Send and Receive of File System Protocols: Userspace Approach With *puffs*

Antti Kantee <pooka@cs.hut.fi>
Helsinki University of Technology

ABSTRACT

A file system is a protocol translator: it interprets incoming requests and transforms them into a form suitable to store and retrieve data. In other words, a file system has the knowledge of how to convert abstract requests to concrete ones. The differences between how this request translation is handled for local and distributed file systems are multiple, yet both must present the same semantics to a user.

This paper discusses implementing distributed file system drivers as virtual file system clients in userspace using the Pass-to-Userspace Framework File System, *puffs*. The details of distributed file systems when compared to local file systems are identified, and implementation strategies for them are outlined along with discussion on where and how to optimize for maximal performance.

The design and implementation of an abstract framework for implementing distributed file systems on top of *puffs* is presented. Two distributed file system implementations are presented and evaluated: psshfs, which uses the ssh sftp protocol, and 9puffs, which uses the Plan9 9P resource sharing protocol. Additionally, the 4.4BSD portal file system and *puffs* user-kernel communication are implemented on top of the framework. The performance of userspace distributed file systems are evaluated against the in-kernel NFS client and they are measured to outperform NFS in some situations.

Keywords: distributed file systems, userspace file systems, software architecture

1. Introduction

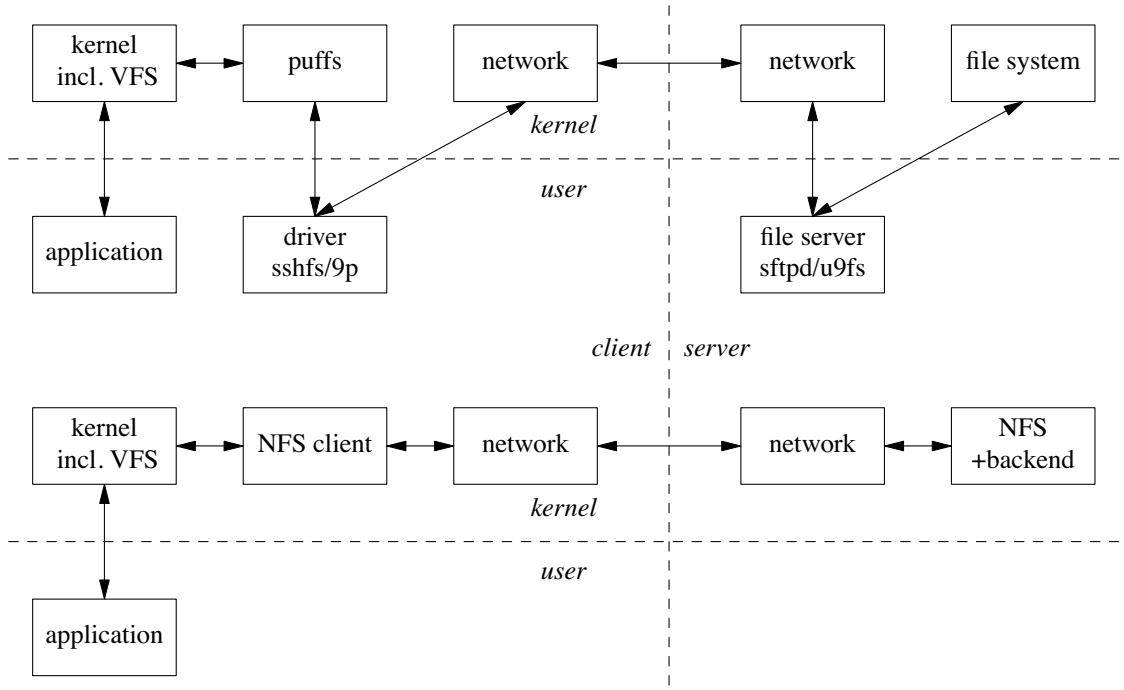
One taxonomy for file systems is based on where they serve data from:

- Fictional file systems serve a file system namespace and file data which is generated by the file server. Examples are *procfs* and *devfs*.
- Local file systems serve data which is located on the local machine on various types of media. Examples are *FFS*, *cdfs* and *tmpfs* for hard drive, CD and memory storage, respectively.
- Distributed file systems serve non-local data, typically accessed over a network. Examples are *NFS* [1] and *CIFS* [2].

A typical distributed file system will serve its data off of a local file system, but it is also free to serve it from a fictional file system, its own database or even another distributed file system.

Distributed file systems can be subdivided into two categories. In client-server type file systems all served data is retained on dedicated servers. The examples *NFS* and *CIFS* given earlier are examples of this kind of a file system. Peer-to-peer file systems treat all participants equally and all clients may also serve the file system's contents. Examples of peer-to-peer file systems are *ivy* [3] and *pastis* [4]. We concentrate on client-server systems, although all discussion is believed to apply to peer-to-peer systems as well.

Figure 1:
Structural comparison of userspace and kernel distributed file system drivers



Core operating system services such as file systems are historically implemented in the kernel for performance reasons. With ever-growing machine power, more and more services are being pushed out of the kernel into separate execution domains. This provides both improved reliability and an easier programming environment. The idea of abandoning a monolithic kernel itself is not new and has been around in systems research for a long time with operating systems such as Mach [5]. The idea has, however, recently gained interest especially in file systems because of the FUSE [6] userspace file system framework.

It is, however, incorrect to assume that a userspace file system implementation will solve all problems by itself. In fact, it is nothing more than pushing the problems of implementing a file system from one domain to another.

This paper explores implementing distributed file systems in userspace on NetBSD [7]. While details are about NetBSD, the ideas are believed to have wider usability. The attachment for file systems is provided by *puffs* [8]. The file systems interface already exported to userspace is not replaced for distributed file systems [9], but rather extended by building a framework upon it.

The following contributions are made:

- Explaining file system concepts relevant to implementing distributed file systems in userspace.
- Presenting the design and implementation of a framework for creating distributed file systems in userspace.

Two file systems have been implemented:

- **psshfs**: a version of the ssh file system written specifically to use the features of *puffs* to its maximum. As its backend, psshfs uses the ssh sftp [10] sub-protocol.
- **9puffs**: a file system client implementing the Plan9 9P [11] resource sharing protocol.

Unforeseen uses include:

- **portalfs**: the 4.4BSD portal file system
- **puffs**: by treating *puffs* itself as a peer-to-peer file system, the framework can be applied for transmitting requests from and to the kernel.

The remainder of this paper is organized as follows. Chapter 2 gives a very short overview of the concepts of *puffs* relevant to this paper. Chapter 3 presents an overview of what a file system is and points out key differences between local and

distributed file systems from an implementor's point of view. Chapter 4 presents a framework for implementing distributed file systems. Chapter 5 contains experimental results for the implementations presented in this paper. Chapter 6 provides conclusions and outlines future work.

2. Short Introduction to *puffs*

This section provides readers unfamiliar with *puffs* the necessary overview to be able to follow the paper. A more complete description of *puffs* can be found elsewhere [8,12].

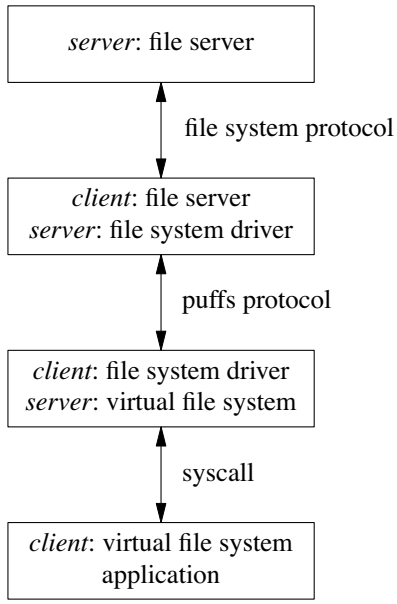
puffs is a framework for building file system drivers in userspace. It provides an interface similar to the kernel virtual file system interface, VFS [13], to a user process. *puffs* attaches itself to the kernel VFS layer. It passes requests it receives from the VFS interface in the kernel to userspace, waits for a result and provides the VFS caller with the result. Applications and the rest of the kernel outside of the VFS module cannot distinguish a file system implemented on top of *puffs* from a file system implemented purely in the kernel. Addtionally, the kernel part of *puffs* implements the necessary safeguards to make sure a malfunctioning or mischievous userspace component cannot affect the kernel adversely.

For the implementation of the file system in userspace a library, libpuffs, is provided. libpuffs not only supplies a programming interface to implement the file system on, but also includes convenience routines commonly required for implementing file systems. An example of such convenience functionality is the distributed file system framework described in this paper.

A file system driver registers a number of callbacks with libpuffs and requests the kernel to mount the file system. The operation of a file system driver is driven by its event loop, in which the file system receives requests, processes them and sends back a response. Typically, a file system driver will want to hand control over to `puffs_mainloop`() after initialization and have it take care of operation. For example, the file system drivers described in this paper hand control over to the mainloop. Nevertheless, it is possible for the file system also to retain control with itself if it so desires and dispatch incoming requests using routines provided by libpuffs.

Since distributed file system operations cannot usually be completed without waiting for a response from a server, it is beneficial to be able to have multiple outstanding operations. In most programs this is accomplished by threads or an event loop with explicitly stored state. *puffs* takes a different route: it provides cooperative multitasking as part of the framework and allows file system builders to schedule execution when needed. This, as opposed to using threads, means that the file system driver is never scheduled unexpectedly from its own point of view. Each execution context has its own stack and machine context, so yielding and continuing can be done with minimal programming involvement and without explicitly storing register and stack state.

Figure 2:
Multiple Levels of Client and Server



Every file system callback dispatched by the library has an associated execution context cookie, *puffs_cc*. This is used to yield execution by calling `puffs_cc_yield`(). Execution is resumed by calling `puffs_cc_continue`() on the same context cookie. The cookie may be passed around like any variable. It is invalidated once the request has been handled.

3. Structure of a Distributed File System

Distributed file system architecture along with this paper's terminology is presented in *Figure 2*. The term *file server* is used to describe the entity servicing the file system namespace and file contents using the file system protocol. The *file system driver* translates requests from the *kernel virtual file system* to the file server.

Distributed file systems operating over the network send out queries to the server to satisfy requests. Queries include an identification tag, which is used to pair responses from the server to issued requests. Of the file system protocols discussed in this paper, the Remote Procedure Call [14] mechanism used by NFS contains a transaction identifier, XID, ssh sftp [10] uses a 32bit request identifier and 9P [11] uses a 16bit tag.

The format of a message inside query frames is dependent on the file system protocol. However, at least the following operations, in some form or another, are common:

- open files to create file handles (and close file handles)
- read and write given file handle
- read the entries in a directory
- get and set the attributes of a file
- create and remove files, directories and special files

The discussion in the rest of this chapter applies to both the 9P and sftp protocols, although some of the mentioned features have been so far implemented only for psshfs. 4.4BSD NFS [15,16] is used for comparison in select places.

3.1. Network vs. Local Media

File system protocols commonly use TCP[1] for transport. A TCP connection is effectively a FIFO queue with latency and bandwidth characteristics. Once data is placed into the network socket, it will be transmitted in-order. This means that on a slow link with a large amount of data already in the buffer, it can take several seconds before anything inserted into the buffer will reach the peer. To take a concrete example, consider one thread doing a bulk read of a file and another thread doing `ls`. If several hundreds of kilobytes of incoming data has been requested and already queued into the socket by the server, it will take several seconds for the response to the directory read to reach the requesting end. Additionally, since reading a directory typically requires an EOF confirmation, it will take a minimum of two of these several second round trips. It is important to notice that after we send a request which causes the server to queue up large amounts of data, we cannot "unrequest" it any longer even though we might need the bandwidth for something more urgent in interactive use.

A local file system's media access is not as limited. Requests are queried and can be answered out-of-order depending on how the multiple layers from the disk scheduler to the driver and device itself see best. While large bulk transfers will slow down smaller requests such as a directory read, they will not necessarily completely stall them.

There are two approaches to dealing with this in distributed file systems:

Request throttling: do not allow one thread to issue requests saturating the pipe for several seconds. This is notable especially when the virtual memory system does read-ahead requests for large amounts of data. Since the purpose of read-ahead is make sure data is already locally cached when an application demands it, disabling read-ahead would cause application request latency. Ideally, the amount of read-ahead should be based on latency, available bandwidth and the total number of outstanding requests in the file system driver. As the heuristics to optimize this get complex fast, a much more pragmatic approach was taken: a command line option to limit the number of read-ahead requests per node. While far from perfect, this takes care of massive bursts and mitigates the problem for the most part.

Two separate channels: one for bulk data and one for metadata. Even though both connections share the same bandwidth, they will operate in parallel, and bulk transfers will not completely stall other requests. However, opening two TCP connections brings additional complications. First, we must authenticate twice. Second, all operations which create state must be duplicated for both channels, e.g. we must *walk* the file hierarchy for both connections with 9P. While in theory this option will provide better benefit, due to these complexities, it was not implemented – it is better to wait for the adaption SCTP [17] to solve the difficulties of multistreaming for us.

3.2. Distributed vs. Local File Systems

Some virtual file system operations are biased to the file system driver having direct access to the storage medium. This is not an issue for local file systems and also for distributed file systems specifically designed to inter-operate well with the virtual file system layer (e.g. NFS). However, all file system protocols (e.g. sftp) do not support the necessary functionality and must resort to alternative methods.

[1] NFS is transport-independent and has support for e.g. UDP transport, but as that is not applicable for remote sites, it is not discussed here.

This section discusses differences between distributed and local file systems and points out what is important to keep in mind when implementing a distributed file system driver in userspace. It also includes tips on increasing the performance of distributed file systems.

Permissions

Access control is not done in operations themselves, but rather using the *access* method. This presents problems for distributed file systems in several places: the typical I/O system calls (read, etc.) are not expected to return `EACCES`.

Some file system protocols do not present an opportunity to make access checks without making calls themselves. For example, with sftp we have no definitive idea in the file system driver of how our credentials map at the other end and therefore cannot do access checks purely by looking at the permission bits. The options are either to ignore the proper *access* method all together or execute shadow operations to check for access.

Luckily, in most of the cases applications deal well with returning `EACCES` from an I/O call, especially *read*/*write*. However, *readdir* is an exception and without implementing the *access* method properly, applications will only see an empty directory without any error message even if *readdir* returns an error. This is because `readdir`() is implemented in the system library and ignores permission errors from the `getdents`() system call. However, most file system protocols allow and require a directory to be opened for reading before fetching the contents. If the file system driver returns a permission error already when opening the directory for reading, the error is displayed properly in userspace.

Lookup

Lookup is the operation by which a file system converts a pathname component into an in-memory data structure to be used in future references to that file. This means that the file system should create an internal node for the file if found. In addition to a structural pointer, *puffs* requires three other pieces of information on the file:

- file type (regular file, directory, ...)
- file size (if a regular file)
- device number (if a device)

Typically the best strategy for implementing *lookup* in distributed file systems is doing *readdir* for the directory the *lookup* is done from and scanning the results locally.

Permissions also present an extra step for *lookup*. Lookup should return success for an entry which is inside an unreadable directory. To circumvent this, *lookup* can first attempt to read the directory, and if that fails, issue the equivalent of the protocol's *getattr* operation to check if the node exists.

It is possible to implement the *lookup* operation directly as a *getattr* operation, but it must be kept in mind that this will introduce an *n*latency* network penalty for looking up all the components in a directory, while doing a directory entry read once, caching the results and just scanning the locally cached copy introduces a much smaller cost.

While some file system protocols provide attributes for the files directly in the *readdir* return response, others might require extra effort such as real *getattr* operation. Next we discuss some optimizations possible in those cases.

The Unix long ls listing, `ls -l` is a fairly typical operation, which lists directory contents along with their attributes. Unless done right, this operation will also reduce performance down to *n*latency* because of waits for the *getattr* operations to complete and essentially doing nothing meanwhile.

While both 9P and sftp already supply attribute information as part of the *readdir* operation, an experimental version of psshfs was done to simulate a situation where it does not. This involved opportunistically firing off *getattr* queries for each of the directory entries found already during *readdir* and using cached values when *getattr* was issued to the file system driver. Two issues affecting performance were discovered and are listed here as potential pitfalls.

1. *readdir* operations generally require at least two round-trips for any file system protocol: one to deliver the results and a second one to deliver EOF. If *getattr* queries are queued or sent before parts 2-n of the *readdir* operation, the *getattr* requests are processed before the *readdir* operation completes. The file system driver will be waiting for the file server to process a lot of *getattr* operations to which the results are not needed yet. Therefore, the *getattr* operations should be fired off only after the *readdir* operation is completely done.
2. The attributes of the first file in the directory are requested from the file system driver after

readdir finishes; almost always before the results for the opportunistic *getattr* arrives from the file system server. If the file system driver discovers there is no cached result waiting and just fires off another query without checking if there is an outstanding request that should be waited for, all of the *getattr* requests targeted at later directory entries will be processed before the one we are currently after. Therefore, if an outstanding request is already active, it should be waited for instead of firing a new one.

Inactive

The *inactive* method for a file system node is called every time the kernel releases its last reference to a node. The purpose of *inactive* is to inform the file system that the node is no longer referenced by anything in the kernel and the file system may now free resources associated with the node. As, for example, executing the common command `ls -l` will issue an *inactive* for most of the files in the directory (all the ones without other references), *inactive* is an extremely common operation. However, typically a file system requires a call to *inactive* only in special cases, such as when a file is removed from the file system. Calling the *inactive* method in the kernel just costs a pointer indirection through the VFS layer and a function call, so it is cheap. When calling a userspace method the cost is much higher and should be avoided if possible.

The currently implemented solution to the cost problem is giving a file system the option for *inactive* to be called only when specifically requested. This is done with a *setback* operation. When the file system driver discovers the operation it is currently performing requires *inactive* to be called eventually, it issues an inactive *setback*. The means that a flag is piggy-backed on the request response to the kernel and set for the node structure in the kernel. In addition to incurring next to zero cost, the *setback* also solves problems with locking the kernel node – deadlocks could occur if we simply added a kernel call to flag this condition as we would be making it from the context of the file system driver. In case the *inactive* flag is not set for a node when the *inactive* kernel method is called, the request is simply short-circuited within the kernel and not transported to the userspace file system driver. For example, the *open* method may request inactive to be called for reasons explained in the next section.

Open Files and Stateful File Handles

Local file systems operate on local mass media and access file contents by directly accessing the media. Actual *read* and *write* operations, including their memory-mapped counterparts, do not perform access control. Access control is done earlier when a file descriptor is associated with the vnode. This means that as long as the file descriptor is kept open, the file can be accessed even though its permissions might change[2]. Local file systems do not open any file system level handles, as they can access the local disk at any time a request from above mandates they do so. The same applies to stateless versions of the NFS protocol.

However, most distributed file systems behave differently. For example, 9P and sftp require an explicit protocol level file handle for reading and writing files. These file handles must be opened and closed at the right times for the file system to operate correctly. For instance, assume that our file system driver opens a file with read/write access. Now our local system is guaranteed to be able to write to the file. Even if some other client accessing the file system changes the permissions of the file to read-only, our local system is still able to write to the file because of the open file handle.

Two different file handles are required for each file: one for reading and one for writing. If a file is opened read/write, it is possible to open only one handle. However, individual read and write handles must be opened separately, as the file's permissions might not allow both.

Opening handles for reading and writing is done when file opening is signaled to the file system by the *open* operation. It should be noted that this operation can be called when the node is already open. The file server should prefer to open only one handle if possible. It is possible to open a node only once for all users due to the credentials of the file handle being irrelevant; recall, access should be checked earlier. As some file servers and file system protocols might limit the amount of open file handles, the file handles should be closed once there are no users for the file on the local system.

On any modern operating system, file contents are accessed in two ways: either with

[2] The BSD kernel provides a routine called *revoke*(), which can be used to revoke open file handles. However, it is not typically used for regular files.

explicit read/write operations or through the virtual memory subsystem using memory mapped I/O. Regular I/O requires an open file descriptor associated with the file. However, memory mapped I/O can be performed without the file descriptor being open, as long as the mapping itself was done with the descriptor open. Even if the file is closed, it is still attached to the virtual memory subsystem in the kernel and therefore has a reference. Assuming there are no other references, once the virtual memory subsystem releases the file (due to `munmap`() or similar), *inactive* will be called. Therefore *inactive* is the right place to close file handles instead of the *close* method.

In the course of this work closing file handles in *close* was attempted. It included keeping count of how many times a file was opened in read-mode and how many times in write-mode. Also, the *mmap* method was changed to provide information about what type of mapping, read/write/execute, was being done so that the file system driver could keep track of it. However, the rules for determining if it was legal to close a file handle in *close* proved to be very convoluted. For instance, a file might have been closed less or more times than it was opened depending on the special circumstances. Also, as already noted, the only way a file system is notified of the virtual memory subsystem no longer using a file is *inactive*. The conclusion was to avoid *close* and prefer *inactive* unless there is a pressing reason to attempt to do otherwise.

Caching

Local file systems have exclusive access to the data on the file server. This means that every change goes through the file system driver. The same does not hold for distributed file systems and the contents can change on the file server through other file system drivers as well. This presents challenges in keeping the cache coherent, i.e. how to make sure we see the same contents as all other parties accessing the file server.

Currently, the *puffs* kernel virtual file system caches file contents (page cache) and name-to-vnode lookup information (name cache). The userspace file system driver, if it chooses to, caches the rest, such directory contents and file attributes. While caching in the userspace file system driver is less efficient, in practice the difference is minimal: compare the cost of network access to a peer with the cost of a local query to userspace. The important point is that caching in userspace allows a policy decision in the file system driver. Based on knowledge of the file system protocol, this can be made correct.

Neither sftp nor 9P support leases and therefore it is not possible to implement fully coherent caching. However, it is possible to add one based on timestamps and timeouts. Every time a file's attributes or a directory's contents are requested by the virtual file system, the current timestamp is compared against a stored one[3]. If the difference is smaller than the timeout value, the cached data is returned. Otherwise the file server is consulted and if a mismatch is found, the kernel virtual file system is requested to invalidate its cache: page cache for regular files and name cache for directories.

In addition to being able to specify a timeout value in seconds, it is also possible to make the cache always valid and never valid. It is important to note that an always invalid cache is not the same thing as no kernel caching at all. The system's ability to do memory mapped I/O and therefore execute files is based on the page cache. If we completely disable caching for a file system (mount with the *puffs* option `nocache`), we are no longer able to execute files off of it. By always invalidating the cache at our checkpoints, the cache is still frequently invalidated but MMIO is functional. However, for typical use a timeout of a few seconds is better even if data is frequently modified from under the file system driver. Finally, a user-assisted method of invalidating caches is provided: sending `SIGHUP` to the server invalidates all server and kernel caches.

As was mentioned above, the kernel caches file contents in the page cache. The page cache works in two ways: first, file content can be satisfied from the cache when read, and second, writes can be coalesced in memory and written to stable storage later to avoid lots of small I/O requests. The second case sometimes poses a problem for distributed file systems. For example, if copying a file to the mail server from where it is to be sent as an attachment, one expects it to be fully transferred after `cp` finishes. Instead the file might still reside completely in the local page cache. To avoid these kinds of situations, the *write through* cache mode for the puffs virtual file system is used: all writes are flushed immediately after they are done. Notably though, this does not cause modifications via memory mapped I/O to be flushed immediately. This can be solved by

[3] NFS checks timestamps also during file read.

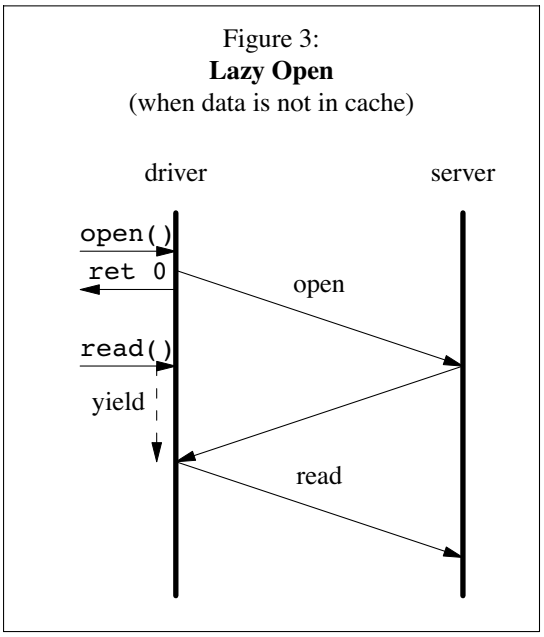


Figure 3:
Lazy Open
(when data is not in cache)

periodically issuing a flush request from the file system driver, but in practice there have not been any problems, so this has not been implemented.

Lazy Open

A typical operation sequence to read a file is *lookup*, *open*, *read*, *close*. The results for lookup and read can be cached at least to some degree as they are idempotent and we can make (user-assisted) assumptions about the stability of the data on the server. Open and close are different, as they change state on the file server and therefore cannot be cached. If the file content is cached locally, waiting for the opened file handle from the server is unnecessary, as it will not be used for serving data from the local cache. This can be a problem especially over slow links with hundreds of kilobytes of outstanding requests – it will take several seconds for the response from the server to be received.

This can be solved by lazily waiting for the file handle. The driver's open method sends a request to open a file handle, but returns immediately. Only if a read or write is actually issued, the file handle is waited for. This way data can be immediately served from the local cache if it is available. When implementing this scheme, care must be taken to handle open and close properly. The file might be closed (and reopened) before the original open request from the server returns, so state must be maintained to decide if a response to an open request should prompt closing the handle immediately.

Unix Open File Removal

An example user of *inactive* in the kernel is the Unix file removal semantics, which state that even after all links to a file are removed from the file system directory namespace, the file will continue to be valid as long as there are open references to it. A removed file will actually be removed only when *inactive* is called.

NFS client implementations on Unix systems feature the *silly rename* scheme, whereupon if a file is removed from a client host while it is still in use, the NFS client renames the file to a temporary name instead of deleting it. For example, 4.4BSD uses the name *.nfsAxxxx4.4* [16]. When the open file is finally closed, the *inactive* routine is called and the renamed file is removed. This scheme is due to the statelessness of the NFS protocol and has four problems.

1. If the client crashes between rename and the call to *inactive*, the renamed file is left dangling [1].
2. The file is still accessible through the file system namespace, although by a different name.
3. If another client removes the file, this scheme does not work.
4. Empty directories with silly renamed files are unremovable until the files have been closed.

A file handle's usefulness in dealing with the Unix open file semantics depends on file system protocol. In NFS, file handles are stateless; they are not explicitly opened and closed making it clear they cannot support this kind of behavior. The ssh sftp protocol uses file handles which are opened or closed, but the protocol specification [10] says that stateless or stateful operation is up to the server implementation. However, upon examination at least the OpenSSH sftpd supports the semantics we desire. The 9P protocol specification [11] leaves it open to the implementation and states that Plan 9 itself will not allow to access a removed file while implementations such as the Unix server *u9fs* will allow it.

While a file system driver should transparently support the semantics local to the system it runs on, such effort has not yet been made with *puffs* and the distributed file system drivers described in this document. They will work correctly with some servers and fail with others. As distributed and local semantics can never truly fully match, we do not consider this a big problem. If it is considered a problem, a *silly rename* scheme can be implemented.

4. Framework

Next, an abstract framework for implementing distributed file systems [18] is presented. The following properties of the framework are discussed:

- A buffering scheme for allocating memory for protocol data units (PDUs) and matching incoming buffers as responses to sent requests.
- Routines for cooperating multitasking, which handle scheduling automatically for file systems using the framework.
- An I/O descriptor subsystem, which allows to supply the framework with file descriptors used for data transfers.
- An event loop which reads incoming data from the I/O descriptors and the kernel, dispatches requests and writes outgoing data.

To use the framework, the file system driver must register callbacks which handle the driver semantics. An overview is presented here and each callback is later discussed in more detail.

`readframe`
: Read a complete frame from the network into the buffer provided by the framework.

`writeframe`
: Write a complete frame. A buffer given by the framework is used as the source for data.

`framecmp`
: Compare two frames to see if the one is the response to another.

`gotframe`
: Called for incoming frames which are determined to not be responses to outstanding requests.

`fdnotify`
: Notify the file system driver of changes the framework detected in I/O descriptor state.

4.1. Buffering

Sending and receiving traffic over the network requires buffers which host the contents of the protocol data units (PDUs). While the contents of a PDU are specific to the file system, the necessity of allocating and freeing memory for this purpose is generic.

For the purpose of memory management, *puffs* provides routines to store data in automatically resizing buffer: the puffs framebuffers [18], *puffs_framebuf*. In addition to automatic memory allocation, the buffering routines provide a read/write cursor, seeking ability, maximum written data offset and remaining size. When writing to a buffer it is possible to write as much data as there is available memory, but reading from the buffer will fail for locations beyond the maximum written data offset.

Additionally, the buffer supports opening a direct memory window to it. This is useful especially when reading or writing the buffer to or from the I/O file descriptor, because it avoids having to copy the data to a temporary buffer. As the framework does not know if data is being read or written in the window, the maximum size is also increased to the maximum mapped offset. Therefore, readers of the buffer should only map the buffer size's worth.

For processing the buffer contents, file systems typically want to add another layer which understands the contents of the buffer. For example, `fs_buf_write4`() would write 4 bytes of data into the buffer using *puffs_framebuf* routines after adjusting the byte order if necessary. Similarly, `fs_buf_readstr`() would read a string from the buffer using the protocol to determine the length of a string at the current cursor position. For example, for a protocol with "Pascal style" strings, the routine would first read n bytes to determine the string length and after that read the actual string data.

4.2. Multitasking

As mentioned in the *puffs* introduction earlier in Chapter 2, *puffs* implements its own multitasking mechanism without relying on platform thread scheduling. This means that in addition to not requiring any data structure synchronization calls in the file system driver[4], resource sharing can be better implemented and taken into account by the framework.

Commonly, threaded programs rely on implicit scheduling and contain local state in the stack. If a threaded program executes a blocking call, another thread is scheduled by the thread scheduler. The blocked thread is released when the blocking call completes. However, for distributed file systems the resource upon which blocking calls are made is the shared network connection, and therefore pure implicit state management will not do: received data must be mapped to the caller and additional state management is required. The *puffs framebuf* framework takes care of this state management and automatically

[4] Unless the driver chooses to create threads on its own, of course.

schedules execution where required, therefore making the task of file system implementation easier.

The points to suspend execution of a request are when a request is queued for network transmission. The framework automatically resumes the suspended request when the response has been read from the network. This functionality is discussed more later in the chapter "I/O Interface".

4.3. I/O File Descriptor Management

By default the framework is interested in the file descriptor which communicates *puffs* operations between the kernel and userspace. If the file system driver wishes the framework to listen to other descriptors, it must register descriptors using the `puffs_framev_addfd`() call. This can happen either when the file system driver is started or at any point during runtime. The prior is a likely scenario for client-server file systems after having contacted the file system server, while the latter applies with peer-to-peer file systems as new peers are discovered. Conversely, descriptors can be removed at any point during execution. This releases buffers associated with them, incoming and outgoing, and returns an error to blocked operations allowing them to run to completion.

I/O file descriptors have two modes: enabled and disabled. A disabled file descriptor will not produce any read or write events and therefore the callbacks will not get executed. The difference between disabling a descriptor and removing it is that disabling leaves the buffers associated with the I/O descriptor, incoming and outgoing, intact. This is useful for example in cases where the protocol has a separate data channel and the file system driver wishes to read data from it only when a VFS read request has been issued (see Chapter 4.6).

In addition to descriptor removal by the file system driver, the the framework must deal with abruptly closed connections. This means that it must provide the file system driver a notification when it detects an error condition with a descriptor. The `fdnotify`() callback is used for this purpose. As it is legal to half-close a file descriptor and still use the other side [19], the framework must track and notify the file system driver separately of the closing of either side. Similarly to file system driver initiated descriptor removal, the framework automatically releases all blocked waits and flags them with an error also in this case.

Once a descriptor is closed, certain conditions are imposed by the framework. It is not possible to write to a descriptor with the write side closed and attempting to do so immediately returns an error. However, if only the read side is closed, it is still possible to write to a file descriptor but waiting for the result is not allowed. The file system driver can further decide if this is a sensible condition in the `fdnotify`() callback. It also has the option of just giving up completely on a file descriptor when it receives the notification of either direction closing.

4.4. I/O Interface

Each file descriptor has its own send queue. A PDU can be queued for sending using four different routines:

- *enqueue_cc*: yield the current execution context until a response is received after which continue execution.
- *enqueue_cb*: do not yield. instead, a callback function, a pointer to which is given as a parameter, will be called from the eventloop context when the response is received.
- *enqueue_justsend*: just enqueue and do not yield. A parameter controls whether or not a response is expected. This is required to differentiate between a response and a request from the file server. However, the contents of the possible response are discarded.
- *enqueue_directsend*: yield until the buffer has been sent. Does not assume a response.

As the framework is completely protocol agnostic, it delegates the job of reading and writing frames to and from the descriptor to the file system driver via the `readframe`() and `writeframe`() callbacks. Readframe is called for incoming data while writeframe is used to transmit buffers in the send queues. As these routines are in the file system driver and can examine the buffer contents, they also know when a complete PDU was received or written. They signal this information back to the framework.

Of the above enqueueing routines, the first three require the ability to match an incoming response to a request sent earlier. The `framecmp`() callback provided by the file system driver is used for this. Once a complete frame has been read from the network, all the outstanding requests for the descriptor the frame was read

from are iterated over. As requests typically arrive in-order and even for a very busy file system the maximum number of outstanding requests is typically tens, the linear scan is cheap. Once the original request for the newly arrived response is located, execution is resumed.

If no matching request for the frame is found, the `gotframe`() callback is called. If the callback does not exist, the frame is dropped.

In case the comparison routine can determine from the incoming frame under examination that it is not a response at all, it can set a flag to short-circuit the iteration. This avoids going through all outstanding requests in cases where it is evident that the incoming frame is a request from the server and not a response to any of the file system driver's requests.

4.5. Event loop

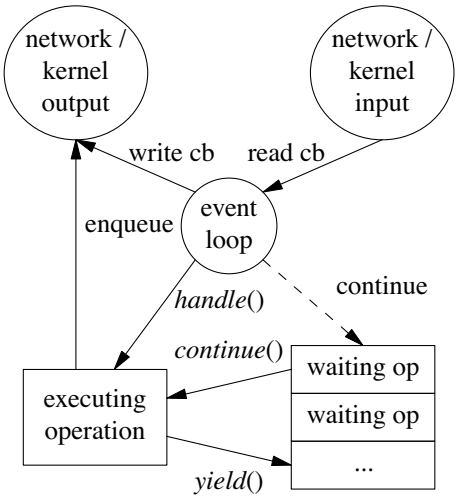
Finally, the event loop is discussed. It is the driving force behind file system driver operation and dispatches handlers for requests and responses as they come in.

The event loop, `puffs_mainloop`(), provided by libpuffs is a generalized version of the event loop first used directly in the psshfs file system [8]. Initially psshfs and 9puffs had their own event loops due to the standard libpuffs event loop lacking the features to support a distributed file system. However, as the framework was created the event loop was enhanced so that distributed file systems can use it. This new version is presented as a diagram in *Figure 4* and as pseudocode in *Figure 5*.

Each file system driver can specify a "loop function". This is a simple callback which is called once every loop. Single-threaded file servers can use it for tasks which need to be executed periodically. If the loop function needs periodical execution, maximum blocking time for the async I/O multiplexor in the event loop can be set using `puffs_ml_settimeout`(). While not realtime quality, this timeout is fairly accurate for correctly implemented file system drivers until very high loads.

For enabled descriptors, read polling is always active. Write polling is enabled only selectively, as otherwise the write event would always trigger. Its enable and disable in the event loop depend on the previous status and if there is data in the queue. Also, since the common case is that all enqueued data can be written immediately, the event loop attempts to write enqueued data

Figure 4:
Event Loop



before enabling write polling for a certain descriptor. Only if all data cannot be written, write polling is enabled.

The I/O multiplexor `kevent`() system call uses the kqueue [20] event notification mechanism. It operates similarly to `poll`() and `select`(), but is a stateful interface which does not require the full set of descriptors under surveillance to be communicated to the kernel in each call. However, changes can be made simultaneously to event query, and the event loop uses this to change the status of write polling when necessary.

4.6. Other Uses: The Portal File System

Distributed file systems are not the only application for the *puffs* buffering and event framework. Another example for the use of such a framework was found in the reimplementation of the portal file system [21] using *puffs*.

The portal file system is a 4.4BSD file system which provides some support for userspace file systems. It does not, strictly speaking, implement a file system, but relies on a provider to open a file descriptor, which is then passed to the calling process. What happens is that a process opening the file `/p/a/file` will receive a file descriptor as the result of the open operation and is in most cases not able to distinguish between an actual file system backing the file descriptor. A configuration file specifies which provider the portal daemon executes for which path.

Figure 5:
Event Loop Pseudocode

```
while (mounted) {
   fs->loopfunc();

   foreach (fd_set) {
      if (has_output)
         write();
   }

   foreach (fd_need_writechange) {
      if (needs_write && !in_set)
         add_writeset();
      if (!needs_write && in_set)
         rm_writeset();
   }

   kevent();

   foreach (kevent_result) {
      if (read)
         input();
      if (write)
         output();
   }
}
```

To facilitate this type of action, the original portal file system passes a pathname to the userspace portal daemon as part of the open method. Upon receiving a request, the daemon *fork*()s, lets the child take care of servicing the request, and listens to more input from the kernel. After the child has opened the file descriptor, it communicates the result back to the kernel. The kernel then transfers this descriptor to the calling process. The child process exits, but depending on the type of provider it might have spawned some handlers e.g. using *popen*(). Other types, such as TCP sockets, require no backing process.

The *puffs* portal file system driver behaves toward applications exactly like the old portal file system and even reuses most of the code of the original portald userspace implementation. However, *puffs* portalfs operates like a real file system in the sense that the file system driver interprets all the requests instead of a file descriptor being passed to a caller.

The problem in using original portald code is that the portal providers can execute arbitrary blocking sequences, and allowing one to execute in the context of the file server blocks the access to other files. This can be avoided by either multiple processes or multiple threads. The original portald code we use relies on processes for cleanup in some cases, such as cleaning up after *popen*(), so processes were chosen.

Operation of the new portal file system driver is as follows. When the file system *open* method is called, the file system driver opens a socketpair and forks off a child process. The driver then yields after enabling the child socket descriptor as a valid I/O descriptor. Meanwhile, the child proceeds to open a file descriptor and sends it to file server using descriptor passing[5]. After receiving the descriptor the file system driver returns success to the process calling *open*.

Read and write calls require asynchronous I/O for the file system driver to support concurrent access properly. As opposed to psshfs and 9puffs, the descriptors produced by the portal daemon are enabled for reading only when an incoming read request arrives from the kernel. This way data is consumed from the descriptor only when there is a read request active.

The read request uses the framework's *directreceive* routine for receiving data in which the file system driver supplies the buffer to receive data to without having to get it via `gotframe`(). By threading the number of bytes the kernel wishes to read from the file to the *readframe* routine using the buffer, the driver can also avoid reading too much data. Reading too much data would result in having to store it for the next read call.

The reimplementation performs better in some cases. Since *puffs* calls are interruptible, the calling processes can interrupt operations. The original portalfs implementation had a problem that if the open call on portalfs blocked, the calling process could not be interrupted. An example is an unreachable but not rejected network connection, which will stall until the *connect*() system call of the portald provider child times out. As a downside for this implementation, calls now need to traverse the user-kernel boundary three times instead of operating directly on the file descriptor in the calling process. However, as portalfs is rarely, if ever, used in speed-critical scenarios, this does not constitute a problem.

[5] Another option would be to issue a version of *fork*() which shares the descriptor table between the parent and the child, but some form of wakeup from the child to the parent is required in any case.

4.7. Other Uses: Kernel VFS Communication

If we return to Figures 1 and 2, we notice that the situation between the file server and and kernel virtual file system is symmetric: both are used by the file system driver through a communication protocol. After writing the framework, kernel communication could be adopted to use it instead of requiring special-purpose code. All incoming requests from the kernel are treated as `gotframe` and are dispatched by the library to the correct driver method. Using the framework for kernel communication also enables forwarding the *puffs* protocol to remote sites just by adding logic to route PDUs.

Not all communication in the file system is originated by the kernel. For example, the file system driver can request the kernel to flush or invalidate its caches. In this case a request is send to the kernel. As the response to the request is not immediate, it must be waited for. By treating the kernel virtual file system just as another file server, the framework readily handles yielding the caller, processing other file system I/O meanwhile, and rescheduling the caller back when the response arrives. This scheme also works independent of if the kernel virtual file system is on the local machine or a remote site.

5. Comparisons

This section presents three comparative studies for the framework. The first one compares implementation code size before and after the framework was introduced. The second measures performance between userspace file system drivers and the in-kernel NFS. Finally, a feature and usage comparison between psshfs and NFS is presented.

5.1. Code Size Comparison

Originally, both psshfs and 9puffs were implemented with their own specific buffering routines and event loop. These routines were first written when developing psshfs and were adapted to 9puffs with some changes.

Figure 6 (NFS included for comparison) shows that two thirds of the code used for networking and buffering could be removed with the introduction of the framework; what was left is the portion dealing with the file system protocol. As a purely non-measurable observation, the code abstracted into the framework is the most difficult and error-prone code in the file system driver.

Figure 6:
Code Size Comparison

total lines of code

	before	after	save (%)
psshfs	2885	2503	11%
9puffs	2601	2140	18%
NFS	24286	n/a	n/a

code involved in events / networking / buffering

	before	after	save (%)
psshfs	355	119	66%
9puffs	411	150	64%

Building packets is done by linear construction code while parsing is the reverse operation. On the other hand, the network scheduling code and event loop depends on timings and the order in which events happen. Moreover, the data structures required to hook this into the *puffs* multitasking framework are not obvious. With the networking framework the file system driver author does not need to worry about these details and can concentrate on the essential part: how to do protocol translation to make the kernel virtual file system protocol and the file server talk to each other.

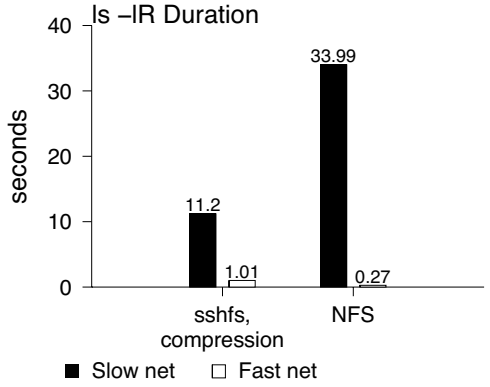
5.2. Directory Traversal

In this section we explore the performance of a file system driver and issues with the file system protocol when executing the commonplace Unix long listing: `ls -l`.

The command `ls -l` is characterized by three different VFS operations. First, the directory is read using *readdir*. Second, the node for each directory entry is located using the *lookup* operation. Finally, the node attributes for the `ls` long listing are fetched using the *getattr* operation. These operations map a bit differently depending on the file system protocol. For example, on NFSv3 the *readdir* operation causes an *NFS_READDIR* RPC to be issued. For each *lookup*, *NFS_LOOKUP* procedures are issued. Since the *NFS_LOOKUP* operation response also contains the node attributes, they are cached in the file system when *getattr* is called and no network I/O will be required for satisfying the request.

As discussed already in *Chapter 3.2*, the bottleneck in the above is the serial nature of the process: one operation must complete before the

Figure 7:



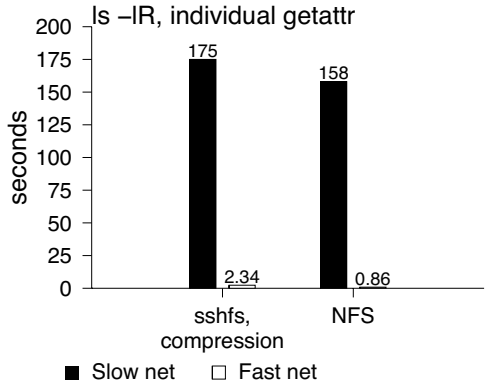
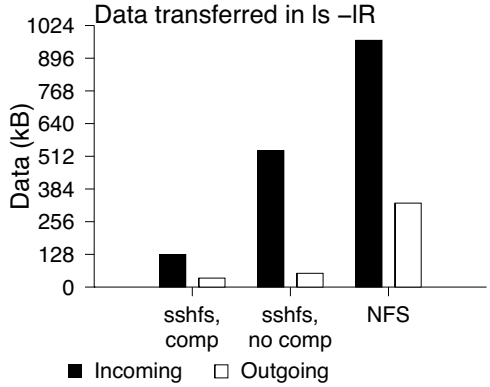
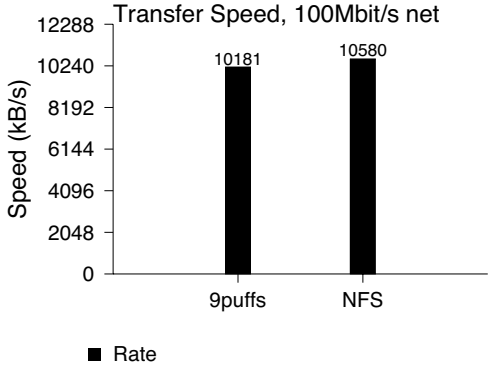


Figure 8:





next is issued. NFS solved this by introducing the *NFS_READDIRPLUS* procedure in protocol version 3. It returns the attributes for all the nodes in the *readdir* response. That way the file system driver will already have the information for *lookup*/*getattr* cached when it is requested. However, this information might well be wasted, since *readdir* is acting only opportunistically. Even further, as the BSD NFS implementation creates a new vnode for each previously non-existing one (to cache the attribute information in), listing a directory might prompt valid vnodes to be recycled. This is why the use of *NFS_READDIRPLUS* is disabled by default and recommended only for high latency mounts.

The sftp (and 9P) protocol always includes the attribute information in *readdir* responses. Our implementation differs from the kernel version in such a fashion that we cache the attributes and the directory read results in the directory structure; not new nodes. Therefore this solution does not force kernel vnodes to be recycled – although it could not do so even if it wanted to, as vnode life cycles are completely controlled by the kernel in *puffs*.

The `ls -lR` measurements are for the time it takes to traverse a directory hierarchy with around 4000 files. The initial measurements were done over a 11Mbps wireless link with a 3ms RTT. These results results showed the psshfs was faster than NFS – a result quite unexpected. The cause was discovered to be the bandwidth use, so another measurement was done with the ssh compression option being on (default compression level). This improved results even further. Technically it is possible to compress the NFS traffic also using the IP Payload Compression Protocol (IPComp), but doing so is multiple times more difficult than using the ssh compression option and the effects of doing so were not investigated.

The measurements presented in *Figure 7* contain both the duration for the operations on a high-latency, low bandwidth link and a low-latency high bandwidth local area network.

Performance is measured both for coalesced *getattr*s and individual ones. NFS is mounted using a TCP mount. Apart from compression, psshfs is used with OpenSSH default options.

For the preloaded attributes case, it is easy to see that psshfs wins on the slower network because it requires much less data to be transferred. However, on the high speed network the performance penalty inherent in multiple context switches per operation is evident. Even though latency is canceled for the link by using attribute preloading, the psshfs file system server must still *getattr* each file using individual system calls when the NFS server can simply perform these operations inside the kernel without context switch penalty. Additionally, psshfs must encrypt and decrypt the data. Finally, we do not control the sftp server and cannot optimize it.

Without preloading attributes ("individual getattr") NFS dominates because the operation becomes driven by latency, and NFS as a kernel file system has a smaller latency.

5.3. Data transfer

To measure raw data transfer speeds, large files were read sequentially over a local area network using both NFS and 9puffs[6]. The results are hardly surprising, as reading large files uses read-ahead heuristics. Data requested by the application has already been read into the page cache by the read-ahead code and can be delivered to the application instantly without consulting the file system server. It should be noted, though, that the userspace model uses more CPU and on fast networks such as 10GigE, the performance of the userspace model may be CPU-bound.

5.4. psshfs vs. NFS

As NFS and psshfs are roughly equivalent in performance, it is valid to question which one should be preferred in use. The following section lists reasons NOT to use the protocol in question:

psshfs:

- No support for hard links. Two hard-linked directory entries will be treated as two files.
- No support for devices, sockets or fifos.
- No support for user credentials in the protocol: one mount is always one set of credentials at the server end.
- No support for an async I/O model: there is no certainty if written data is committed to disk.

NFS:

- Setup is usually a heavyweight operation meaning the protocol cannot be used without considerable admin effort.
- It is difficult, although entirely possible, to make the protocol operate from a remote location through IPSec tunnels.
- There is no real security model in the currently dominant NFSv3 version.

[6] psshfs was attempted first, but the CPU requirements for the encryption capped out the CPU of the server machine.

6. Conclusions and Future Work

This paper explored implementing distributed file system drivers in userspace on top of the *puffs* Pass-to-Userspace Framework File System. It explained concepts relevant to implementing distributed file systems and pointed out unexpected pitfalls.

A framework for implementing distributed file systems was presented. The I/O file descriptors, callbacks, continuations and memory buffers were discussed and interfacing with them from the file system driver was explained.

The performance characteristics of userspace file system drivers and userspace file systems was lightly measured and analyzed. The conclusion was that even though in-kernel file systems usually perform better, userspace file systems can shrink the gap by the possibilities in a more flexible programming environment.

Future work includes implementing a peer-to-peer file system on top of the framework. Notably though, the portal file system implementation briefly mentioned in this paper already shares some similar characteristics to peer-to-peer file systems in that it communicates using multiple I/O descriptors concurrently.

As distributed file systems have a high price to pay for reloading information from the server, the information should be cached as much as possible. File data is cached effectively in the kernel page cache, although a file system driver wanting a persistent cache will have to implement it by itself. However, metadata caching is currently completely up to the file system driver. This could be improved in the future by providing a method for caching metadata.

Related to metadata caching is the observation that the optimal way to perform directory reading and *lookup* is a similar procedure in both

of the distributed file systems we went over in this paper. The procedure should be generalized for any file system driver. This includes attribute caching for directory entries along with optionally preloading the attributes even though the file system protocol does not directly support it.

Availability

All of the code discussed in this paper is available for download and use in the development branch of the NetBSD [7] operating system. This development branch will eventually become the NetBSD 5.0 release.

For information on how to download the code in source form or as a binary release, please see *http://www.NetBSD.org/*. Documentation for enabling and using the code is available at *http://www.NetBSD.org/docs/puffs/*

Acknowledgments

This work was funded by the Finnish Cultural Foundation and the Research Foundation of Helsinki University of Technology. Karl Jenkinson provided helpful comments.

References

1. Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, *Design and Implementation of the Sun Network Filesystem,* pp. 119-130, Summer 1985 USENIX Conference (1985).
2. Christopher Hertel, *Implementing CIFS: The Common Internet File System,* Prentice Hall (2003). ISBN: 978-0-13-047116-1.
3. Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen, *Ivy: A Read/Write Peer-to-peer File System,* Fifth Symposium on Operating Systems Design and Implementation (December 2002).
4. J-M. Busca, F. Picconi, and P. Sens, *Pastis: A Highly-Scalable Multi-User Peer-to-Peer File System,* Euro-Par 2005 (2005).
5. Michael Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, *Mach: A New Kernel Foundation for UNIX Development,* pp. 93-113, Summer USENIX Conference (1986).
6. Miklos Szeredi, *Filesystem in USErspace, http://fuse.sourceforge.net/* (referenced January 2008).
7. The NetBSD Project, *The NetBSD Operating System*. http://www.NetBSD.org/.
8. Antti Kantee, *puffs - Pass-to-Userspace Framework File System,* pp. 29-42, AsiaBSDCon 2007 (March 2007).
9. Yousef A. Khalidi, Vlada Matena, and Ken Shirriff, "Solaris MC File System Framework," TR-96-57, Sun Microsystems Laboratories (1996).
10. T. Ylönen and S. Lehtinen, *SSH File Transfer Protocol draft 02,* Internet-Draft (October 2001).
11. Bell Labs, "Plan 9 File Protocol, 9P," *Plan 9 Manual*.
12. *puffs -- Pass-to-Userspace Framework File System development interface* (January 2008). NetBSD Library Functions Manual.
13. S. R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX,* pp. 238-247, Summer Usenix Conference, Atlanta, GA (1986).
14. Sun Microsystems, Inc., *RPC: Remote Procedure Call Protocol Specification Version 2,* RFC 1057 (June 1988).
15. Rick Macklem, *The 4.4BSD NFS Implementation,* The 4.4BSD System Manager's Manual (1993).
16. Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *Design and Implementation of the 4.4BSD Operating System,* Addison-Wesley (1996).
17. Randall Stewart and Chris Metz, "SCTP: New Transport Protocol for TCP/IP," *IEEE Internet Computing,* Volume 5, Issue 6, pp. 64-69 (2001).
18. *puffs_framebuf -- buffering and event handling for networked file systems* (January 2008). NetBSD Library Functions Manual.
19. W. Richard Stevens, *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI,* Prentice Hall (1998). ISBN 0-13-490012-X.
20. Jonathan Lemon, *Kqueue: A Generic and Scalable Event Notification Facility,* pp. 141-154, USENIX 2001 Annual Technical Conference, FREENIX Track (June 2001).
21. W. Richard Stevens and Jan-Simon Pendry, *Portals in 4.4BSD,* USENIX Technical Conference (1995).