text section of a binary. Depending on the configuration and tools used to build an Inferno system, this can lead to a substantial reduction in RAM usage at run-time.

Task switching

Cortex-M4 cores differ from traditional ARM cores in both the instruction sets used and in the available processor modes. This means that some techniques used by many existing ports of Inferno to ARM systems are not applicable to these cores. The way that task switching is performed is one area that needed revising to fit a different processing model.

Existing ARM ports rely on the ability of the processor to switch between the processor modes during exception handling, using this ability to implement a mechanism for task switching that ensures that `setlabel` and `gotolabel` calls are always called in the same processor mode.

On Cortex-M4 cores, exceptions cause the processor to enter a Handler mode that it can only leave by returning from the exception back to the interrupted code. While it is possible to perform task switching while running in this mode, the task switching mechanism also needs to be able to run in the normal Thread mode. As a result, a different mechanism is used to ensure that Thread mode is always used when performing task switching.

Floating point support

Cortex-M4 cores tend to provide some level of hardware floating point arithmetic support, and this is the case with the STM32F405 microcontroller. The existing code in ports such as the Raspberry Pi port [4] is useful in terms of providing a framework for supporting floating point instructions, but required adjustment to work with the floating point instructions supported by the newer cores.

Use of a floating point emulator is necessary for Cortex-M4 cores because they only tend to support single precision arithmetic, making it necessary to trap instructions that operate on double precision values. Many existing ports tend to rely completely on an emulator to handle all floating point instructions when an undefined instruction exception occurs. This maintains a set of emulated floating point registers associated with each process. However, since we want to take advantage of the hardware support for floating point, an approach that maintains the actual registers is used. This relies on the modification of stacked floating point registers during exceptions.

One simplification made to the floating point emulator is to only use it to handle one undefined instruction at a time instead of trying to handle as many consecutive instructions as possible. Since supported and unsupported instructions may be mixed together, the additional work of opportunistically checking for consecutive instructions is time that could be spent letting the hardware either execute them or raise another exception.

One disadvantage to this kind of partial emulation solution is the additional overhead it places on the stack for the kernel and each process, as space for a full set of floating point registers is reserved on the stack when an exception occurs. A pure software solution might prove to be sufficient if use of hardware floating point support doesn't provide substantially faster floating point arithmetic performance.

It should be noted that Cortex-M7 cores can be provided with double precision floating point units. For these, either minimal or no emulation is needed.

Memory savings

As described in the previous work [5], some effort was needed to reduce the amount of RAM used by Limbo modules on a running system. This was achieved by expanding the code sections of Dis files during compilation of an Inferno system and running them directly from flash memory.

It was observed that the freezing process had the effect of moving much of the root file system from the data section to the text section of the binary. This has the effect of reducing the initial RAM overhead of the system. To take advantage of this feature for general use, the `data2texts` utility was created to replace the existing `data2s` utility which normally translates binary data into assembly language `DATA` structures for compilation into the system. This variant simply uses `TEXT` directives instead, enabling the data to be stored alongside kernel code.

Additional savings to the initial RAM footprint of the system are obtained by passing the `-t` option with the `tl` loader. This has the effect of placing string constants in the text section of the binary, again reducing the amount of data copied into RAM.

The following table shows the effect of using `data2texts` instead of `data2s` to encode data, and the larger effect of including string constants in the text section of the binary for a microcontroller with 384 kilobytes of RAM. The upper part of the table shows the effect on a system with non-frozen Limbo modules; the lower part shows the effect with frozen Limbo modules.

		Data strings		*Text strings*	
		data2s	**data2texts**	**data2s**	**data2texts**
Non-frozen Limbo modules	**kernel**	477272	477272	477272	477272
	maxsize	165120	324608	339712	339968
	hw	132832	132832	132832	132832
Frozen Limbo modules	**kernel**	614584	614608	614584	614608
	maxsize	309504	324608	339712	339968
	hw	94048	94048	94048	94048

The **kernel** rows show the size of the system binary in bytes, containing the kernel and root file system. With non-frozen modules, this value is not affected by the changes, and there is only a small effect on the system with frozen modules. What is noticeable is the increase in size of the binary when modules are frozen.

The **maxsize** rows show the number of bytes of RAM available to the system. Using the `data2texts` utility provides a memory saving that is largely overshadowed by the effect of compiling string constants into the kernel text. Since there is an overlap in these approaches, the combination of them yields only a marginal additional memory saving. Freezing Limbo modules also provides a similar benefit, but only when these other approaches are not used.

Where freezing modules provides a distinct benefit is in the run-time use of RAM by modules. The **hw** rows show the corresponding values in bytes from the summary provided by reading the `/dev/memory` file, indicating the allocation high water mark after running the `ls` utility. Note that this is only affected by the use of frozen modules, with the other approaches having no effect on the run-time RAM usage. This is where we see clear RAM savings by eliminating the need to allocate memory for parts of the module loading process at run-time.

New platforms

The initial choice of the STM32F405 microcontroller as a target for Inferno was determined by its low RAM provision, but was also a pragmatic decision based on the perceived effort needed to port Inferno to the ATSAMD51J20 microcontroller. Returning to this microcontroller later, the port was not as difficult as expected.

Similar microcontrollers with more RAM were obtained to determine how much of Inferno's functionality could be used on these small devices. The first of these was the SparkFun MicroMod Artemis processor, which uses an Ambiq Apollo3 core. In an attempt to expand coverage to related processors, a development board with an i.MX RT1062 processor was also obtained due to its improved features and its use of a Cortex-M7 core.

Generally, the limiting factors to a viable Inferno port appear to be the amount of flash memory and RAM provided by a target. Informal tests have indicated that about 256KB is the minimum amount of flash memory that a system image needs for Thumb-2 code, and probably at least 512KB would be a comfortable minimum. Although some activities are possible with less than 256KB of RAM, there is more room for experimentation with 384KB or greater.

Since the ports created after the first used the initial port as a base, the result was four ports with a fair amount of duplicated code. To make this easier to work with, the shared code was consolidated in a single `cortexm` directory in much the same way that the `sa1110` directory contains common code for a number of older related ports.

Conclusion and future work

Porting Inferno to microcontrollers based on ARM Cortex-M cores continues to be an interesting activity, with plenty of scope for improvements and optimisations. It is hoped that future work will help to reduce Inferno's memory footprint even further, making it possible to run it on a wider range of microcontrollers.

The source code associated with this work can be found in the following repository:
https://github.com/dboddie/inferno-os/tree/cortexm

References

1. David Boddie, "STM32F405 Port of Inferno",
   https://github.com/dboddie/inferno-os/tree/stm32f405
2. Caerwyn Jones, "Raspberry Pi Pico Port of Inferno",
   https://github.com/caerwynj/inferno-os/tree/pico
3. Wikipedia, "ARM Cortex-M", *https://en.wikipedia.org/wiki/ARM_Cortex-M*
4. Lynxline, "Porting Inferno OS to Raspberry Pi",
   http://lynxline.com/porting-inferno-os-to-raspberry-pi/
5. David Boddie, "Hell Freezes Over: Freezing Limbo modules to reduce Inferno's memory footprint",
   9th International Workshop on Plan 9, Waterloo, Canada, 2023
   https://9e.iwp9.org/

Adapting Plan 9's listen to GNU Guix

Edouard Klein

Abstract

Here is a comprehensive adaptation of Plan 9's elegant network service management design to the Linux environment, focusing on the Guix System distribution. The proposed `listen` utility initiates network services by executing files named after the protocol and port they serve. This approach offers significant advantages over traditional Linux setups: per-user, per-port, and per-program allocation of ports, to be contrasted with the binary privileged/unprivileged model on Linux; enhanced security through process isolation; and network transparency for service scripts. We also detail the development of auxiliary tools and contributions such as a Go-based 9P2000.L FUSE client needed for container isolation, improvement to the `p9ufs` 9P2000.L server, and a network-transparent implementation of the finger protocol. We straighforwardly achieve a level of simplicity and security that is currently only achievable on Linux with complex configurations or not at all. The paper concludes with reflections on the challenges and limitations encountered in adapting Plan 9's models to the Linux platform, pointing out the inherent difficulties in reconciling Linux's legacy structures with Plan 9's more streamlined and network-native approach.

1. Introduction

Plan 9 uses a piece of software called `listen` to start network services.

Network services are defined by the presence, in a directory watched by `listen`, of executable files whose name is of the form `<protocol><port>`.

For example, the `tcp7` file implements the `echo` protocol on port 7.

When a client connects on one of the ports, `listen` starts the corresponding executable file, and

- forwards incoming data from the connection to the process' `stdin`,
- forwards outgoing data from the process' `stdout` back to the connection.

This elegant design presents three advantages over the current state of affairs on Linux:

- Instead of a root-owned configuration file, the use of one file per port allows a per-user, per-port, per-program allocation of ports (see section 5).
- Plan 9 being Plan 9, each process gets its own *namespace*, that is, its own view of the system resources, isolated from other processes. Such a level of security is available on Linux only through the use of containers (see sections 7.2 and 7.3).
- Network services written for `listen` are said to be *network-transparent*: they need not contain any network code at all. As they read and write on the standard streams, they require less work to write than the same networked service would; they can rely on the plethora of existing command line filters that, too, read from `stdin` and write to `stdout`. An example implementation of echo (section 6.1) and finger (section 6.2) are provided, that illustrate this concept.

This document describes our effort in adapting this design to Linux in general and to the Guix System distribution in particular.

The deployment of our version of `listen` on `the-dam.org`[1] frames the exposition that follows. However, it should be noted that `listen` is not dam-specific, or even Guix System (the operating system) specific. While it requires GNU Guix (the package manager, see section 3.2) to be installed, it can be[2] deployed on other Linux distributions than Guix System. Doing so requires careful system administration and painful wrangling with `systemd`. Guix System (the operating system) avoids this thanks to its declarative configuration primitives (see section 4) and absence of `systemd`.

The version of `listen` presented here is in active use on `the-dam.org`, and the echo (section 6.1), finger (section 6.2), and git (section 6.3) services are reachable with, respectively:

[1] our public access UNIX ("PubNix") server, running Guix System with Beaver Labs' guix channel

[2] and has been, on my own workstation running Arch

```
echo hello | nc the-dam.org 7
echo | nc the-dam.org 79
git clone 'git://the-dam.org/listen'
```

2. Contributions

2.1. `listen`

The work presented herein yielded more than the `listen` bash script one can download with:

```
git clone git@the-dam.org:listen
```

2.2. `f29p`

A Golang 9P2000.L FUSE client was developed from scratch:

```
git clone git@the-dam.org:f29p
```

This client is needed to mount 9P servers in the containers in which `listen` isolates the network services (section 7.3). It is a seldom documented fact of Linux that unprivileged mounting is reserved to a select few filesystems, among which 9P is not present. No other piece of software has ever been made public that allows one to mount a 9P2000.L server from within a Linux container.

2.3. `p9ufs`

A pull request implementing UnlinkAt was merged into p9ufs, a Golang 9P2000.L server: `https://github.com/hugelgupf/p9/pull/87/files`

2.4. `os/listen`

`listen` requires deep changes in the system it is installed on (for example, a change in the default privileged port ranges, see section 5), but we wrote a set of operating configuration functions (see section 4) that makes installing `listen` a literal one-line change in a configuration file. These functions can be added to Guix System through our channel (Klein 2023).

```
(channel
  (name 'beaverlabs)
  (url "https://gitlab.com/edouardklein/guix")
  (branch "beaverlabs"))
```

2.5. `fingerd`

A network-transparent implementation of finger (see section 6.2) was developed to work with our `listen`. It should work with Plan 9's `listen` as well, as soon as somebody writes a Python 3 interpreter for Plan 9, which should be right around the time we sunset IPv4 globally.

Fetch it with:

```
git clone git@the-dam.org:fingerd
```

3. Glossary

Before we delve any further into technical explanations, we need to make explicit that there are three sets of words whose meaning is highly ambiguous.

This ambiguity needs to be removed lest the article become very confusing for the reader.

3.1. Namespace

First comes *namespace*. A namespace in Plan 9 is the view a process has of the filesystem. Because every system resource is available as a file, a namespace is the view a process has of the system's resources (Pike et al. 1993).

On Linux, namespaces are an isolation mechanism bolted upon processes. Linux processes are not, by default, as isolated from one another as they are in Plan 9.

Because on Linux everything is definitely *not* a file, there are multiple kinds of namespaces. One can be in a network namespace, a mount namespace, a user namespace, etc. (Kerrisk 2013; *unshare(1) – Linux User's Manual* 2023; *namespaces(7) – Linux User's Manual* 2023)

To remove this ambiguity, we will explicitly specify "Linux namespace", "Linux mount namespace", etc. or use the improper term of *containers* when talking about the specific Linux isolation mechanism, and reserve the bare *namespace* term for the Plan 9 generic notion of a process' view of the system resources.

When we talk about the *namespace* of a Linux process, we talk about what it sees of the underlying system. This view is constructed with the help of different, potentially nested Linux namespaces, and 9P mounts through FUSE.

3.2. Guix

GNU Guix is a package manager (Courtès 2013), that can be installed on any Linux distribution.

Guix System is a Linux distribution that uses GNU Shepherd as its daemon manager, lacks systemd, and of course uses Guix as its package manager. It goes further by providing a declarative configuration system for the whole operating system, with atomic updates, roll-backs, etc. (Neidhart 2019)

Waters are muddied by the fact that the `guix system` command, provided by GNU Guix the package manager (which can be installed on any Linux distribution) allows one to instantiate a Guix System operating system as a VM, a container, a docker image, etc.

We will use *Guix* to refer to the package manager, and *Guix System* to the operating system.

3.3. Service

In UNIX parlance, a *service* is a background process (a *daemon*), typically launched by a service manager (SysVinit, rc, shepherd, systemd, etc.), examples include network services such as a web server, but also the cron daemon.

In the context of `listen`, *services* refer to the network services that `listen` manages.

On Guix System, a service is a broader notion, that encompasses network services, daemons, but also any aspect of the system configuration, such as udev rules, user accounts, etc.

We will always specify, unless the context makes it absolutely clear, whether we are talking about network/listen services, or Guix services.

We will avoid calling UNIX services services, and use *daemon* instead.

4. Operating system configuration functions

As mentioned in section 2, `listen` is not merely a bash script, but also a set of deep modifications to the operating system it runs on:

- installing `listen`, `f29p` and `p9ufs`;
- creating nobody-like user and group `listen`;
- Making, chowning, and chmoding the following directories, according to some port attribution policy:
  - `/srv/listen/`, owned by `root:root`, with perms `rwxr-xr-x`,
    - in it, the `tcpXXX` and `tcpXXX.namespace` scripts, owned by *e.g.* `alice:listen`, with perms `rw.r-.--`;
  - `/run/listen/tcpXXX/`, owned by *e.g.* `alice:listen`, with perms `rwxrwx--`;
  - `/run/9p/`, owned by `root:root`, with perms `rwxrwxrwt`;
  - `/var/log/listen/tcpXXX/`, owned by *e.g.* `alice:listen`, with perms `r-xrwx--`.
- creating the default guix profile in `/run/listen/profile`;
- making sure `/run/listen` is deleted on reboot;

- setting the available source ports for outgoing tcp connections to 49152-65535 away from its default of 32768-60999;
- setting the privileged port range to 0-48152 away from its default of 0-1024;
- setting the `cap_net_bind_service +p` on the `with-cap-bind` wrapper, and making it `r-x----` for `listen:listen`.
- creating, for each user, an HTTPS redirect from *e.g.* `https://alice.example.com` to the first port of `alice`'s range;
- creating services aliases as in section 7.5, e.g. `/srv/listen/finger -> /srv/listen/tcp79`;
- starting the `listen` daemon on boot,
- but only after all the daemon it needs have started first.

While it is technically possible to apply these modifications to a Linux system using the usual system administration tools (`sysctl`, `adduser`, `chown`, etc.), it would be an ill-advised tall order.

Using our additions to Guix System's declarative configuration mechanism is easier and safer.

Familiarity with GNU Guix, Guix System, or GNU Guile is not expected from the reader, as the examples provided here are quite self-explanatory to anyone with any programming experience. Just understand that what is written `f(a, b)` in most languages is `(f a b)` in GNU Guile, and that the last expression of a function is its return value.

The provided primitives are based on an extension of Guix System's configuration mechanism. Guix System relies on a directed acyclic graph: nodes being Guix System services, and an edge from *e.g.* `nginx` to `account` denoting that installing `nginx` will create a user account on the system. This graph is folded[3] into a script, collapsing all the extensions into a GNU Guile script that actually changes the system so that it conforms exactly to the declaration (Courtès 2015; *Service Composition – (GNU Guix Reference Manual)* 2024).

While powerful, this mechanism is hard to extend as it requires familiarity with both Guix System and GNU Guile. We abstracted it away thanks to the use of functions that take an operating-system record as an argument, and return a modified operating-system record. These functions can thus be chained, human-centipede (Six 2010) style, in a syntax much more familiar to users of imperative languages, not unlike a Dockerfile, while keeping all the power of the Guix System service-graph mechanism.

5. A fine-grained access control API for ports

Without `listen`, ports on Linux fall under the coarse dichotomy of *privileged* and *unprivileged*. *Privileged* ports are traditionally ports below 1024 [4]. One used to need to be `root` to bind to a privileged port, gaining privileged access to the whole operating system as a side effect. This changed when Linux got a new feature called *capabilities*. The `CAP_NET_BIND_SERVICE` capability allow one to bind to privileged ports on a per-program basis. Having this capabilities grant no other rights ; but this capability applies to all privileged ports: `CAP_NET_BIND_SERVICE` accepts no port-based configuration.

`listen` solves this problem because it equates ports with file names. Using UNIX's file access control API (`chown`, `chmod`, etc.), one controls access to ports on a per-port, per-user (and in turn, with the `setuid` bit, per-program as well) basis.

By keeping an empty, world-writable `/srv/listen/` directory, `root` can let any user bind to any port.

At the other end of the spectrum, `root` can create all the `tcp*` files (from `tcp1` to `tcp65535`, one per port), and `chown` each of them to whomever should control the associated port. The files remain non executable, until the appropriate user wants to activate the server, at which point she runs `chmod +x` on the file.

On `the-dam.org`, the last few bits of the hash of the username yields a port number, and we give that user a 12-port range starting at the number derived from the hash. This let us avoid any kind of bookkeeping for port allocation: users come and go, each get an automatically assigned range with a negligible risk of collision. We also offer an https redirection from *e.g.* `https://alice.the-dam.org` to `the-dam.org`'s first port in `alice`'s range.

This port allocation is automatically derived from the list of human users any time a user is added or removed, with no need for human intervention.

[3] in the functional programming sense of `fold`

[4] but a `sysctl` call to set the `net.ipv4.ip_unprivileged_port_start` kernel variable will change that.

6. Network service scripts

Let's see how `listen` works on `the-dam.org`, from the point of view of the user. The implementation is detailed later, in section 7.

We will study three different network services in increasing order of complexity:

- the `echo` service on port 7, which echoes back whatever the clients send,
- the `finger` service on port 79, implemented as a simple Python script reading and writing to and from the standard streams,
- the git protocol on port 9418, implemented with `git deamon`, which insists on listening on a port instead of using the standard streams.

In contrast with what almost always happen on Linux, where processes gets a full view of the system limited only by its owner's identity, `listen` launches network service scripts in an extremely limited namespace, as Plan 9's `listen` does. Our `listen` must use a container to achieve this while Plan 9's `fork` (*fork(2) – 9front's Manual* 2024) offers this kind of isolation for free. Network services running in an empty namespace are not very useful. This section shows how users can populate their network service's namespace.

6.1. Echo service

The `echo` protocol is implemented by linking `cat` to `/srv/listen/tcp7`:

```
ls -l /srv/listen/tcp7
lrwxrwxrwx 1 root root 65 Mar 21 21:14 /srv/listen/tcp7 -> \
    /gnu/store/mppp9hwxizx9g9pikwcvvshb2ffxyq7p-coreutils-9.1/bin/cat
```

This link is created by Beaver Lab's `listen/echo` function:

```
(define (listen/echo os)
  "Return a copy of OS, in which listen's echo (tcp7) service is active."
  (extend-service
     os
     activation
     #~(begin
         (when (file-exists? "/srv/listen/tcp7")
           (delete-file "/srv/listen/tcp7"))
         (symlink #$(file-append coreutils "/bin/cat") "/srv/listen/tcp7"))))
```

This service can run in the empty namespace that `listen` provides by default.

6.2. The finger service

The above `echo` example ranks among the simplest things one can do with `listen`. Let's study now a more complex service, the `finger` service listening on port 79. Our finger implementation is a simple 68-lines python script that reads a query on its standard input, parses it, sets some of the env vars from the CGI specification[5], and execs the requested script in `/srv/finger/`. This script's output is what the remote finger client gets.

This network service is active now, and one can simply query it with a finger client or by running:

```
echo | nc the-dam.org 79
```

to get a list of the available user names.

`finger` illustrates how to clear two hurdles most real life use-case will meet:

- it needs to access data from outside the container,
- it relies on custom software, unavailable from the vanilla version of `guix` provided by `listen`'s container.

[5] Allowing queries like `finger 'hello?name=alice&greeting=Howdy@the-dam.org'`

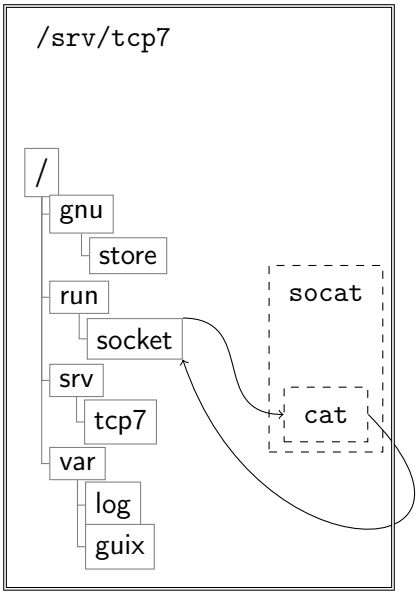


Figure 1: The container in which `listen` runs the echo service. From the point of view of the service script (`cat`), data comes from the standard input, and the system is almost completely empty.

Accessing data from outside `listen`'s container is done, as it is in Plan 9, by mounting 9P servers in the service's namespace.

`listen` will look for a `/srv/finger/tcp79.namespace` file. This is not, as it is in Plan 9, a namespace file understood by `newns` or `addns` (*auth(2) – 9front's Manual* 2024). It is a proper script, called by `listen`, and expected to setup its own environment, its own view of the filesystem, and its own network before calling `exec $@`, relinquishing control to a command built by `listen`.

The default namespace script, `/srv/finger/listen.namespace`, called in the absence of a service-specific script, is quite simple:

```
#!/gnu/store/aslr8ym1df4j80ika5pfxy5kbfv4iz3w-bash-5.1.16/bin/bash
set -euo pipefail
# This is the default namespace file for listen services. This is where you
# mount the 9P services all your listen services need.
exec "$@"  # Once done, call the command provided as an argument by listen
```

The declarative configuration will not overwrite this file if it already exists, making it easy to configure the default namespace in a site-specific manner.

`finger`'s namespace file contains:

```
#!/usr/bin/env bash
set -euo pipefail
mkdir -p /srv/finger
f29p unix!/srv/9p/finger /srv/finger &
exec "$@"
```

This mounts the 9P server listening on `/srv/9p/finger` to `/srv/finger`.

This 9P server daemon is started and managed by `shepherd`, Guix System's daemon manager, thanks to a call to `os/9p-serve` in the function that configures `finger`:

```
(os/9p-serve "/srv/finger/" "/srv/9p/finger" #:name '9p-finger
             #:user "listen" #:owner "listen" #:group "listen" #:mode "700")
```

This call creates a socket on `/srv/9p/finger`. On the socket listens an instance of `p9ufs`, a modern Golang 9P2000.L implementation. This process is owned by `listen`, and so is the socket.

On Plan 9, namespace operations fail or succeed thanks to the authentication mechanism embedded in 9P. This authentication mechanism relies on the `factotum` process mounted on `/mnt/factotum` having the required credentials (Cox et al. 2002).

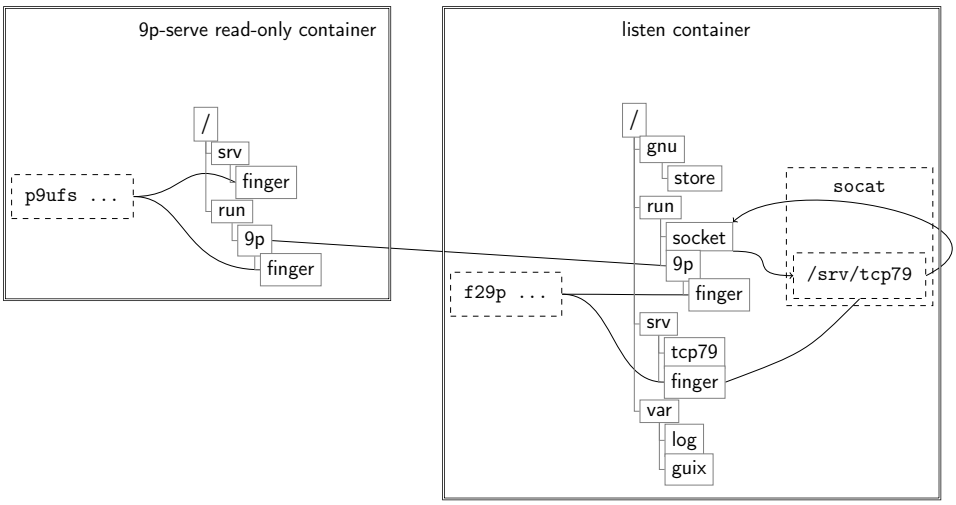


Figure 2: The containers in which `listen` runs the finger service: the `p9ufs` 9P server has read-only access to the global `/srv/finger` directory, and this access is transferred via the `/run/9p/finger` socket to the `tcp79` network service script, which 9P-FUSE-mounts the server in its own namespace. Note that here, as in `echo`, the network service script has no access to the internet and receives data on its standard input, and sends data on its standard output.

Despite a previous attempt at porting this mechanism to Linux (Klein and Gette 2023), this mechanism is not readily available to `listen`. Instead, to control the operations on files outside of the service container, `listen` relies on the ownership of the `p9ufs` process and the ownership and permissions of the socket file.

In this particular case, `listen` gets a read-only by default (one has to pass `#:read-write "1"` to `os/9p-serve` to get read-write access) view of `/srv/finger`, with the same rights as it would have outside of the container. In the next section we will see an example of user `git` "lending" its read rights on `/srv/git` to `listen` (which can't read `/srv/git` whose owner, group, and mode are `git:users r-xrwx--`).

This choice makes it possible for e.g. `alice` to provide `finger` information to her servermates only, who, belonging to the `users` group, may call her script while logged-in, whereas anonymous users from the internet, being confined to `listen`'s identity, won't see her script as executable:

```
ls -l /srv/finger/alice
-rwxr-x--- 1 alice users 121 Mar 18 10:57 /srv/finger/alice
```

Using custom software is done with a Guix profile. While on Plan 9 one makes software accessible by binding various directories over `/bin`, the story is more involved on Linux. Due to constraints imposed by ubiquitous dynamic linking, interpreted languages, and mostly-standardized-but-not-quite defaults paths, one has to rely on a myriad of *search-paths*. *search-paths* are environment variables that tell the software where to find the resources it needs.

`guix` can setup profiles (Courtès 2018) that are links to its immutable, content-addressed, *store*. In it are synthetic directories with links to all the needed resources (them, too, living in the *store*), as well as a `profile` script that will set the search-paths to those resources.

`listen` will load the profile in `/run/listen/tcp79/profile` before calling the service namespace script and the service script. A default profile is used if `/run/listen/tcpXXX/profile` does not exists.

In the case of `finger`, the profile is created by a call to `os/profile`:

```
(os/profile "/run/listen/tcp79/profile"
            #:name 'finger-profile
            #:packages '("python" "the-dam-org-f29p"))
```

which sets up two packages: the `f29p` 9P FUSE client, and the Python interpreter.

6.3. The git daemon

As revealed by an analysis of the fourth edition source code, 9front's source code, and a question asked on 9fans (*Content of your /rc/bin/service or /dis/svc ? – 9fans thread* 2024), Plan 9 relies, for the following protocols (as well as a few others), on servers reading and writing on the standard streams: ftp, (ssh), telnet, smtp, http, pop, imap, samba, rlogin, lp, 9P (of course).

Some of those servers, such as `rc-httpd` have been ported to BSD or Linux, but others rely for example on `/net` (Presotto and Winterbottom 1993), which does not exist (yet ?) on Linux.

On the-dam.org, git repositories are made available through a user called `git`, as which anybody can connect through ssh, *e.g.*:

```
git clone git@the-dam.org:listen
```

However, a bug (*Can't clone a git repo over anonymous SSH — issues.guix.gnu.org* 2023) prevents Guix from using this kind of access, and so we must open port 9418 to let git use its own unauthenticated, unencrypted protocol (but Guix checksums the code after fetching so those two points are not that big of a deal).

Git does provide an easy way to run a server speaking this protocol: just run `git daemon`. However, `git daemon` binds to port 9418, and does not use the standard streams.

One could extract the git protocol from the code and create a standalone standard stream network transparent server, but that is a lot of work.

What we do instead is abuse the namespace script. Instead of relinquishing control to a command crafted by `listen`, here is what `tcp9418.namespace` does:

```
#!/usr/bin/env bash
set -euxo pipefail
mkdir -p /srv/git
f29p unix!/srv/9p/git /srv/git &
ip link set lo up
git daemon --base-path=/srv/git --export-all&
exec socat -dd UNIX-listen:/run/socket,fork TCP-CONNECT:127.0.0.1:9418
```

Lines 3 and 4 play the same role as for `finger`: making the necessary data available to the service by mounting a 9P server.

Said 9P server is started thanks to this line in the operating system configuration function `listen/git-daemon`:

```
(os/9p-serve "/srv/git/" "/srv/9p/git" #:name '9p-git
             #:user "git" #:owner "listen" #:group "listen" #:mode "700")
```

Here we see that only user `listen` can use the socket to mount the 9P server, but the process listening on the other end is not owned by `listen` but by `git`. This is because `listen` is not privileged enough to read the actual `/srv/git`. By making user `git` run the 9P server, `listen` gets exactly the kind of access it needs, on a very specific and well-defined subset of the global filesystem.

Line 5 of the git daemon namespace script sets up a loopback network interface. Behind the generic "container" term hide Linux' namespaces (same name as Plan 9, different feature). Important here is a Linux network namespace, that makes a process believe it owns all network interfaces (it does, but not the actual ones). Setting a dummy loopback interface on line 5 lets us start `git daemon` on line 6, where it will happily listen on port 9418, blissfully unaware that nobody but its parent can see the interface it is listening on.

Instead of calling `exec $@` as expected, the namespace script instead execs `socat` in such a way than any connection to the `/run/socket` socket is forwarded to the container's port 9418. Creating this socket is the signal `listen` is waiting for before it starts forwarding incoming data from the internet to the service. Failure to make it appear within a handful of second is a sign that the service failed to start.

Usually `listen` calls `socat` inside the container like so:

```
socat -dd UNIX-listen:/run/socket,fork EXEC:/srv/$service
```

where `/srv/$service` is of course the service script, which will communicate with the internet through its standard stream.

By hijacking the expected behaviour and calling socat by itself, `tcp9418.namespace` can forward data to `git daemon`'s port instead.

As for the `tcp9418` script, it is just a link to `true`, it is never called anyway, it is only used so that by detecting its presence `listen` will start the service.

To make `git`, `ip`, etc. available to the service, `/run/listen/tcp9418/profile` is created by a call to:

```
(os/profile "/run/listen/tcp9418/profile"
            #:name 'git-profile
            #:packages '("git" "iproute2" "coreutils" "bash" "socat"
                         "the-dam-org-f29p"))
```

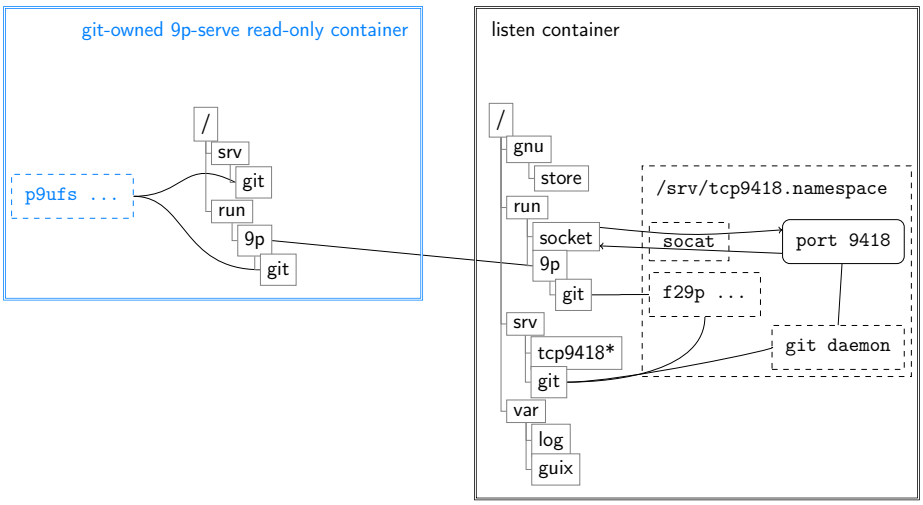


Figure 3: The `p9ufs` 9P server has read-only access to the global `/srv/git` directory, and is owned by the `git` user (user `listen` wouldn't be able to read `/srv/git`). This access is transferred, via the `/run/9p/git` socket and a 9P FUSE mount, to the namespace of the git service script, itself owned by `listen`. The git service script is itself never started, instead the `/srv/tcp9418.namespace` sets up a loopack network interface, starts the `git daemon`, and hijacks the usual procedure by launching socat, instead of relinquishing control to `listen`. This allows the incoming data to be redirected to the container's port 9418 instead of `/srv/tcp9418`'s standard streams.

6.4. To recap

From the user's point of view, service scripts are fully network transparent. They read data on their standard input and write answers on their standard output. One instance of the script will be launched for each new connection.

Service script may specify software dependencies by setting a Guix profile in `/run/listen/tcpXXX/profile`, either manually (by calling `guix install --profile=/run/listen/tcpXXX/profile ...`) or with the declarative operating system configuration function `os/profile`.

Service scripts are run in a container fully isolated from the rest of the system. To access data from the outside, one mounts a 9P server in the service's companion namespace script. This 9P server can be started manually (calling `p9ufs`), or through the use of the declarative operating system configuration function `os/9p-serve`.

Abusing the service namespace script is possible to let non-network transparent servers run without having to rewrite them to use the standard streams. This facility allows typical web-applications to be run with no modifications. `the-dam.org` redirects `https://alice.the-dam.org` to `alice`'s first port in her range, and does so for all users, making web application deployment easy, and rootless.

6.5. To go further

Additional configuration abstractions are available, for example the unprivileged counterparts to `os/profile` and `os/9p-serve` are `home/profile` and `home/9p-serve`. They allow non-root users to run their own 9P servers and create their own profile, so that the service script they own can access them without root having to intervene. Describing them is outside the scope of this document, but they are available to `the-dam.org` users.

Also, the `register-listen-service` function is available to inform the `shepherd` that some daemons (typically, the 9P servers) shall be started before `listen`.

The bareness of the container in which the service script runs provides two main advantages:

- It increases security by isolating the service script from the rest of the system. An attacker gaining remote code execution in the container has no obvious way to impact the rest of the system except a denial of service through resource exhaustion (*e.g.* a fork bomb.).
- It forces the explicit declaration of all used resources (software dependencies, files) and processes (in our example, the git server). This explicitness makes it trivial to run the exact same script in a slightly different container for development, testing, or failover purposes.

7. How `listen` works

7.1. Configuration

`listen` will loop over the files in `/srv/listen/` whose name matches `tcpXXX` where `XXX` denotes a port number.

When it finds an executable file, for example `/srv/listen/tcp7`, `listen`:

- creates an isolated container in which it:
- activates the service's profile at `/run/listen/tcp7/profile` if it exists, or the default one at `/run/listen/profile` otherwise,
- runs the service namespace script `/srv/listen/tcp7.namespace` if it exists, the default namspace script `/srv/listen/listen.namespace` otherwise,
- runs `socat` to link the `/run/listen/tcp7/socket` socket to the service script `/srv/listen/tcp7` standard streams,
- logs the standard error in `/var/log/listen/tcp7/service.log`.
- Then outside the container, it launches a daemon that will listen on port 7, and forward data back and forth to and from the service script (via `/run/listen/tcp7/socket`).

`listen` uses `inotify` to watch the `/srv/listen/` dir for changes. When notified of a change, `listen` scans the whole directory again:

- `listen` leaves alone running daemons
  - whose service script remains executable,
  - and whose content has not changed since it started[6],
- `listen` restarts running daemons
  - whose service remains executable,
  - but whose content changed since they started,
- `listen` stops every running daemon whose service script has lost its execution bit.
- `listen` starts every service script that has become executable.

7.2. First security layer: almost unprivileged user `listen`

The `listen` process runs owned by `nobody`-like user `listen`. This protects user data from unauthorized reads or writes should `listen` become compromised or misbehaving.

As a second line of defence, we would have liked `listen` to run in a container, seeing of the system only what its job requires [7]:

- GNU Guix's paths:
  - the read only store `/gnu/store/` where GNU Guix installs all software,
  - the `/var/guix/` dir to communicate with the guix daemon,
- the service scripts in `/srv/listen/`,
- the log directory in `/var/log/listen`,
- a runtime directory in `/run/listen/`,
- the host's network.

[6] To achieve this, `listen` caches the hash of the script at start time, and compares the current hash with the cached one during the new scan.

[7] Note that Plan 9 or Inferno make this trivial

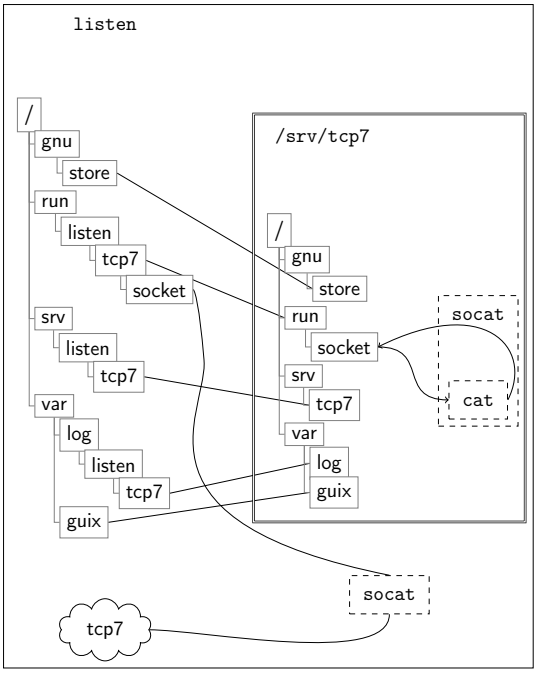


Figure 4: `listen`'s container prevents damage, even when the service process (here `cat`), gets compromised.

Alas, Linux mirrors our low-growth,high-capital-returns economy: privilege remains an inherited property (Piketty and Zucman 2015). Despite Linux namespaces and capabilities, this inheritance model prevents unprivileged access to non-file resources like ports from a container [personal communication with the kernel developers]. One example: Linux does not honor `setuid` or `setcap` binaries within containers. Solving this issue would require

- either moving away from the inheritable `root`-owns-everything model, or its somewhat finer-grained capabilities substitute,
- or exposing non-file resources like ports as files.

We must for now fall back on basic Linux inter-process isolation.

To prevent any other user from messing with the ports `listen` manages, we set all ports as privileged.

Despite their complexity and coarse-grainedness, we make use of capabilities: user `listen` can remain wholly unprivileged but for its ability to bind to all ports. A good step in the right direction from `inetd` mandatorily running as `root` !

We use Linux capabilities to grant user `listen` the right to bind to privileged ports (*i.e.* all the ports), using a `setcap`-ed wrapper named `with-cap-bind`, that only user `listen` can execute.

7.3. Second security layer: service script containerization

Despite `listen` having quite a low footprint on the system, it remains a sensitive target:

- it can access the internet,
- it can listen on any port,
- it can kill or mess with existing service script processes.

Yet, the service script processes constitute a more probable target than `listen` itself. Indeed they do not just pass data around as `listen` does: they *handle* this untrusted user data. Pirates use complex operations on untrusted data, *e.g.* parsing, as a typical attack route.

Because service scripts present a big attack surface, they must run in an environment so isolated that a full takeover would not negatively impact the rest of the system much.

To this effect they run in containers. GNU Guix's `guix shell` creates the service containers. They only expose:

- GNU Guix paths,
- the service-specific (e.g. `/run/listen/tcp7`) runtime directory,
- the service-specific (e.g. `/var/log/listen/tcp7`) log directory,

Note that the script can not access the network ! Communication with the outside happens through the UNIX domain socket `/run/listen/tcp7/socket`.

7.4. Passing data to the unnetworked daemon

The `/srv/listen/tcp*` service scripts get their data via the `/run/listen/tcp*/socket` [8] socket.

The brilliant utility `socat` acts as plumbing between any two kinds of process input/output (sockets, files, standard input/output/error, etc.).

For *e.g.* port 7, `listen` calls it this way:

```
/run/setuid-programs/with-cap-bind socat -dd \
    TCP4-LISTEN:7,fork,reuseaddr \
    UNIX-connect:/run/7/socket \
    >> /var/log/tcp7/listen.log 2>&1 &
```

`socat` runs with the following wrapper, arguments, and options:

`with-cap-bind` grants `socat` the right to bind to privileged ports (see section 7.2).

`-dd` enables logging the remote IP addresses. This then feeds tools like `fail2ban` to react against abuse.

`fork` connect to the UNIX domain socket each time a client connects to the tcp socket.

`reuseaddr` the `socat` daemon might die with its TCP socket on port 7 in the `TIME_WAIT` state. With the socket in this state, no new `socat` daemon can start again: it would get an "address already in use" error. Going out of the `TIME_WAIT` state can take up to 4 minutes. To avoid such a delay, `socat` ignores the `TIME_WAIT` state thanks to the `reuseaddr` option.

Interested readers will find an explanation of the `TIME_WAIT` state in the UNIX socket FAQ.

7.5. Aliases

Port numbers make for a terrible user interface. To use service names instead, one just has to symbolically link the service name to the service script, *e.g.*

```
ln -s /srv/listen/finger /srv/listen/tcp79
```

This allows the owner to edit `/srv/listen/finger` to change the finger service, instead of having to remember that finger clients connect to port 79.

It also allows authenticated users to start the script by name:

```
ssh alice@the-dam.org /srv/listen/finger
```

The script will have an effective user id of `alice`[9], and might therefore give more information than the anonymous version accessible to anyone. By contrast, `listen` starts the anonymous version with an effective user id of user `listen`, who can not read much on the system.

[8] Note that this path appears as `/run/socket` to the service script, see figure 4.

[9] Unless the script's author has set its setuid bit.

8. Alternatives to `listen`

8.1. Plan 9's listen and Inferno's svc

A study of the source code of the fourth edition of Plan 9, 9front, and inferno, as well as asking for `listen` service script examples of 9fans (*Content of your /rc/bin/service or /dis/svc ? – 9fans thread* 2024) make some usage patterns emerge.

Namespace files: while in the default installation few services appear to have a custom `tcpXXX.namespace` file, some services like ftp and http do. The others use the site-specific default namespace file `/lib/namespace`. These put the files to be served where the service expect them, without any complex software configuration mechanism.

Seamless authentication: On Plan 9 and inferno, a user's identity spans a whole cluster, not just a single machine. Authentication is baked into the 9P protocol, and requires either less than 15 lines of boilerplate, or direct handling by some standard middleware. Switching a process' identity, say from `alice` to `bob`, requires the caller to prove to the `auth` server that it knows `bob`'s secret. This is automagically handled by the `factotum` at both ends of the connection. The namespace file benefit implicitly from that mechanism, provided the currently mounted `factotum` knows the required secrets.

Simple scripts: Only the two simplest services (the `echo` and `discard` protocols) in the default installation are fully implemented as shell scripts. The rest are custom servers written usually in C for Plan 9 and Limbo for inferno, or thin shell wrappers around those custom servers. Reports from the 9fans mailing list, however (*Content of your /rc/bin/service or /dis/svc ? – 9fans thread* 2024), praise the ease with which one can write a custom network service for a specific use case, such as collecting `vac` hashes after a backup, a text-based zine, a connection helper to some MUDs, etc.

`/net` **usage**: The synthetic `/net` hierarchy (Presotto and Winterbottom 1993) ref is wildly used among the `listen` service scripts, which get it as an argument. As with all the resources on Plan 9, this part of the filesystem can be exported and shared, in total or in part, between different processes on different machines.

We want to transpose the patterns to Linux, despite the lack on integration of two of the basic primitives of the Plan 9 experience:

- per-process view of the filesystem (namespaces),
- unprivileged modification of said view.

This lack of integration on Linux was compensated by

- the use of Guix containers (which offer a per-container view of the filesystem, see section 7.3),
- and our new `f29p` FUSE 9P client, which allow for unprivileged mounting of 9P2000.L servers from within a container, something that was absolutely impossible before, see section 2.2.

Namespace scripts replace namespace files, as there are no `newns` or `addns` on Linux.

Process and file ownership and permission make a poor, but workable, substitute to the seamless 9P authentication mechanism of Plan 9. As detailed in section 6.3, one can run a 9P server as one identity, and use the socket's ownership and permission to allow another user to benefit from this identity's rights. The alternative, which `inetd` uses, would be to run as `root`.

`/net` is replaced by a single socket from which the service script can talk to the incoming connection. A Linux network namespace isolates the service script from the host's network. Compromises made due to the complex and mostly undocumented interactions between Linux namespaces and Linux capabilities (see section 9.2) make porting `/net` to Linux a very worthwhile target to improve `listen`.

8.2. (x)inetd

On Linux and BSD, `inetd` used to be the tool that would, like `listen`, launch a network service when a client connects and forward data back and forth between the connection and the service's standard streams.

`inetd` has fallen out of fashion:

- most servers have grown in complexity, they do request management on their own.
- `inetd` listens on behalf of several servers, these servers start on-demand when a request comes. This saves CPU and memory, but increases latency, and costs one process per request.
  - The CPU and memory gains decreased to insignificance because of decreasing hardware costs.
  - Conversely, from a business perspective, the cost of latency has gone up.

- Processes cost more than threads (on Linux). Self-managed servers (such as *e.g.* `nginx`) use a single process (sometimes one process per core) and handle each request in a thread. This makes `inetd` more expensive under load than a single more complex server.
- Nowadays, people use containers (with `podman`, `docker`, `kubernetes`, etc.). Containers typically simulate a whole operating system to run one or a small handful of services. Container orchestration software fulfill `inetd`'s role and more.
- Like most good things from UNIX, `inetd`'s role now eschews to `systemd`, the cancer that will eat Linux's userspace. Almost all modern distributions lack an `inetd` (except for a handful of indomitable ~~Gauls~~ sensible Linux distributions like GNU Guix). BSDs have kept their sanity and their versions of `inetd`.

`inetd` reads its configuration from a text file, typically `/etc/inetd.conf`. Editing this file requires privileged access. This leads to difficulties in a multi-user context. `listen` addresses this.

`inetd` also provides each service process with a complete view of the system, limited only by the ownership under which said service process starts. In contrast, `listen` places security first by running the service scripts in a initially void namespace.

8.3. Authbind

`authbind` exists since 1998. It allows access to privileged ports on a per-user and per-port basis. It does so by masking an application's call to `bind`. The application ends up calling `authbind`'s version, which uses a privileged program to call the real `bind` in the application's stead.

8.4. Systemd

`systemd` offers to run ephemeral services, as inetd does. However using this functionality means you also have to chose systemd for everything else the systemd developers bullied their way into (including, but not limited to: your login manager, your desktop environment, your DNS, your logs).

9. Further work and position

9.1. Unprivileged 9P mounts

When we begin this adaptation work, we naively believed that Linux capabilities, containers, and the Linux kernel's support of 9P would allow us to emulate Plan 9's per-process namespace on Linux.

The first bad surprise happened when we discovered that within a Linux mount namespace, one can only mount a handful of filesystem types, the only non trivial of which is FUSE. 9P is excluded. We therefore had to write a 9P-to-FUSE wrapper (see section 2.2) instead of relying on the kernel's v9fs.

We would like to work in the direction of a patch that would allow in-container unprivileged mounts of 9P filesystems on Linux.

9.2. Porting `/net`, to avoid mixing capabilities and Linux namespaces

Another bad surprise happened when we discovered that, first setuid binaries, and worst, setcap binaries, had no effect in Linux user namespaces. After emailing the kernel developers themselves we got confirmation that what we wanted to achieve: use a `setcap` binary to grant `listen` the ability to bind to privileged ports *from within a Linux user namespace*, was impossible.

This makes it impossible to run `listen` in its own namespace, as we initially wanted to do (see section 7.2), instead it gets a full view of the system, limited only by its identity.

Porting `/net` to Linux, would allow us to run `listen` in a container in which the host's `/net` is 9P-imported, and stop caring the Linux capabilities feature of which was mildly said that "coherence in its design and implementation are not particularly evident"[10].

9.3. Pledge

After running their initialization code, and just before exec-ing into `listen`'s provided command, namespace scripts should freeze their namespaces, and prevent the service script from doing anything not strictly necessary to answer the clients' requests. One way to do that is to port OpenBSD's `pledge` to bash. Justine has written

[10] `https://lwn.net/Articles/632520/`

an awesome port of pledge() to Linux. We wish to incorporate it in a loadable bash builtin, which would allow pledges to happen mid-process in bash, and not just as a wrapper to a command.

9.4. Tighten the capable wrapper

`listen`'s ability to bind on privileged ports hinges on the `with-cap-bind` wrapper. As of *<2024-01-08 Mon>* this wrapper supports any command. Its only use consists of a `socat` invocation that redirects data from a port to a service container's `/run/socket`:

```
/run/setuid-programs/with-cap-bind "$(which socat)" -dd \
    TCP4-LISTEN:"$port",fork,reuseaddr \
    UNIX-connect:/run/listen/"$service"/socket \
    >>/var/log/listen/"$service"/listen.log 2>&1 &
```

We should make the port number the only argument to the wrapper, and bake the `socat` invocation into the wrapper. That way, when an attacker gains remote code execution as user `listen`, it can not bind arbitrary services to any port unless it also gains write rights to `/srv/listen` (which user `listen` does not have).

9.5. Listen services

Our work on `listen` started when looking for a clean way to write a `finger` server and getting frustrated that none existed. Our `finger` service being operational, our sights are now set on new protocols aiming for simplicity like gemini and nostr.

9.6. Preventing resource exhaustion

As mentioned in section 6.5, an attacker taking over a service process could exhaust the system's resources. We are working on a fix to that, but Linux' API for resource limits has complex interactions with Linux namespaces, that we do not fully understand yet.

9.7. Down with port numbers

Securing the `listen` process proved the most frustrating part of the design work.

For example: the question "how do unprivileged users bind to privileged ports ?" admits no good answer, because the question itself is wrong on so many levels:

- Port number themselves stem from TCP emerging from earlier protocols (see the early RFCs 322, 349, 433 and those that obsolete them), and a clean design would probably elect to eschew them, leveraging a 2^{128} address space to allow process-to-process communication, instead of the route-to-host, then route-to-process dance we do know.
  - The host to process frontier should be an implementation detail on the receiving end, not baked so deeply in the stack.
  - This barrier may even change from request to request as new hosts come up or down depending on load.
  - This already happens anyway with *e.g.* kubernetes, but we would have less cruft if it was baked into the protocol.
- We should use strings instead of numbers to specify the protocol.
- Apart from ease of implementation and historically underpowered hardware, why does one need to specify privileged ports as a single range ? One should be able to restrict access on a per-port basis.
- Even then, why should one manage ports as a monolithic entity that either all users, or no user but `root`, can access ?
- Ports should benefit from a fine-grained access control API, like files do. Exposing ports as virtual files would allow that.

10. Conclusion

The work herein proposed is but a leaky dam, failing to contain the waterweight of historical legacy.

11. Bibliography

References

auth(2) – 9front's Manual (2024). [Accessed 25-03-2024].

Can't clone a git repo over anonymous SSH — issues.guix.gnu.org (2023). https://issues.guix.gnu.org/64648. [Accessed 25-03-2024].

Content of your /rc/bin/service or /dis/svc ? – 9fans thread (2024). https://marc.info/?t=170937608300001&r=1&w=2. [Accessed 25-03-2024].

Courtès, Ludovic (2013). "Functional package management with guix". In: *arXiv preprint arXiv:1305.4584*.

— (2015). *Service composition in GuixSD*. https://guix.gnu.org/en/blog/2015/service-composition-in-guixsd/. [Accessed 25-03-2024].

— (2018). *Multi-dimensional transactions and rollbacks, oh my!* https://guix.gnu.org/en/blog/2018/multi-dimensional-transactions-and-rollbacks-oh-my/. [Accessed 25-03-2024].

Cox, Russ et al. (2002). "Security in plan 9". In: *11th USENIX Security Symposium (USENIX Security 02)*.

fork(2) – 9front's Manual (2024). [Accessed 25-03-2024].

Kerrisk, Michael (2013). *Namespaces in operation, part 1: namespaces overview [LWN.net] — lwn.net*. https://lwn.net/Articles/531114/. [Accessed 25-03-2024].

Klein, Edouard (2023). *Tutorial: Add Beaver Labs' Channel to Guix — the-dam.org*. https://the-dam.org/docs/tutorials/BeaverLabsChannel.html. [Accessed 25-03-2024].

Klein, Edouard and Guillaume Gette (2023). "Dr Glendarme or: How I Learned to Stop Kerberos and Love Factotum". In: *IWP9 2023*.

namespaces(7) – Linux User's Manual (2023).

Neidhart, Pierre (2019). *Guix: A most advanced operating system*. https://web.archive.org/web/20210528220842/https://ambrevar.xyz/guix-advance/index.html. [Accessed 25-03-2024].

Pike, Rob et al. (1993). "The use of name spaces in Plan 9". In: *ACM SIGOPS Operating Systems Review* 27.2, pp. 72–76.

Piketty, Thomas and Gabriel Zucman (2015). "Wealth and inheritance in the long run". In: *Handbook of income distribution*. Vol. 2. Elsevier, pp. 1303–1368.

Presotto, David L and Phil Winterbottom (1993). "The Organization of Networks in Plan 9." In: *USENIX Winter*, pp. 271–280.

Service Composition – (GNU Guix Reference Manual) (2024). https://guix.gnu.org/manual/en/html_node/Service-Composition.html. [Accessed 25-03-2024].

Six, Tom (2010). *The Human Centipede (First Sequence)*. https://www.imdb.com/title/tt1467304/.

unshare(1) – Linux User's Manual (2023).

centre, left and right*

beyond the stereotype

Daniel Maslowski

IAOTAI

Abstract

We propose a refresh of what a system and its environment shall look like. The times they have a-changed. No more terminals connect many people to a shared “mini”computer that really is the size of three fridges, and no longer do we pile up PCs. We have very capable phones, tablets and laptops now, plus lots of gadgets, IoT and building automation equipment. That calls for radically controversial shifts in multiple directions to rethink compute models and operating systems yet again.

Introduction

Before looking at the ideas presented in this paper, we begin with questioning and breaking traditions. We observe a bizarre evolution of *computers*. Having started in labs just like a lot of other innovation, the first eventually electronic computers had their applications in research and paperwork replacement. We neglect the military use case deliberately. Today, computers are widely used for bureaucratic processes, ecommerce, communication, entertainment, and still some research. The ideas of the *personal* computer and online *services* have a huge impact on privacy. Under those preconditions, we take a look at how things worked out.

Text is not universal

Unix developers have so often talked about text as a universal language. How is that working out? Misunderstandings are omnipresent in communication, not limited to meltdowns on mailing lists. So today’s messengers offer sending multimedia messages, with pictures, audio and video. In the 90ies, mobile services included MMS already.

There are different alphabets and languages. Different cultures are contributing to the possibility of clashes. Think of some Asian alphabets with imagery characters instead of abstract sounds represented by letters. Emoji may help to a degree. Either way, is text a universal primitive in computing?

*Thanks to Ron Minnich as well as the Open Source Firmware, Plan 9 and RISC-V communities

We propose to learn from graphical applications that have been developed in the last three decades. Keep in mind that some things are easy for some people, not necessarily simple, and hard for others. A good bunch of work has gone into accessibility for that reason.

For example, we frequently fight with compilers. Clever designers try to figure out how to explain errors well. Would graphics help a lot with that? Certainly in documentation, graphs are used frequently. The existence of ASCII graphics itself shows that text is insufficient, e.g. when a compiler error message points to a position in a code snippet using dashes, pipes and caret characaters.

Files, folders, cabinets, archives... where dust gathers

The traditional desktop has grown from paperwork imagery. The abstractions therein cause mental complexity. Some information may get lost from that. A *virtual desktop* may expose overwhelmingly much and too little at the same time. Where are the documents? What are all those files? A first look at a typical computer leaves many people puzzled. And even experienced software engineers are having a hard time getting URL parsers right.

As another example, let us explore a Linux filesystem and look at a random file we may find: what the heck is `/proc/234565/smaps`? How is it with hardware access? We argue that writing arbitrary strings to files for hardware effects is weird at best. And noone can keep those many IDs used all over the place in their mind at any given point. So they are solely leakages for reasons that remind us of bookkeeping. We associate that with boring administrative jobs. People make errors in those because it is easy to to make them. Looking at compilers again, they strive to *mitigate* common errors today, mainly by being very strict, like a school teacher.

Nobody likes paperwork and administrative processes except those who want it: legislators, governments, and institutions. Forms, files, folders and even papers are very tiring. Is it time for new concepts and/or naming? Web applications shifted away from bare forms that feel like editing databases by hand to specific interactive widgets. For example, posting on social media does not require filling in your name all the time. A lot of information is already present and can be reused.

Graphical applications

Our hardware evolves and explodes in diversity. A lot of special purpose SoCs with various peripherals, especially with the royalty-free RISC-V instruction set architecture now, keep popping up like mushrooms in the woods. Parts for specific applications such as machine learning and multimedia processing ASICs have become ubiquitous. Mediocre laptops come with a lot of processing power and integrated graphics acceleration, media encoders and decoders, and buses for high enough throughput to watch a 4k movie.

From the days of text based terminals, we are still left with lots of non-graphical tools that may as well be integrated GUIs. Yes, `drawterm` is a very fine start! With `cpu` we bring our tools when connecting to remote machines. How about bringing the GUIs as well? We could come with

a self-serving binary that can be attached to from other machines again and keeps running persistently.

Not invented here

Do we really have to have our own kernel? What are the benefits? Experiments are nice on clean slate. With protocol design, we can leverage way more existing infrastructure. For example, Linux has thousands of drivers that would not need porting. Plan 9 concepts and primitives have been transfered to Linux, such as namespaces and process isolation.

Projects like Glendix, *Plan 9 from User Space* aka plan9port and 9base have brought the systems together. Ron Minnich created https://github.com/u-root/cpu and https://github.com/u-root/sidecore in the vein of a Plan 9 feel.

There are many implementations of and readily available systems offering file servers, such as TrueNAS with NFS and many network storage appliances on the market. It doesn't have to be 9p alone. And streaming media has its own requirements.

The ideas

The following ideas can be summarized as elements of a *personal compute infrastructure*[1].

WebGAP

The *Web General Application Protocol* is a reiteration and combination of a multitude of existing concepts. Our vision is a generic protocol for heterogeneous computing environments consisting of many different devices and platforms in the sense of several architectures and operating systems running on a diverse mix of chips and further hardware from various vendors.

We are deploying web applications today that run in part in remote environments and in part in the browser by shipping initial bundles which provide an interactive experience via widgets, fetch additional data at runtime, and communicate with external services. This is concenptually similar to some of the first RFCs from the 60ies proposing command based protocols that draw on the screen of a terminal. Similarly, there were ideas for running an application on the terminal that was pushed out from a *minicomputer* or *mainframe* before.

To support today's rich applications with multimedia content, we already have protocols in place for streaming such as WebRTC and active connections via Web Sockets. For the interactive part, the web browser has grown to a runtime environment beyond a traditional desktop, inherently integrating with the outside world and the hosting operating systems with WebUSB, WebSerial, and other components as described in project Fugu. Many libraries and frameworks have grown around those concepts, such as React, Angular and Vue. However, some gaps still remain.

[1] https://qlyoung.net/blog/personal-infrastructure.html

WebALE

The *Web Application Launch Environment* replaces traditional desktops with a simple environment similar to what you know from phones and tablets. However, the *model* differs, in that apps are *self-serving*. Think of an environment like `rio`, but with a web based feel. Similar to running code via `cpu` through `drawterm`, you drop a *syslet* to a remote that serves you back both initial UI and data. I.e., it's like `cpu`, but with apps in mind. However, the point is to leverage *resources* beyond just compute. As of now, IoT sensors and actors (temperature, humidity, door locks, fans, window blinds…) and gadgets (cameras, watches, NVRs…) are all connected over networks, yet cumbersome to set up and *manage*.

Having an application specific protocol decouples from the remote's architecture, in that only a well-defined daemon on the remote is sufficient and code for its runtime. Base the runtime on WebAssembly and WASI, and the OS is abstracted again. We already see a similar trend with cloud environments, and that is what Wasm is being tailored for. Remember *Dis*?

Mind that we suggest a *model*, not an implementation. You can already do most of this with currently existing primitives.

Putting things together

The web as a platform has evolved from documents to applications, and we strive to further develop it into a full computing environment. All that is necessary for the plumbing is protocols with authentication in mind. Besides *factotum*, the well-established SSH protocol may serve as another primitive. We neglect discussing further details at this point and list a few historical examples for comparison to round up our thoughts instead.

Web browsers in the 2000s until the 2010s had plugin APIs for extension, so third party runtimes such as Flash and Java created their own platforms, with a nuance of applications called *applets*. The promised portability had worked out only to some degree, and local appliances with management interfaces would hardly ever get upgrades. The security situation became a nightmare. WebALE puts one's own devices in control in such a way that less additional parties are involved.

Given that everyone has a laptop or more these days, as well as phones, tablets, gadgets, IoT and home automation equipment, we seek to change the compute model accordingly to unleash the potential therein. One way to deploy services now is Kubernetes, a distributed container runtime environment that needs no cpu or SSH, being declarative mostly, or *software-defined service infrastructure*. We can let IoT be similar with WebGAP, keeping it all local.

GPU Filesystem for Plan 9

Joel Fridolin Meyer
joel@sirjofri.de

ABSTRACT

Many modern computer systems have dedicated hardware for computing graphics and other floating point-heavy operations. GPU manufacturers and organizations try to standardize interfaces, often providing standard APIs and drivers for their specific hardware. This WIP paper describes a potential filesystem for Plan 9 for dedicated GPU hardware.

Graphics or not graphics?

GPU hardware evolved from very specialized pipelines for 2D (and later 3D) rendering to more generic processing units. In the last few years GPU manufacturers started to sell ''unified shaders'' that allow even more generic computing pipelines. Hardware accelerated AI computation and more specialized graphics processing pipelines[1] justify dropping native graphics processing support, at least in an initial implementation of a GPU filesystem for Plan 9.

This generic use of GPUs allows us to ignore many graphics-specific parts of the API, as well as potential drivers. Of course it also makes our interface a non-standard interface, but due to the nature of Plan 9 filesystems this should be fine.

The implementation

Since most GPU drivers are very complex and it takes a long time to develop them, this first implementation will be fully CPU based, using an interpreter engine to run the shaders. The interfaces however should look as similar as possible to a potential GPU implementation.

Due to the nature of filesystems, this should make it easy to ''upgrade'' applications to actual GPU hardware: Using a different GPU filesystem is all that's needed. The software itself doesn't need to change.

Since real GPU drivers for uncommon operating systems are a rarity, this implementation approach also makes software that relies on the GPU filesystem work on systems that don't have dedicated graphics. Obviously, there'll be a huge speed difference in this case.

The filesystem interface

This implementation will provide a very simple interface that's based on existing APIs, mostly OpenGL and Vulkan. Because of this simplicity the interface is not set in stone and many details are missing. In fact, the proposed interface might not work at all with real hardware, but it can be a start.

---

[1] Epic Games' Nanite does rasterization within a shader. They also plan to make Nanite compute the shading on the software level. [Nanite]

The shader language is SPIR-V, which is used in Vulkan as an intermediate language. The shaders are loaded into the filesystem, which will compile them further down to the specific hardware, so they can be executed. [SPIR-V]

Buffers are represented as files within the filesystem. This gives us the flexibility to access the buffer contents by standard file IO.

Management is handled via a console like interface as a control file. Using this interface, it is possible to initialize new shaders and allocate new buffers, as well as control binding and program execution. For debugging purposes, an additional `desc` file is used to display all shader and buffer bindings.

Shaders and buffers are represented as the general concept of ''objects''. Each object has its own subdirectory within the GPU filesystem (1). After initializing a new shader or buffer using the control file the client can read back the ID of the object. With that, the application knows which object directory to access.

1. The GPU filesystem `/dev/gpu`.

```
/ctl         (control file)
/desc        (descriptors file)
/0/buffer    (sample buffer file)
/0/ctl       (sample buffer ctl file)
/1/shader    (sample shader file)
/1/ctl       (sample shader ctl file)
```

Shaders and buffers can be loaded by writing to their files; the application can also read their contents (2).

2. Loading shaders and buffers.

```
cat myfile.spv > /dev/gpu/1/shader
cat mydata.bin > /dev/gpu/0/buffer
...
# bindings, see (4)
# compile shader and run, see (3)
...
cp /dev/gpu/0/buffer > result.bin
```

The filesystem can't know when a shader is loaded completely. Because of that, it is necessary to tell it to compile the shader. This can be done by issuing the *compile* command on the shader control file (3).

Since a single SPIR-V program can contain multiple entry points (*OpEntryPoint*), it is necessary to specify which shader function to run (3).

3. Compiling and running a shader.

```
echo c        > /dev/gpu/1/ctl
echo r main   > /dev/gpu/1/ctl
```

When executing the shader, it's essential to specify the number of workgroups. The workgroup size represents the count of shader invocations for that workgroup, and it's defined within the shader program itself. Vulkan uses 3D vectors to specify the amount and size of workgroups. [wkgrp]

Binding shaders and buffers

Shaders and buffers need to be bound in some way that enable shaders to access the right buffers. It's hard to understand all the requirements of the actual hardware without diving deep into GPU architecture and existing APIs. [desc]

Our implementation provides a very simple abstraction that is based on Vulkan and the concept of *descriptor pools* and *descriptor sets* with their bindings. Ideally, the same abstraction may be used for GPU hardware.

Each shader is bound to a descriptor pool. A descriptor pool can describe many descriptor sets, which in turn point to the buffers.

While shaders are bound to a full descriptor pool, buffers are bound to a single binding slot within a descriptor set. Shaders have everything needed to access a specific buffer compiled in their code. They know the set and binding of the buffer they want to access.

Because this information is compiled in the shader, it is still possible to switch buffers by changing the binding itself.

(4) shows how to create a new descriptor pool and set up bindings. In this example, buffer 0 is bound to the second (index 1) binding of the first (index 0) descriptor set of descriptor pool 0.

4. Binding shaders and buffers.

```
# set up descriptor pool with 2 descriptor sets
echo n p 2      > /dev/gpu/ctl
# allocate 4 bindings
echo s 0 0 4    > /dev/gpu/ctl
# bind buffer 0
echo b 0 0 0 1  > /dev/gpu/ctl
# bind shader to pool 0
echo b 0        > /dev/gpu/1/ctl
```

Reading the file `desc` shows us the layout of this structure (5). We can see that only one binding is bound (showing the number of the buffer), while the other bindings are unset (showing -1).

5. Example descriptor table.

```
DescPool 0
    Set 0
        0  -1
        1   0
        2  -1
        3  -1
    Set 1
```

While the `desc` file can be parsed and interpreted, it should be noted that it's only meant for debugging and reviewing. Applications should use the interface provided by the control files.

State of code and future work

The code currently covers the described filesystem interface completely, however not all functionality is implemented. Furthermore, there are bugs to be expected. [gpufs]

There's a rudimentary SPIR-V assembler as well as a SPIR-V disassembler. Both are far from feature-complete according to the SPIR-V specification, missing instructions may be added easily.

To properly test the interface, it makes sense to implement the filesystem as a kernel device of Drawterm. This way we can use existing drivers and APIs on other operating systems.

Due to the lack of actual GPU hardware support, I don't expect much performance gain compared to other implementations with the same logic. However, the interface is generic enough to allow applications to use different GPU implementations: GPU hardware, CPU hardware (single or multi threaded), network scenarios.

It makes sense to think about future integrations into devdraw. The GPU filesystem could control actual images of devdraw and enable faster draw times for graphics rendering.

Since SPIR-V is very low-level, it also makes sense to develop shader compilers for higher level languages like GLSL or HLSL. Applications are developed by different people and for different reasons, so those compilers should not be part of the specific filesystem implementations.

References

[Nanite] Epic Games, ''Unreal Engine Public Roadmap: Nanite - Optimized Shading'', https://portal.productboard.com/epicgames/1-unreal-engine-public-roadmap/c/1250-nanite-optimized-shading, 2024.

[SPIR-V] The Khronos Group Inc., ''SPIR-V Specification'', https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html, 2024.

[gpufs] Meyer, Joel, ''gpufs'' and ''spirva'', https://shithub.us/sirjofri/gpufs/HEAD/info.html and https://shithub.us/sirjofri/spirva/HEAD/info.html, 2024.

[desc] vulkan tutorial, ''Descriptor pool and sets'', https://vulkan-tutorial.com/Uniform_buffers/Descriptor_pool_and_sets, 2024.

[wkgrp] OpenGL Wiki, ''Compute Shader'', https://www.khronos.org/opengl/wiki/Compute_Shader, 2024.

Portability has outgrown POSIX

Jacob Moody
Sigrid Solveig Haflinudottir

ABSTRACT

We aim to reevaluate the role of POSIX in regards to portable software and expound on a potential redesign of APE that is built with more modern software assumptions in mind.

History

POSIX is a specification of a "standard operating system interface"[Open18] that is designed to allow programs to work portably across different specific operating system environments. POSIX.1 was first codified in 1988, and since then has had varying levels of adoptions across various operating system vendors. Plan 9 also ships with a mostly implemented variant of this standard in the form of APE[TrickeyApe].

Due somewhat to the age of the standard, POSIX is somewhat limited in its surface area. The POSIX surface area is quite small with a focus primarily on lower level system software. Compared to most modern end-user programs the POSIX surface area they use is quite small [Atlidakis16]. Notably it is missing designs for audio, graphics, keyboard and mouse input, Unicode, encryption, among many other core features of operating systems today.

In the current landscape of software engineering, portability is often accomplished through the inclusion of a library that builds on the core operating system interfaces. This would be projects like SDL2[SDL], QT[QT], GTK[GTK], and so on that aim to provide consistent interfaces across the major operating system vendors. APE was designed while POSIX was still a new curiosity[TrickeyApe], and generally has not been revisited since then. With changes to adopting more abstracted libraries for the purpose of portability, it makes sense to reevaluate how this type of design could be leveraged by something similar to APE.

The design of APE

The APE ecosystem on Plan 9 is parallel to the native system, meaning that it provides its own C library, shell, and system utilities which are built on top of the kernel alone. This was done to provide a more strictly compliant interface for using existing software that targeted POSIX. However the Plan 9 kernel has a much smaller system call base than a typical UNIX system and thus in order to provide the same interfaces, such as `select()`, some amount of complexity must instead be handled by the APE library. This requires positioning APE between the calling application and the kernel in a lot of cases, most notably for file descriptors. This makes some amount of sense, if APE is going to provide the facilities of a UNIX kernel, it needs to have some of the same control that a UNIX kernel has over the resulting program.

To give an example of how much is going on with these file descriptors, here is an excerpt from how these files are wrapped:

```
typedef struct Muxbuf {
        int n;
        unsigned char*  putnext;
        unsigned char*  getnext;
        char            fd;
        unsigned char   eof;
        unsigned char   roomwait;
        unsigned char   datawait;
        int             copypid;
        unsigned char   data[PERFDMAX];
} Muxbuf;

typedef struct Fdinfo{
        unsigned long   flags;
        unsigned long   oflags;
        uid_t   uid;
        gid_t   gid;
        char            *name;
        Muxbuf  *buf;
} Fdinfo;
```

While we commend the authors of APE for writing a POSIX compliant `select`() implementation we think that this abstraction raises a lot of problems. Plan 9 uses file descriptors for many things, even more so than UNIX, as in comparison things that would be library or syscall interfaces are instead presented through the use of file systems. The problem that arises here with having this layer on top of file descriptors is that making use of any native interfaces becomes more difficult. Care must be taken to ensure that assumptions made about how file descriptors work under native conditions and under APE match, and often these are at odds.

Things become increasingly tricky as one wants to make use of libraries that use other libraries as they may have assumptions about buffering (or lack thereof). These design components restrict the amount of code that may be shared between the native environment and APE, and instead obligates copying large portions of code. While this is largely harmless it does increase the maintenance burden if not controlled.

A brave new APE: NPE

NPE (pronounced "nope") is a reimagining of a compatibility layer with modern portability in mind. NPE sits on top of the native Plan 9 C library, and uses the native facilities to provide a familiar enough UNIX-like interface along with implementations of portable libraries such as SDL2 and pthreads. NPE strives to be the minimum shim layer needed to get alien code to run. Compared to APE's somewhat involved wrapping of file descriptors, all that NPE does in order to provide `open`(), `read`(), `write`(), and `close`() is:

```
#define O_RDONLY OREAD
#define O_WRITE OWRITE
#define O_RDWR ORDWR
#define O_CLOEXEC OCEXEC
#define O_TRUNC OTRUNC
```

That's it, just get the flags right and use the native code. NPE is designed to prefer a good enough implementation that requires the least amount of fanfare over the more strictly correct alternative. This of course does impose some limitations on what programs may expect of the library, but for typical use cases this issue is quite uncommon. In the rare event where there is some POSIX detail that differs, a `#ifdef` or `#define` suffices to resolve the issue.

The other large benefit of such a design is that it allows leveraging existing native libraries. NPE presents two examples of this in the pthread and SDL2 implementations,

which make use of the native threading and graphical libraries. This type of design removes the need for maintaining code duplicated solely for compatability and allows for improvements and novel designs of the native interfaces to be tested against larger external projects with minimal overhead. In addition, the API shims are also not very expensive: the SDL2 implementation is meager three thousand lines and the pthread implementation clocking in at a dash over two hundred.

There of course are some limitations to this approach. NPE works best when targeting programs which don't assume strict POSIX compliance but rather more abnstract operating system interfaces. We find that code targeting Windows, UNIX and MacOS to be typically within this realm, whereas code written for a specific operating system, suchg as an OpenBSD daemon, is more outside of the comfort zone for something like NPE. Naturally, NPE really shines when used with a program that makes heavy use of SDL2, which in turn provides many core operating system-esque interfaces due to its application in embedded systems like game consoles. SDL2's interfaces impose little restrictions or name conflicts and are generally pleasant to write implementations for.

It could also be said that code which works easily with NPE's constraints is likely not too much further from being directly portable to native Plan 9 code. This seems to be true generally, however for recent and active software projects keeping a tidy diff against upstream makes it easy to adapt to upstream changes. Since NPE allows the user to pull in native libraries in harmony with NPE itself, this makes for a nice progressive porting experience when the goal is to produce an entirely native port.

Success Stories and Future Work

NPE was first used to port UXN[UXN] to Plan 9 but quickly found use for additional programs. As of the writing of this paper NPE has been used to port Fast Tracker 2, Pro Tracker 2, Duke Nukem 3D, Rise of the Triad, Wipeout, tinygl, Cave Story and RGBDS among other smaller utility programs. None of these programs required conflicting implementations of functions but only light modifications to the original source. With each program NPE has slowly grown to cover a larger range of functions, and at this point covers a majority of the SDL2 surface area.

Discussions are being had to consider NPE for inclusion with 9front. APE's footprint within 9front has become smaller and smaller as contributed native variants have gradually replaced APE-only programs. As such, reimplementing existing APE programs using only NPE is now easily within reach. Further work and expansion of NPE is still required in order for this to succeed, but surveys of the remaining software do not indicate anything preventing the core NPE design from being sufficient. This transition would cut down on the maintance burden of these programs, allow for greater reuse of code between both sets of programs, and would overall be a reduction in complexity.

References

[Atlidakis16] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, Jason Nieh ''POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing'', *EuroSys 2016* London, April, 2016.

[Open18] The Open Group, ''The Open Group Base Specifications Issue 7'', *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* https://pubs.opengroup.org/onlinepubs/9699919799/

[TrickeyApe] Howard Trickey, ''APE - The ANSI/POSIX Environment''

[SDL] https://www.libsdl.org/

[GTK] https://www.gtk.org/

[QT] QT Group https://www.qt.io/

[UXN] Hundred Rabbits https://100r.co/site/uxn.html

Neoventi

Noam Preil

<noam@pixelhero.dev>

ABSTRACT

`neoventi` is a forwards-looking backwards-compatible reimplementation of the venti disk storage system, intended to address shortcomings in the prototype system while retaining the useful properties it revealed. A further benefit of the creation of a secondary implementation is the development of a better understanding of the original prototype, enabling further refinements to the original project.

Background

The venti disk system is a networked archival data storage system, which can be most effectively understood as a disk-resident write-once hash table, in which keys must always be the hash of their corresponding value [1]. Venti is intended to be one piece of a larger system, useful as the backbone for backup systems and snapshotting file systems [1]. Fossil is an archival file system built on top of venti, originally intended to supplant the Plan 9 file server [5].

The venti disk system has interesting properties that enable unusual features, largely borne out of simple but niche design choices, such as hash-based addressing and an append-only primary data log. Unfortunately, venti is also explicitly a prototype [1] and has both design and implementation limitations that cannot be easily addressed [2].

Further, many of the issues that plague the venti prototype's code base are complicated and entangled. The process of writing an implementation intended, from the beginning, to *not* be merely a prototype inherently lends itself to a better understanding of the design of the original prototype and, thus, of how to fix the issues. Multiple bugs in venti have been fixed as a result of the neoventi work; rewriting the code does not need to mean abandoning the original but can instead complement it. Having divergent implementations provides a practical method of iterative improvement: testing them against each other.

From a client's perspective, the venti system is straightforward. A trivial binary protocol is implemented over TCP. Variable-sized blocks written using the protocol yield an address as a hash of the block's contents; the block may then be read back using its hash. This simple design yields a system with interesting properties, including idempotent writes, immutable blocks, and implicit corruption detection [1].

That client's model is inadequate for understanding a venti server, however, as implied by venti's man pages.

```
BUGS
     Setting up a venti server is too complicated.
```

[3]. Setting up a venti server is complicated not merely because the tooling for doing so is inadequate – although this is certainly true – but because the venti server itself is complicated, and the primary inadequacy of the tooling is in failing to protect the operator from this complexity.

In many places, the venti documentation refers to "convention" with regards to how data is stored [4]. This includes the process by which large files and directories are stored in venti and the way files are stored in venti. The documentation is misleading; many of the conventions are hardcoded into various parts of the venti tooling as assumptions, and violating the assumptions can cause problems up to and including data loss.

For instance, venti blocks are not merely data. They are associated as well with *type tags,* which the documentation claims is done by venti [4]. This is contradicted by the documentation of the network protocol, which notes that each write operation must include a type tag, which venti stores. This mismatch of assumptions results in missed opportunities for useful tooling and, if the client puts in a tag that the data verification tooling thinks is "wrong," it may lead to the tool marking the block as invalid and rendering it unreadable, as can be seen in the `venti/syncarena` code:

```
if(vttypevalid(cl.info.type) < 0){
        ...
        broken = 1;
}
if(broken && fix){
        cl.info.type = VtCorruptType;
        if(writeclumphead(arena, aa, &cl) < 0){
                ...
        }
}
```

[6].

Additionally, each block is tagged with the ID of the user who first wrote it, which has privacy and security implications; security in general is a major shortcoming of venti. The protocol was designed to support authentication and encryption but neither the server nor any clients implement this support in any way, and the documentation outright acknowledges that the system was designed so that it *could* be secure [4].

The implementation has also been plagued with a long string of bugs. Most known bugs have been fixed, but not all. Attempts have been made to fix some of the remaining issues, but the code base is complicated and entangled.

Fossil has historically been even less reliable than venti, and there are multiple known serious bugs that have never been fixed.

In the time since fossil was originally conceived, the Plan 9 file server has indeed been supplanted, but not by fossil. The current predominant file systems in use are the Plan 9 world are cwfs and hjfs [7], both of which run in user space. In fact, the file server / auth server / cpu server / terminal distinction has largely been erased, as there is now a unified kernel that can act in any desired configuration [8]. There have been other file systems proposed, as well, all of which run in user space [9] [10].

However, largely for reliability reasons, fossil has since been supplanted as well. While there are a few remaining users of the system [11], most have abandoned it, and for good reason: for a long time, there were serious data loss / corruption bugs!

Despite the flaws, the venti system possesses many useful and interesting properties that are worth preserving, which motivates the redesign and implementation of a successor system. Some of these features are shared with other file systems (most notably Plan 9 file systems, due to their shared histories), and others are unique to the venti system, or to the venti/fossil combination.

Many of the flaws have also been slowly improved over the decades. Fossil supposedly had one major data loss/corruption issue that bothered many of its users, which was reportedly fixed around the early 2010s, though confirmation of this is tricky [13]. There are few users today, but we do exist, and these issues seem largely gone.

On the other hand, there continue to be many known bugs, from outstanding locking issues to bad behavior when fossil runs out of disk space to tooling claiming there are multiple PiB of free space on a file system sized under 10GiB [14].

The idea of writing neoventi arose slowly during my work on the venti system. Some of the bugs were simple enough to fix, but months of work and issues ranging from the usage of *signed integers* for memory sizes, and an alignment bug that would cause a crash if the cache was set to *exactly* 128MiB, and the discovery that over 1KLoC in libventi that were intended as a performance optimization actually made things *slower,* and the realization that a code cleanup would not work because one of four layers of entangled abstractions depended on undocumented behavior in another in subtle ways, I eventually got fed up enough that I no longer wished to work on the venti code base. I still wished to *use* a venti system, though, and there were features I wanted to experiment with adding to venti that I did not think could be reasonably added to the existing code base, so after months of wavering, I gave in and decided to rewrite venti from scratch.

neoventi

neoventi currently implements a read-only venti server in under 900 lines of code. It is fully compatible with the venti disk format, and has been tested on multiple existing ventis, including the one that backed the fossil it was running from. neoventi is, right now, intended primarily as a proof-of-concept, and a demonstration that venti can be simplified greatly from its current incarnation.

neoventi is largely written as a reaction against perceived shortcomings in venti. For instance, to process a read request, the venti server goes through more than five layers of abstraction: fcalls, RPCs, the Packet layer, Fragments, ZBlocks, and packing and conversion code to go between the layers. This complexity necessitates a lot of code to manage and copy data around. This is a burden for maintenance, and introduces a lot of subtle behavior.

By contrast, to implement the same code, neoventi deliberately goes as far in the opposite direction as possible, and does not use even a single layer of abstraction. This difference in approach can be seen easily in the code for establishing a connection with the client. Here is the code for reading the hello packet from the client in the original server:

```
if((p = vtrecv(z)) == nil)
        return -1;

if(vtfcallunpack(&tx, p) < 0){
        packetfree(p);
        return -1;
}
packetfree(p);
```

This code receives a packet as a Packet abstraction, and unpacks the request into a VtFcall, before freeing the raw packet. It also has error handling code mixed throughout, as errors are, in C convention, passed up the stack. For contrast, here is the equivalent code in neoventi:

```
vtrecv(conn, buf);
```

There's a few major differences. First, the rest of the routine for handling the hello request in neoventi operates on the incoming buffer directly. Where venti requires multiple memory copies and packet preprocessing, neoventi shuffles bytes around and slams them back over the network. Where venti constructs a Fcall response, and sets all of its fields, neoventi just reuses the same buffer into which the client send the request, and preserves the packet tag.

Secondly, neoventi uses stack allocation for buffers, and avoids heap allocations. This means that there is no need for constant memory allocations and frees, and is enabled by the first change.

Thirdly, neoventi uses a setjmp buffer for error handling, instead of passing errors up the stack. This is enabled in large part by the usage of stack allocation: the stack does not need to unwind, because there is nothing to clean up!

In practice, the venti protocol is not terribly complicated, and it is not difficult to get right - and bugs in packet processing are trivial to investigate and fix, where leaky abstractions and manual memory management and error handling are not. It is noteworthy that the entirety of neoventi needed for read-only operations occupies less than 900 lines of code, whereas venti's packet layer alone is over 1,000 lines of code, and it uses another 400 lines for the RPC/Fcall code. Venti needs significantly more code just to *read packets* than neoventi needs for an entire functional server!

neoventi's performance leaves much to be desired, in large part due to the total lack of caching. Nonetheless, it tests within 10% of venti's performance for cold reads, and that's despite the fact that manual investigation of the block structure being read suggests that blocks were being read from disk on average more than three times each.

neoventi's read pathway has been tested thoroughly. Testing was conducted using a fossil system root mounted via vacfs as a read-only root, a ramfs for a write buffer, and wbfs to deeply unionize them [16]. Tests included playback of multimedia, a full system compile of my Plan 9 branch, full scans of the git history and every version of every file ever stored in the distro, and streaming of game ROMs using Plan 9's built-in emulators [17]. No bugs were found, and performance was more than acceptable: notable results include 720p 30FPS video streaming through neoventi in real time with no stuttering [18].

Use Cases and Future Work

Venti/fossil is already a useful system. I have accomplished much with it that I would not have been able to with any other system I am aware of. During the last 9front hackathon, I was able to sync my laptop's venti with my file server 50 miles away in under fifteen seconds. This works only because syncing two ventis can be reduced to a copy of fully linear data. Data in venti exists independently of its physical address, and so can be copied to an arbitrary location on the destination machine. After syncing from one venti to another, one can note down the latest address in the source venti. The next sync is effectively just a copy of all data between the prior highest address and the current latest block - there is no need for any indirection or cleverness whatsoever.

Fossil's snapshotting mechanism also allows for a file system to be forked across machines and merged as with branches in a git repository. At present, this is entirely a manual process, and can be painful if the trees have diverged. In principle, however, git can be implemented directly on top of a venti+fossil system.

Further, it is possible to fully automate the synchronization of an arbitrary number of ventis using a peer-to-peer torrenting system. This would require engineering effort, but should not be terribly complicated; Plan 9 already has a native torrenter, and there's no need to reinvent the mechanisms here [19]. venti itself could be modified to do this in the background; multiple file systems can be automatically and seamlessly synced across arbitrary sets of machines.

Venti currently uses SHA1 for its hash, but SHA1 is no longer sufficient; there are real-world examples of SHA1 collisions. Ideally, it should be made trivial to swap the hash function for future-proofing, so that when the hash function which replaces SHA1 is itself broken, it should not be painful to swap the function again.

There are many security and privacy holes in venti's design that should be fixed. Anyone with access to a venti system has, in essence, full permissions to all data in every file system stored in the system. The only current practical mitigation is to simply prevent access to the venti in the first place, and only expose a file system on top of it, such as fossil, but this can greatly reduce the benefits of venti's deduplication.

neoventi is intended to fully replace venti for practical use, and to do so by providing a superset of venti's functionality, with none of its bugs, and better performance and security. Currently, the only one of these goals which is met is the lack of venti's bugs, but that's a ridiculous claim to make when neoventi does not currently even support accepting new data, which makes avoiding data loss or corruption a trivial matter!

The most pressing immediate work is to finish venti compatibility by adding in write support. After that, more directly useful projects, such as improvements to the disk formats, upgrading the hash function, and replacing the venti protocol with a 9p implementation, will be possible. Performance is a tertiary goal, though it will likely get some focus before new features are added since optimization is often a pleasurable endeavour.

Conclusion

neoventi provides a read-only drop-in replacement for the venti server, and meets its goal as a proof-of-concept: it demonstrates irrefutably that it is possible to implement a venti system with a fraction of the complexity used by the original implementation. neoventi needs more work to be able to advance beyond being simply a competiting prototype, but a fresh start makes it easier to do so than to clean up venti. The code is available [20], and feedback is welcomed.

References and Footnotes

[1] Sean Quinlan and Sean Dorward, ''Venti: A New Approach to Archival Storage,'' *Usenix Conference on File and Storage Technologies*, 2002.

[2] My only source here is personal experience.

[3] venti(8)

[4] venti(6)

[5] Sean Quinlan, Jim McKie, and Russ Cox, ''Fossil, an Archival File Server,'' /sys/doc/fossil.ms

[6] Venti source code, /sys/src/cmd/venti/srv/syncarena.c

[7] This is statistical: 9front is the most active branch of Plan 9 at the moment, and most 9front users use either cwfs or hjfs, both of which run as user-space programs.

[8] 9front's *Frequently Questioned Answers,* https://fqa.9front.org

[9] ''Good Enough File System'', http://shithub.us/ori/gefs/HEAD/info.html

[10] ''Another file system'', http://git.9front.org/plan9front/mafs/HEAD/info.html

[11] Source: I'm typing this paper on a machine that uses venti+fossil :)

[12] Sean Quinlan, ''A Cached WORM File System'', *Software – Practice and Experience,* December, 1991

[13] Evidence leans towards http://9legacy.org/9legacy/patch/fossil-deadlocks.diff being the fix, though the history here is muddled and it's really hard to find a meaningful history of 9legacy's patches; since fossil appears to have been removed from 9front before the fix was made (in large part as a *result* of the bug!), and the patches in both plan9port and 9legacy are unclear, it's hard to tell, and nobody seems interested in discussion. My one attempt at submitting a fix to 9legacy got no response; while my patch was accepted into 9legacy, I was not made aware of this and did not find out until I was looking through 9legacy's public information to try to piece together the history.

[14] These are all issues I have personally run into. I fixed a deadlock last year, but I've since run into another; I did not grab a trace at the time, and have not run into it again since.

[15] Patches can be found in the 9front tree, available via hjgit://git.9front.org/plan9front/plan9front.

[16] wbfs is a write-buffer file system, forked from kvik's unionfs. It takes two file systems, one read-only and the other read-write, and presents a unified image. Files in the read-only file system can be "modified" by writing an updated file to the write buffer file system, and further accesses will read back the instance from the write buffer. It is available at https://git.sr.ht/~pixelherodev/wbfs - it is not complete, but was sufficient for testing neoventi.

[17] See nintendo(1) for details on the emulators in question.

[18] In the interest of transparency, there was stuttering observed, but it was tested against a ramfs and determined to be a result of the CPU being unable to keep up with decoding and falling out of sync, and not related to neoventi at all.

[19] torrent(1).

[20] https://git.sr.ht/~pixelherodev/neoventi