# Pigeon Filesystem: A Novel Distributed Network Storage Solution for Extranet

*Lang J., Xiao T., Yue B.*
`jaytlang, mitimmy, byue`

Recitation Instructor: *Michael Cafarella*
Tutorial Instructor: *Michael Trice*

May 8, 2020

# Contents

# 1 Motivation

Our project's principal goal is to build a reliable, scalable, and efficient network storage subsystem for NASA's ExtraNet. This system must disseminate telemetry, management, and mission critical data in a distributed fashion across interplanetary satellites, and meet these goals as effectively as possible: Astronaut lives often depend on how quickly and reliably we can transmit data through the system. To this end, we present a system we dub the "Pigeon File System" which draws inspiration from the Venti archival storage system. It complements the already developed specifications for network routing and packet transmission by presenting a uniform, distributed file access API called the "Pigeon Protocol". PFS is optimal for this application because it enables reliability through a novel hash-addressing scheme, scalability through simplicity via a uniform, UNIX-like interface, and consistent speed via selective use/augmentation of the routing and transmission protocols.

# 2 Existing Subsystems

## 2.1 Hardware

We are provided with an existing satellite communications structure with some built-in storage. Communications are done optically and, in combination with the Routing Protocol, leverage NASA's Bundle Protocol for the packetization of data. The existing hardware along with their respective storage size for such a communications structure is shown in Figure 1 below:

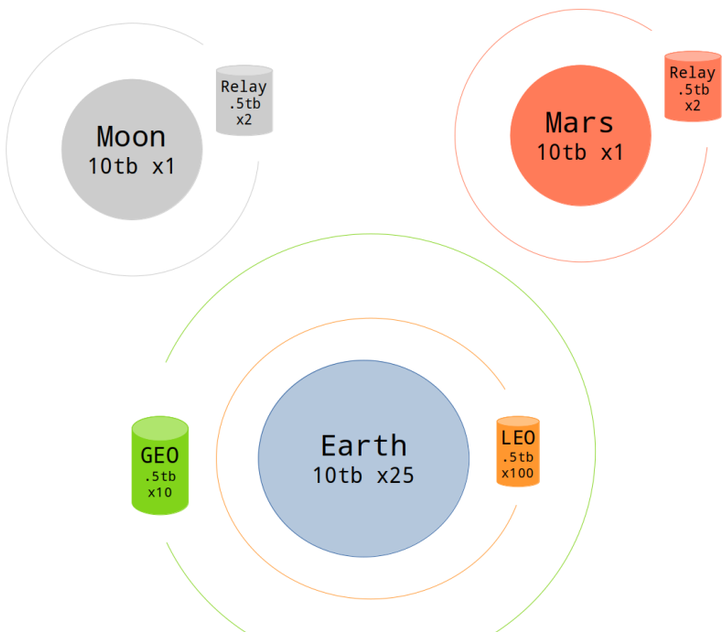


*Figure 1: ExtraNet hardware: planet/moon-based antennae, LEO satellites, GEO satellites, and relays. Data travels through concentric rings through the system when able.*

The separation of these systems from PFS enforces modularity and in turn enables scalability.

## 2.2 Bundle Protocol

The bundle protocol (BP) is a protocol used to packetize and transmit data over a delay-tolerant network. It utilizes a store-and-forward architecture, placing received application data "bundles" into a queue and forwarding them to the next node when possible. Additionally, NASA has provided several optional blocks to include in each bundle protocol transmission. Thus, to order this queue and guarantee reliable transportation, we propose augmenting each bundle with a priority block.

We propose implementing three tiers of message-type-dependent priority for the bundle transmission queue. Within each tier, bundles will be sent ordered by least recent arrival time. If two bundles arrive at the same time, they will be sent based upon which node is the farthest away - this ensures the most efficient transmission possible of important messages to isolated destinations. Finally, if two bundles arrive within the same priority tier for the same destination simultaneously, send order will be determined with respect to MD5 hash of the bundle's contents. (see section 3).

## 2.3 Routing Protocol

Our implementation largely uses the Routing Protocol as-is: we reference a provided routing table to acquire information about proximal satellites along the shortest path to a destination, and queue bundles accordingly. The bundle protocol and routing protocol work by default to fragment packets across the shortest path(s) and move them as efficiently as possible.

Additionally, for messages PFS deems “high priority” in the bundle protocol queue, we can optionally duplicate the message across the top 3 fastest available routes and repeat until we get confirmation that the message reached its destination intact. This increases the reliability of messages reaching their destination. PFS enables efficient integration of duplicate received transmissions - a powerful property we discuss in section 3.

# 3 Pigeon File System (PFS)

## 3.1 Inspiration

PFS is a distributed filesystem inspired by the *Venti* archival storage system. Venti uses a SHA-1 content-based addressing scheme to remotely locate files on a permanent, Write-Once-Read-Many (WORM) file server. This system sees use in the Plan 9 operating system from Bell Labs, where it serves as a backend for potentially several distributed, user-facing filesystems. Called Fossils, these act as caches for Venti, writing back new files and user changes into Venti snapshots once per day.

As a consequence of this unique addressing scheme, Venti installations can be partitioned into pools called *arenas*, consisting of many geographically disparate hard disks, and files can be found and reread by Fossils via hash instantly, without querying an additional, master server. PFS brings this secondary idea from Venti to the forefront - building a content-addressed *writeable* filesystem based on geographically distinct parts without the need for centralization, using a globally agreed upon hashing scheme.

PFS and its series of RPCs (see Section 4) also draw significant inspiration from Plan 9’s ”more UNIX than UNIX” philosophy. Extending beyond the ”everything is a file” idea seen in most modern UNIX-like systems, Plan 9 uses network-transparent text-based file I/O to control all devices attached to the system. This enables novel features such as remote CPU clustering and the attachment of diskless machines to disparate file servers. PFS takes this idea and its resultant simplicity, and scales it up in terms of both network size and latency tolerance.

## 3.2 Overview

PFS uses MD5 hashing to address disk space throughout ExtraNet: the MD5 sum of a given block’s contents (its *score*) can be used to find its location in constant time. MD5 hashes are inexpensive to compute and carry a low chance of collision, even with respect to exabytes of data. Hash-based addressing implicitly guarantees reliability, enables rapid insertion, and prevents the duplication or destruction of data. Additionally, this representation enables a Plan 9 - like extension of UNIX’s “everything is a file” philosophy to all devices, which can be hashed by their identifier - including network devices and bounded buffers. Entire satellites can thus be controlled through the filesystem namespace alone, through the same text-based I/O used to control files. These interactions thus have the same reliability guarantees, and not inventing new interfaces for each device creates unparalleled simplicity.

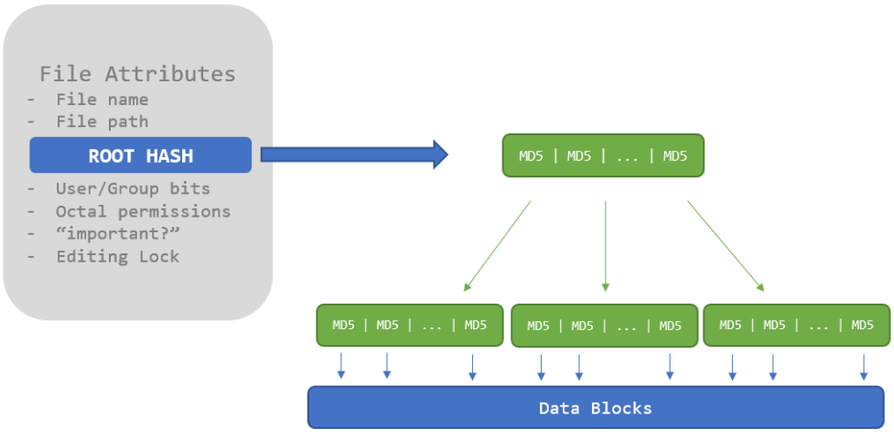


*Figure 2: This diagram illustrates how file attributes are referenced to compute the span of a file’s data on disk. MD5 hashes are referenced recursively in a Merkle Tree - like structure, which at the lowest level points to an arbitrary amount of data.*

## 3.3 Blocks

PFS uses blocks of 4 kilobytes each (the size of 256 MD5 hashes). A file is a sequence of contiguous scores, each pointing to a data block - shown in blue in Figure 2. This is a recursive data structure: since a file may consist of more than 256 data blocks, PFS uses an approach catered to scalability: arbitrary-height hash trees (similar to Merkle Trees in principle) can be built to address larger files, consisting of scores pointing to other score blocks as in Figure 2. Writes change the structure of this tree on-disk, creating necessary new branches and discarding old ones. The approach is efficient because it only writes branches in logarithmic time, and doesn’t face fragmentation issues typically seen with other filesystems.

Linking to the base of this tree is a *directory entry*, a file consisting of the hash tree's root score, as well as *attributes* including a lock for editing, absolute path, file type, UNIX-like RW/UGO bits enabling scalable access, and an "important" data flag - defined per the ExtraNet specification. When a file is written, its hash tree is mutated and the root score changes; this is updated in the directory entry. However, when a directory entry is re-scored, the root of the file's hash tree it is associated with is omitted - ensuring that edits to a file don't require propagation of changes up the entire namespace in logarithmic time.

This approach has a number of implications. For one, file duplication becomes obsolete within PFS, as identical hash trees produce identical writes. This way, duplicate messages can be used to enforce reliability implicitly without compromising efficiency. Moreover, because hashes are used to address blocks directly, this reliability can easily be guaranteed: a process can verify the root of the tree written to disk matches what it has manually scored itself. This prevents data corruption before it even arises.

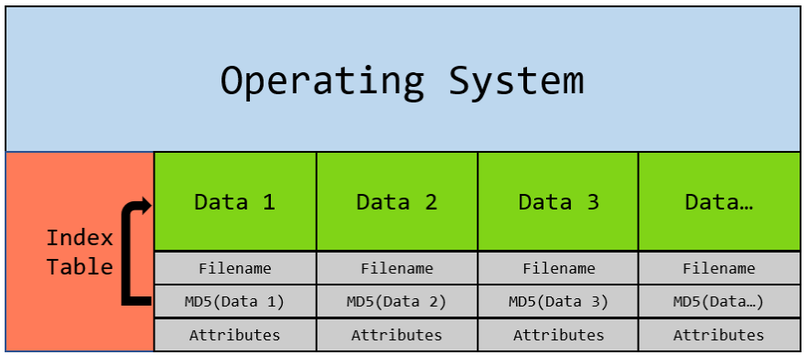


*Figure 3: The components of a PFS pool of are fairly simple: the overarching OS works in tandem with the addressing scheme and the index table to interchange a file's MD5 and data behind it. The gray blocks above represent directory entries - whose MD5 sums are dereferenced to provide access to data.*

### 3.4 File Pools

PFS supports multiple disks by the inclusion of file pools. Each file pool has size 0.5TB - equal to the minimum-sized satellite disk within ExtraNet. Assignment to a file pool is determined by running scores through a *pool hash function* - which partitions MD5 hashes and returns a pre-designated location per hash. This enables even distribution of files across the system and arbitrary scalability with the right hash function.

Additionally, the case of node failure can be mitigated by specifying more than one destination per partition within the pool hash function for 'important files' as defined within file attributes. This opens up the opportunity for backup nodes, where important files are double written, and which could be switched online in the event of data loss. In the event of partial data loss, where the disk is still functional, the disk can be completely restored from the backup image rather than being swapped out entirely. This capability serves as an efficient diagnostic as well, and in conjunction with the satellite's availability over the network (via routing protocol), can dictate next steps to restore this part of the system.

### 3.5 Indexing

Scores are translated to addresses via a procedure called indexing. Each satellite has its own *index table*, shown in orange in Figure 3 - a separate partition on disk from where data is stored. Given a file's location via the pool hash function, that location's index table is invoked to determine the address associated with the passed score.

In the event of index table corruption, the table can be reconstructed by iterating through disk blocks and recomputing scores. In order to support this feature, when a file is deleted the root of its hash tree is garbage collected immediately - ensuring it is never restored later. Should the satellite itself become unreachable, index table reconstruction may still occur, but backup pools would be necessary to get back important files. Thus, the system is recoverable and reliable against corruption via these means.

In all, using this sort of indexing with file pools creates a scalable, hierarchical management system that minimizes the number of ExtraNet queries, since file location can be determined instantly on disk. This is novel, and the lack of a need to query other satellites minimizes additional bandwidth overhead and promotes efficiency without compromising reliability.

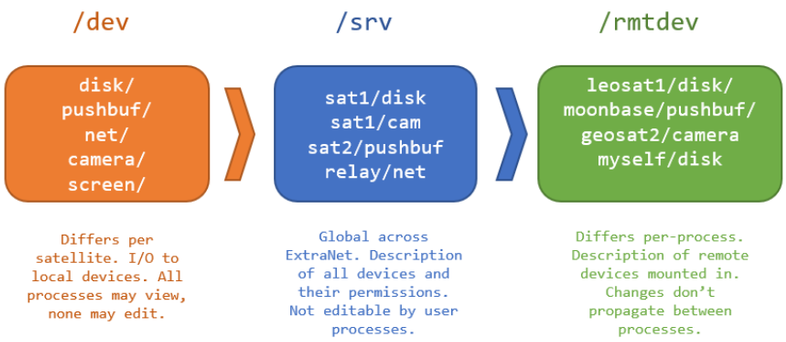


*Figure 4: A snapshot of part of the PFS root file tree. These three directories capture the differences between PFS and other UNIX-like filesystems: satellites post their devices into a global `srv` directory, which can be mounted by other devices in a network-transparent manner into `/rmtdev`. These devices act as subtrees within the filesystem, and accept I/O calls much like regular files do.*

### 3.6 Abstraction of Files

Similar to the UNIX family of operating systems, all devices and directories in PFS are abstracted as individual filesystems and interacted with via the same network-transparent, text-based protocol that files employ (see section 4). However, we diverge from UNIX in that this philosophy extends to networking devices - which along with other hardware are fully hashable and mountable from anywhere within a user's tree.

Whereas files are identified based on the hash of their contents, devices on every satellite are hashed by their serial number. When a PFS file service comes online, it mounts all of its local devices in `/dev`. and populates a globally shared service directory with hashes of each device coupled with its location, at `/srv`. Files posted in `srv` illustrate devices which can be bound into the local namespace using `mount` (see section 4). By convention, when mounted these remote devices are typically placed at `/rmtdev` to distinguish them from the necessary devices at `/dev`. Figure 4 shows an example of a layout for an abstraction of devices, with example names for each one.

### 3.7 Per-process Namespaces

The other feature of PFS that distinguishes it from typical UNIX involves us setting out name spaces for every process. Every process is able to control its own tree, which modularizes the process and imposes an implicit security mechanism: processes are only instantiated with a partial view of the filesystem tree, which can be controlled via an OS-level flag as they're forked off. Additionally, this synergizes well with the concept of devices-as-files: different processes may interact with different devices in different ways.

Like in UNIX, we set up a `/proc` namespace where every process lives and can be controlled - though unless specifically instantiated as otherwise, a process may not view other processes. Additionally, each process is the sole arbiter of the namespace under the local directory `/rmtdev`. This means that every process sees a different view of these directories, and has full control over the latter through the course of its lifespan. For instance, if two processes point the name `/rmtdev/hdd` to two distinct hard disks - or the same hard disk - there will be no collision as each process namespace is treated independently.

Because device filesystems can be implemented with virtual as well as physical devices, a common example of the former in motion is what we call the 'push buffer' - a bounded buffer in memory that can be used to notify PFS users of important, mission-critical information. This implements a device interface: writing to another satellite's buffer while mounted into `/rmtdev` has the effect of delivering the message to said satellite. Given many processes running on the same satellite, each may mount a different satellite's push buffer in the same location, and not interfere with each other's operation - guaranteeing consistent, simple message transmission.

### 3.8 Compatibility With Existing Systems

For purposes of scalability, PFS maintains backwards-compatibility with all already-launched satellites: it may be mounted to an existing filesystem from userland. In this case, PFS populates its own binary directory `/bin` with executables from the host filesystem, does device discovery for the directory `/dev`, and uses the newly mounted network device to connect to ExtraNet and set up as normal. System calls are passed through an intermediary API that could be implemented per-kernel - enabling the network-transparent abstractions we've made to persist across platforms.

### 3.9 Ownership and Users

PFS supports UNIX-like user definitions - RW and UGO bits are implemented to enforce permissions. This allows a scalable number of users to be registered efficiently via a well-tested medium.

### 3.10 Caching

For further efficiency, PFS uses a number of caches to accelerate data retrieval in delayed-access environments. Each satellite maintains a cache at the indexing level for its 'partition' of scores as determined by the pool hash function, allowing some I/O savings by skipping iteration through the hash table. Cache lines are flushed when the associated index table entry is changed or deleted.

Another efficient cache PFS uses is the block cache - which associates addresses with data. It is implemented on each satellite, and allows one to completely skip calls to the network if a score-data mapping is present. Lines are cleared upon writes, and much like the cache implemented in DNS, the cache is periodically flushed to prevent accidental incorrect reads.

### 3.11 Thoughts About PFS

In sum, PFS enforces reliable file access without compromising efficiency. It achieves modularity for individual devices and files themselves, and as we will discuss further, exposes a simple API for implementing scalable applications for both file-wide and device-wide interaction.

## 4 Pigeon Protocol

### 4.1 Overview

The Pigeon Protocol is the interface exposed by every PFS file server, and enables abstraction and interaction with the filesystem from any point connected to ExtraNet. The interface is composed of UNIX-like system calls, and is interfaced with through atomic transactions: PFS may only fulfill one Pigeon Protocol call at a time, and associates each transaction with a unique session ID tied to each participating file server. The system aims to build a simple, lightweight API; to implement scalable transactions; and to prioritize reliability via PFS scoring. This interface is illustrated below:

### 4.2 Hello and Goodbye

To begin or end any Pigeon Protocol session, we issue a `hello` call to begin session management, and allocate a new session ID for the coming transaction. Generally, PFS calls are grouped together by these transactions, and the contents of each are sent all at once via the Bundle Protocol. The receiver returns the session ID to the caller after the `hello` is received, along with returns from any subsequent calls, and the caller checks these values for integrity - storing the session ID if needed. If anything goes wrong, the client can simply terminate the old transaction and attempt a new one: any miswritten data will simply be redirected to a new root hash upon success, and misread data is discarded.

`goodbye` performs this transaction termination, deallocating session IDs and thus killing connections - not returning any value to the caller for efficiency. `hello` and `goodbye` in this sense are analagous to SQL's `begin` and `end` calls.

### 4.3 Read

When a client wants to retrieve a file within a session, we issue a `read(filename)` call on the relevant file object. At the caller, PFS checks attributes containing permissions about ownership and access, and requests this file from the relevant open session - determined earlier via pool hashing. It then recursively scores the newly received data and checks it against the recorded score. This effective three-way handshake (no ACK is sent to the file server after file data is received, though the score is checked) allows for rapid, efficient verification - we can flush relevant cache entries and retry the read call if this check fails.

### 4.4 Write

If a client would like to `write` to a file, we open a new session associated with the file's score. Notably, this score has changed if the file has been edited since an earlier transaction calling `read` - so it is written to a new location. Duplicate writes can be sent and will automatically be written once at the destination due to hash-based addressing. Transactions are aborted with notification early should the receiving device detect a hash mismatch, and re-initiated if the client notices a failure after this hash is returned. The latter approach exploits our ability to effectively double-write without consequence - additionally, writes can be batched multiple times without local impact to create a greater likelihood of success.

If `goodbye` can be considered the analogy to SQL's `end` then `write` can be considered the analogy to SQL's `commit`. For the future, this enables the potential for some sort of write-ahead log in order to preserve and recall state in the event of temporary system failure during a transaction.

### 4.5 Preventing Data Races: Open and Close

To prevent data races, we require an `open` call prior to permitting a write, which attempts to acquire the lock associated with a given file. `close` releases this lock and allows others to write to the file. `open` may also be used to designate a new file with the current user's permissions.

## 4.6 Mount/Unmount

`mount`[1] is used to mount files, devices, and directories to new locations within the PFS tree per-process. Provided both a location as well as the associated hash, potentially referenced from the `/srv` directory if we are mounting a device, `mount` references a given *mount point* within the PFS hierarchy and dereferences the hash of choice. Once this occurs, all PFS calls directed to or from the mounted device are treated with higher priority than pure file operations.

The `unmount` call is the opposite of mount: given a device's location, associated hash, and mount point, the mount point within the processor's namespace is dissociated from the device, file, or directory. Calling `goodbye` implicitly unmounts any connections associated with the killed session in `/rmtdev` or elsewhere.

## 4.7 Deletion

A `delete` call operates by unlinking a specific score from the responsible satellite's index table, and deleting the block referenced by the hash tree root on disk. This is efficient compared to garbage-collecting the entire file. `delete` returns the removed score for reliability as before.

## 4.8 Priority of Pigeon Protocol Calls

Pigeon Protocol calls are placed into the bundle protocol queue with uniquely assigned priority measures. Rather than utilizing per-file priority, we partition based on operation type to distribute use of the bundle protocol queue as effectively as possible.

In sum, operations coming from or directed at a mounted device are designated as high priority operations, followed by ACKs from existing non-device interactions, and then new non-device requests. Entire transactions are enqueued at once, their priority based on their highest constituent call, and are sent within singe continuous data streams. This scheme is evaluated in Section 6. Additionally, high priority calls can optionally be sent through the [up to] top 3 available routes to a destination, exploiting our scheme's ability to double-write or read to no local effect. Figure 5 summarizes this idea.

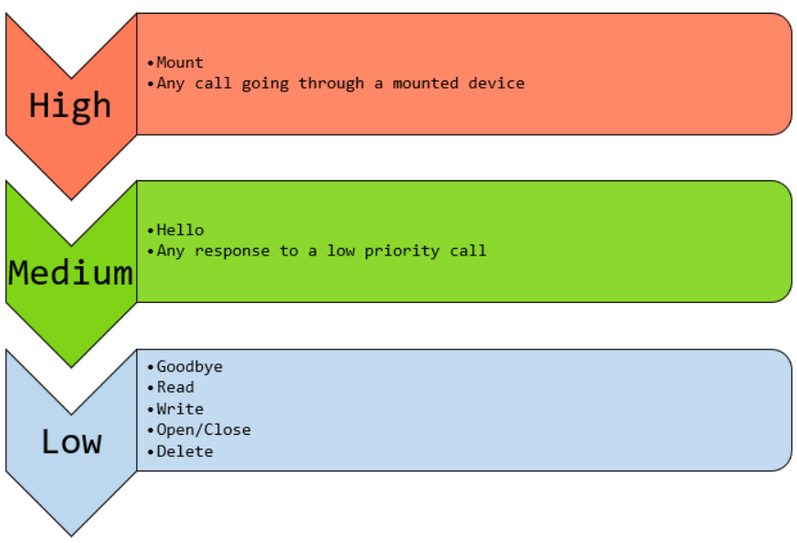


*Figure 5: The Pigeon Protocol's priority hierarchy. Demonstrates how transactions can be implicitly sorted as high, medium, or low priority by inherent importance of each atomic call. Entire transactions are inserted into the queue at once, and are given the same priority as their highest priority constituent call.*

## 4.9 Sending Data to a Predetermined Location

To perform guided data propagation, we have two different options tied to the data's priority. If the data is important, we `mount` the remote push buffer or other relevant devices (perhaps the remote `/dev/screen` - this is self-explanatory) into our tree. Subsequent writes will occur with high priority, exploiting the use of `mount`, and the user on the receiving end will be notified quickly. If the data is of low priority, a receiving user can simply read relevant data. Both of these may be used to stream data in addition to one-time reads: calls to special or regular files may occur continuously on loop. This is discussed further in section 5.

## 4.10 Thoughts about the Pigeon Protocol

In all, the Pigeon Protocol API is simple, and guarantees reliability through atomic transactions with efficiency as a central focus. Implicit hash-based accuracy guarantees don't compromise efficiency and eliminate the need to actively search for a file, while atomic transactions provide the ability to correct errors should something go wrong. The simplicity of this design is further realized through implementing a number of use cases, as shown below.

[1]mount replaces the deprecated push call from a previous iteration of our paper. This drastic change in the Pigeon Protocol originally arose from the addition of mount to account for streaming, but we quickly realized that this call could serve the same function as push and thus push was deprecated to make the interface simpler and easier to use.

# 5 Use Cases

We summarize potential use cases by providing sample implementations via the Pigeon Protocol. Due to the flexibility of PFS, much of this work is done at the local application layer and incurs little bandwidth penalty on ExtraNet itself. In addition, errors are caught as they occur, and by scoring, identical messages sent across different routes can be double-written without duplicating data. This underscores the goal of reliability and allows it to be implemented succinctly.

## 5.1 Routine Communication

1. *Non-critical messages.* NASA requires connection with 2 Moon and 2 Mars nodes. Non-critical messages are sent to these stations within reasonable time frames and require accuracy. A simplistic UNIX-like implementation for this would involve creating an inbox for each astronaut and writing to it as follows:

```
// craft message here...

char dest_path[] = "/mail/astronaut/inbox/msg.txt"
int score = do_score(message);
int session_id = hello(score);
while(open(dest_path) != 0);
while(write(dest_path, data) != score);
close(dest_path);
goodbye(score);
```

2. *Critical messages* Critical messages interlink the connected stations as non-critical messages do, but must be delivered within a specific time frame and require full accuracy. Examples of this include Ground Control advising an astronaut about incoming space debris or changes to the mission plan. We advise utilizing the push buffer (and by extension PFS's data duplication) to perform this efficiently and reliably:

```
// craft message here and define dest

int score = do_score("/srv/dest/pushbuf");
int session_id = hello(score);
mount("/srv/dest/pushbuf", "/rmtdev/pushbuf");
while(write("/rmtdev/pushbuf", data) != score);
goodbye(score);
```

## 5.2 Telemetry and Mission Data

1. *Earth Images* These are sent from GEO satellites to Earth through LEO satellites. These aren't particularly mission-critical - we write these to the filesystem normally for later viewing.
2. *Solar Telemetry* These include solar energy data sent to Earth for flare predictions. These are sometimes important, chiefly when the satellite calculates a high degree of urgency, but not always.

The following implementation is proposed for all telemetry:

```
// load data and compare urgency right off the bat

int score = do_score("/srv/dest/pushbuf");
int session_id = hello(do_score(data));
if(evaluate(data) == URGENT){
    mount("/srv/dest/pushbuf", "/rmtdev/pushbuf");
    while(write("/rmtdev/pushbuf", data) != score);
    goodbye(score);
    return;
}
else{
    open(dest);
    write(data, dest, accuracy_require = 0);
    close(dest);
goodbye(score);
```

## 5.3 Large Correct Transmissions (LCTs)

PFS can guarantee correctness for large files, and our augmentation to the routing protocol alongside the BP reduces associated bandwidth usage without compromising speed. The primary LCTs will be system updates: we assume these will not affect our protocol beyond patches that preserve original functionality. For these, rather than utilizing the push buffer we may be able to mount the hard drive directly, writing changes to the kernel and bypassing user data.

```
int score = do_score("/srv/dest/hdd"));
int session_id = hello(score);

mount("/srv/dest/hdd", "/rmtdev/hdd");
while(write("/rmtdev/hdd/os", data) != score);
goodbye(score);
```

### 5.4 Streaming Data

PFS allows us to stream data between devices via repeated `read` calls, either through a device or a file dependent on priority / use case. A sample implementation illustrates a remote mount of a desired screen, and the creation of a new window using a PFS-assisted window manager. This could be implemented on top of the host filesystem and would abstract graphics devices as files, enabling the same textual I/O used for everything else. Given appropriate permissions, this operation could be performed to stream high-priority video-encoded data:

```
int windowid, score, session_id;
char mywindow[100];
char windowstatus[100];
char ctlfile[100];

score = do_score("/srv/dest/screen");
session_id = hello(score);

mount("/srv/dest/screen", "/rmtdev/screen");
windowid = read("/rmtdev/screen/allocwindow");
sprintf(mywindow, "%s%d", "/rmtdev/screen/window/", windowid);
sprintf(windowstatus, "%s%s", mywindow, "/status");
sprintf(ctlfile, "%s%s", mywindow, "/ctl");

while(write(ctlfile, "windowmode = FULLSCREENVIDEO") != 0);

while(1){
    while(write(ctlfile, data) != score(data));
    if(read(windowstatus) < 0) break;
}

goodbye(score);
```

## 6 Evaluation

### 6.1 Overview

We evaluate ExtraNet against a number of competing, more traditional schemes that could be used to construct an effective distributed filesystem - given specifications provided within the problem description. Qualitatively, it's a useful exercise to compare the traditional UNIX namespace with that offered by PFS, demonstrating the advantages and drawbacks of UNIX's simple, well-received design with our differing model. Another useful comparison exists between PFS's implicit end-to-end MD5 and the default administrative per-node verification present within the BP. Finally, distributed hash functions exist that differ from our fixed implementation, such as a solution called Chord. This could present an alternative to our fixed hashing and make adding satellites easier; it's worth discussing the tradeoffs associated with each.

Quantitatively, we paint a picture of PFS's bandwidth consumption. We considered a relatively average case defined within the specification, because we wanted to ensure our system performs well under relatively "normal" circumstances. For brevity, rather than split the worst case into several scenarios emphasizing several properties, we combined every relevant worst case into one catastrophic scenario to show that system can handle everything going wrong at once.

### 6.2 Quantitative Analysis

**Guiding Assumptions** In the ExtraNet specification, we are provided a number of useful statistics about the hardware available within our system; we make a few necessary assumptions about this hardware to accurately assess our system's performance. First, we estimate an industry standard 5% failure rate for data transmission by the Bundle Protocol: in our system, this would manifest as a hash mismatch and require re-transmission of the data from the sender.

We also assumed that catastrophic solar events would have an average duration of 3 hours, once per week - and like non-critical data, operates under the assumption that 1 in 5 packets must complete transmission for accurate assessment to be possible. Additionally, we arbitrarily assume 5% of routine communications are critical at a given time.

Pigeon Protocol RPCs are tiny - and their size is largely dominated by the size of the MD5 hash sent with them. We made the conservative estimate that each RPC thus measures 32 bytes - a generous amount of space to contain identifying data.

Finally, unrelated to hardware, a fundamental assumption we make is that our system will never suffer a failure due to hash function collision, as the probability that two hashes collide is $(\frac{1}{2})^{128}$ - extremely unlikely.

**Bandwidth Consumption Under PFS** Thanks to its decentralized namespace and the simplicity of Pigeon Protocol RPCs, PFS consumes extremely little bandwidth over that required by the ExtraNet spec itself. Based on the implementations in Section 5, we calculated the precise overhead for each use case, excluding write calls as these are asymptotically smaller in size than the data sent with them. Results are presented in figure 6.

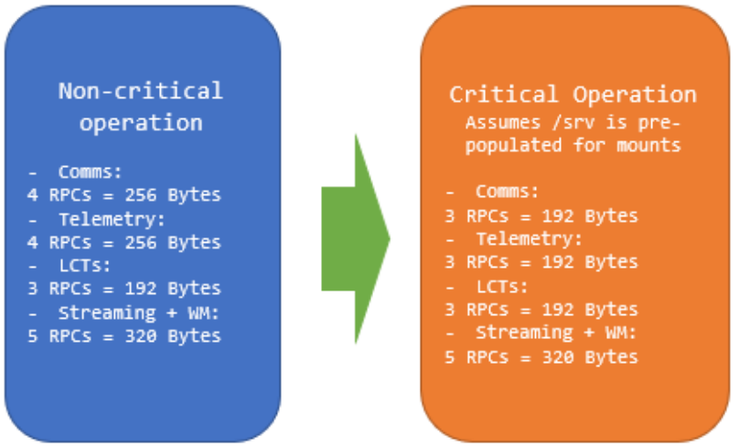


*Figure 6: Extraneous calls required to perform operations within each use case. Note that every Pigeon Protocol call requires 64 bytes of round-trip overhead, assuming a generous 32 byte RPC size. The streaming use case assumes the PFS-assisted Window Managemnt (WM) approach raised in section 5, and mounting operations assume the contents of the `/srv` directory are known.*

These numbers, in conjunction with those given in the specification, enable us to optimistically compute the average bytes per second required for each application. We assume the average rate of routine communications provided in the paper with our 5% criticality assumption, our 3 hr/week critical solar event rate, and a rolling update schedule. The results of this are shown in Figure 7 below:

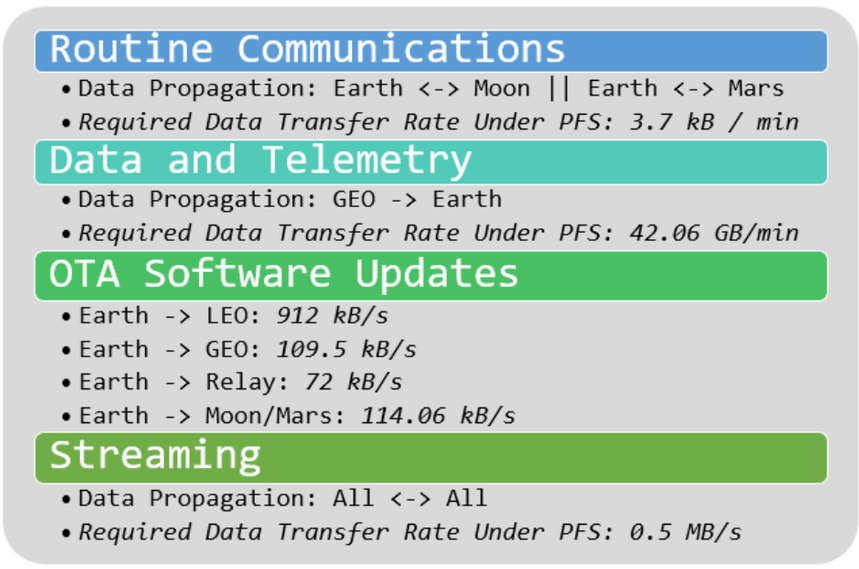


*Figure 7: Necessary data transfer rates per use case, including extraneous PFS RPCs. Once again, these numbers make the same assumptions as provided in Figure 6. Note that software update numbers are averaged as well - this will be addressed below.*

From Figure 7, we may compute the total data usage per link our solution requires - multiplying each by a factor of 1.05 to account for inaccuracies:

- Earth $\Longleftrightarrow$ LEO: 737.94 MB/s
- LEO $\Longleftrightarrow$ GEO: 736.9 MB/s

- GEO $\iff$ Relays: 830.02 kB/s
- Relay $\iff$ Relay: 830.02 kB/s
- Relay $\iff$ Remote Planet: 754.425 kB/s

Thanks to PFS's almost negligible bandwidth demands over raw message passing, no link is saturated under average use according to the ExtraNet spec. Furthermore, additional Pigeon Protocol calls to descend the directory tree as appropriate can also be tolerated without spilling over bandwidth limits.

**Worst Case Scenario** We've shown PFS can stand up to daily use, but a number of averages took place over these use cases - particularly over software updates and critical solar emergencies. To stretch PFS, we've created a worst-case scenario that could be encountered in production:

- Full 10 megabyte video streams are being sent between Mission Control and extraterrestrial antennae once per minute, simultaneously. Time is of the essence, so the application developers opted to use a triple write + PFS deduplication rather than send once and wait for a hash mismatch.
- A solar flare is happening and any GEO satellite with a link to the ground through LEO beams down data through the high priority queue.
- Concurrent with the solar flare, all satellites in the system are to sent update data as quickly as possible.
- Data streaming continues between all satellites in ExtraNet as normal.

PFS can handle this load surprisingly well - according to numbers in the ExtraNet specification, updates can be marshalled to all nodes within 7 minutes even under this degree of load. We would utilize paths through all reachable geostationary satellites (when the LEO satellite is not at the poles; this is accounted for) and the full network of additional LEO satellites. Updates could be sent in reverse order, enabling Martian receivers to lock down and initiate the update well within the 30 minute expected time - and should a hash mismatch be detected the update can be re-sent along.

## 6.3 Additional Analysis and Competing Systems

**UNIX vs. PFS namespaces** The namespace presented by PFS is significantly different than its traditional, UNIX-like counterpart. Under PFS, all devices share text-based I/O interfaces like files do, and the namespace is customizable per-process. Not only is this more powerful than in UNIX-like filesystems, as discussed at length in Section 3, but it comes with the result that a single, global namespace is presented to the user rather than a fragmented one consisting of many mountpoints - as would be typical in a client-server model. This comes with dramatic ease of use benefits, and simplicity for application developers, without compromising reliability or efficiency thanks to hashing.

Additionally, PFS only leverages 9 RPCs, compared to countless calls offered in contemporary UNIX-based solutions. These commands can all even be invoked from the command line, exemplifying their ease of use and abstracting satellite-specific details from the user.

**PFS End-To-End Hashing vs. ACKing at all nodes** The BP utilizes an optional accounting system that checks bundle accuracy at all nodes along a path, which we don't use in favor of scoring at the application layer. We argue this trade-off is superior for our design goals: the BP implementation requires significant additional bandwidth to detect potential failures earlier.

Furthermore, we contend that the benefits of early failure detection are hardly applicable to the design at hand. A system with multiple large hops might benefit from node-to-node ACK'ing of bundles, in order to reduce time required to communicate errors. However, in the existing system only one significantly large hop (Earth/Moon to Mars) presents problems for the timescales required by our specification. Additionally, node-to-node ACK'ing may scale poorly as the number of nodes increases, and may clog bandwidth and pose problems even with multiple large hops. Thus, we find that node-to-node ACK'ing is not in line with our goals of scalability and efficiency.

**Chord as a Hashing Scheme** Chord is a system designed by MIT engineers to be a distributed hash table. We briefly considered employing Chord as an alternative hashing scheme after a recommendation from Professor Michael Cafarella, but ultimately we decided against it. Chord allows for the use of satellites as keys in the hash table, however, it forces us to hop through at most $O(\log n)$ satellites to lookup a key, whereas our hashing scheme allows $O(1)$ networkless lookups. PFS's main drawback compared to Chord is that adding satellites is costly - this tradeoff is perceived to be worthwhile and discussed at length in section 7.

# 7 Issues and Tradeoffs

## 7.1 Within the Specification At Hand

**Walking the Namespace** Another tradeoff of latency for a lightweight file system manifests in delays moving through the namespace. Since walking the tree requires querying for successive directory entries, if these aren't in the cache at query-time we may experience inordinate latency. We recommend implementations with a flatter directory tree to avoid this.

**Node Failure** If a pool fails, and its backup pool also fails, we have no further data redundancy measures to recover lost data beyond the index table. PFS will thus consistently return errors when attempting to write or read data on these pools, and eventually may ignore them altogether via reconfiguration of pool hashing. We do not anticipate failure of backup pools to be a large issue mainly due to the fact that they are located on Earth and thus have more access to maintenance.

**Congestion** We do not have a solution to resolve issues when ExtraNet is congested. However, under heavy traffic, the bundle protocol will prioritize moving important messages via device mounts per the Pigeon Protocol specification. Additionally, when all PFS fileservers are at full capacity and all files in storage are important, we do not have a concrete way to pick files to replace. It's worth noting that UNIX-like systems today don't propose an effective solution for this, either - but thanks to our "important" flag defined in file attributes, it's possible to arbitrarily unlink older unimportant files in a bid to keep important records flowing in the event of disk saturation.

## 7.2 Future Ideas Beyond the Specification

**Double-writes and latency** By choosing to implement a simplistic, duplication-free distributed file system, we inherently trade latency over long distances for increased efficiency and usability. We contend that the resultant potential for races is only an issue for PFS in very use specific cases where machines attempt to write to the same file concurrently.

Though we don't believe file editing will occur frequently given our use cases, our system is able to resolve this issue via a previously-discussed file locking mechanism. We suggest file editors be implemented by first batching a read call with an open call on the file, locking it for the user to edit freely, then batching a later write call with a close call. This prevents race conditions at the cost of some speed. If open is not used immediately (allowing another program to read it from the block cache), or if a file is deleted and then re-retrieved from another machine's block cache, it can be opened/written to and will remain resident in PFS.

**Moving files** Manually relocating a file to a different satellite within ExtraNet using a stock PFS setup is impossible, as file location is solely determined by its hash. Ad-hoc redundancy can be achieved by appending arbitrary escape characters to the end of a file - thus changing its hash - but otherwise, copies of a file would sum to the same root score and would be automatically de-duplicated. While this approach is strong for network resiliency (as discussed, writes can be sent multiple times to ensure their accuracy with little computational penalty), redundancy on disk is limited to the use of back-up pools.

Typically, storing the same file in multiple specific locations is not required of our provided use cases. A potential solution, of course, is to directly mount a desired storage device from `/srv` and `write` data to it - circumventing pool hashing. However, this should not be performed unless absolutely unnecessary: our system distributes files evenly across disks, ensuring full use of the filesystem, and if the balance is upset we could overflow certain disks before others. Additionally, files written in this way won't be hashable by the traditional index table - this functionality should be reserved for one-time messages and/or OS updates.

**Performance of append operations** Certain applications of ExtraNet may require repeated append operations to a given file. PFS would re-hash a file upon every single operation, and would likely repeatedly relocate this file should it be re-opened/closed on every append. Based on the use cases in the ExtraNet specification, there is no real case where this is necessary that we haven't already solved.

Should this become required, a possible approach is to use a virtual device like the push buffer to temporarily append data, then store it en masse when these operations have completed.

**Adding new satellites** Should we add new satellites into PFS, our pool hash function needs to be modified to evenly distribute files to all disks including the new device. This requires large-scale recomputation of file locations and flushing of the index table. This said, satellite deployments are typically few and far between, so this issue should not arise often.

Adding satellites is of course still possible under PFS, but it generally will require some downtime as index tables are re-populated and files are transferred across the system to the limits of the bandwidth given. This should not be done often, and device services can remain online as the transition is performed. In essence, we've traded scalability in the number of pools for raw efficiency in determining file location via a fixed hash. Since the ExtraNet does not address adding new satellites to begin with, implying the system is largely static, we consider this an unconventional but worthwhile trade.

### 7.3 Further Future Augmentations

Our design draws inspiration from an archival storage system - and thus allows us to simply build a variant of version control via existing backup pools. A means of implementing this would be to modify backup PFS servers to act as classic WORM partitions, which would record periodic snapshots of important files within ExtraNet using the existing pool hash function. This VCS would permit the restoration of filesystem snapshots at the cost of some bandwidth (perhaps implemented via device mount of the remote WORM) aiding reliability without breaking modularity and maintaining scalability.

## 8 Conclusion

In all, the design of PFS strives for reliability above all else, without sacrificing efficiency or scalability. Its hash-addressing scheme enables for transmission duplication without efficiency losses, while still supplying a familiar, scalable and easy-to-use interface to its users. Critical messages can be sprayed rapidly across the system, while verification of a message is performed end-to-end and doesn't overuse bandwidth. The system retains scalability via deterministic pool hashing and single-call indexing - in conjunction with per-process namespacing and device filesystems - allowing one to rapidly find a file or send data regardless of the network's size, and providing reliable distributed performance for a large volume of users.

## 9 Contributions

This paper could not have been written without Jay Lang's early concept and overarching design for PFS, Timmy Xiao's Pigeon Protocol and thorough system analysis, and Brandon Yue's grasp of the ExtraNet specification and intuitive infographics.


## 10 Acknowledgements

We like to give special thanks to Michael Trice and Michael Cafarella for their wonderful feedback as we designed this system. We also like to thank Katrina LaCurts and the 6.033 course staff for the awesome semester we had, especially with the unique situation the class was taught under.


## References


[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishn. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, `https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf`

[2] Sean Quinlan, Sean Dorward. *Venti: a new approach to archival storage*, `https://www.usenix.org/legacy/publications/library/proceedings/fast02/quinlan/quinlan_html/index.html`