# Grave Robbers from Outer Space Using 9P2000 Under Linux

Eric Van Hensbergen
*IBM Austin Research Lab*
Ron Minnich
*Los Alamos National Labs*

## Abstract

This paper describes the implementation and use of the Plan 9 distributed resource protocol 9P under the Linux 2.6 operating system. The use of the 9P protocol along with the recent addition of private name spaces to the 2.6 kernel creates a foundation for seamless distributed computing using Linux. We review the design principles and benefits of Plan 9 distributed systems, go over the basics of the 9P protocol, describe 9P extensions to better support UNIX® file systems, and show some example Linux distributed applications using 9P to provide system and application services. We conclude by providing a performance analysis of the protocol versus NFS for sharing a static file system.

## Motivation

This paper describes the implementation and use of the Plan 9 design principles and infrastructure under the Linux operating system with the intent of providing a unified ubiquitous distributed environment for the system and applications. Plan 9 [Pike90] was a new research operating system and associated applications suite developed by the Computing Science Research Center of AT&T Bell Laboratories (now a part of Lucent Technologies), the same group that developed UNIX , C, and C++. Plan 9 was initially released in 1993 to universities, and then made generally available in 1995 [9FAQ]. Its core operating systems code laid the foundation for the Inferno Operating System released as a product by Lucent Bell-Labs in 1997 [INF1]. The Inferno venture was the only commercial embodiment of Plan 9 and is currently maintained as a product by Vita Nuova [VITA]. After updated releases in 2000 and 2002, Plan 9 was open-sourced under the OSI [OSI] approved Lucent Public License [LPL] in 2003.

The Plan 9 project was started by Ken Thompson and Rob Pike in 1985. Their intent was to explore potential solutions to some of the shortcomings of UNIX in the face of the widespread use of high-speed networks to connect machines [Pike04]. In UNIX , networking was an afterthought and UNIX clusters became little more than a network of stand-alone systems. Plan 9 was designed from first principles as a seamless distributed system with integrated secure network resource sharing. Applications and services were architected in such a way as to allow for implicit distribution across a cluster of systems. Configuring an environment to use remote application components or services in place of their local equivalent could be achieved with a few simple command line instructions. For the most part, application implementations operated independent of the location of their actual resources.

Commercial operating systems haven't changed much in the 20 years since Plan 9 was conceived. Network and distributed systems support is provided by a patchwork of middle-ware, with an endless number of packages supplying pieces of the puzzle. Matters are complicated by the use of different complicated protocols for individual services, and separate implementations for kernel and application resources. We are proposing leveraging Plan 9's principles in modern commercial operating systems in an attempt to provide a more coherent, unified approach to distributed systems. The rest of this paper reviews Plan 9's principles, discusses the details of the 9P protocol, describes our Linux implementation, and evaluates its performance compared to NFSv3.

## Plan 9 Design Principles and Practice

Plan 9's transparent distributed computing environment was the result of three core design principles which permeated all levels of the operating system and application infrastructure:

1) develop a single set of simple, well-defined interfaces to services
2) use a simple protocol to securely distribute the interfaces across any network
3) provide a dynamic hierarchical structure to organize these interfaces

In Plan 9, all system resources and interfaces are represented as files. UNIX pioneered the concept of

treating devices as files, providing a simple, clear interface to system hardware. In the 8th edition, this methodology was taken further through the introduction of the /proc synthetic file system to manage user processes [Kill84]. Synthetic file systems are comprised of elements with no physical storage, that is to say the files represented are not present as files on any disk. Instead, operations on the file communicate directly with the sub-system or application providing the service. Linux now contains multiple synthetic file systems representing devices (devfs), process control (procfs), and interfaces to system services and data structures (sysfs).

Plan 9 took the file system metaphor further, using file operations as the simple, well-defined interface to all system and application services. The intuition behind the design was based on the assumption that any programmer knows how to interact with files. As such, interfaces to all kernel subsystems from the networking stack to the graphics frame buffer are represented within synthetic file systems. This unified approach dramatically simplifies access to all system services, evident in Plan 9's use of only 40 system calls compared to Linux's 300.

User-space applications and services export their own synthetic file systems in much the same way as the kernel interfaces. Common services such as domain name service (DNS), authentication databases, and window management are all provided as file systems. End-user applications such as editors and e-mail systems export file system interfaces as a means for data exchange and control. The benefits and details of this approach are covered in great detail in the existing Plan 9 papers [PPTTW93].

9P [9man] represents the abstract interface used to access resources under Plan 9. It is somewhat analogous to the VFS layer in Linux [Love03]. In Plan 9, the same protocol operations are used to access both local and remote resources, making the transition from local resources to cluster resources to grid resources completely transparent from an implementation standpoint. Authentication is built into the protocol, and was extended in its Inferno derivative Styx [STYX] to include various forms of encryption and digesting.

It is important to understand that all 9P operations can be associated with different active semantics in synthetic file systems. Traversal of a directory hierarchy may allocate resources, or set locks. Reading or writing data to a file interface may initiate actions on the server, such as when a file acts as a control interface. The dynamic nature of these semantics makes caching dangerous and in-order synchronous execution of file system operations a must.

The 9P protocol itself requires only a reliable, in-order transport mechanism to function. It is commonly used on top of TCP/IP [RFC793], but has also been used over RUDP [RFC1151], PPP [RFC1331], and over raw reliable mechanisms such as the PCI bus, serial port connections, and shared memory. The IL [PrWi95] protocol was designed specifically to provide 9P with a reliable, in order transport on top of an IP stack without the overhead of TCP.

In the fourth edition of Plan 9 released in 2002, 9P was redesigned to address a number of shortcomings and retitled 9P2000 [P903]. 9P2000 removed the file name length limitation of 28 bytes, and sought to optimize several operations involved in traversing file hierarchies. It also introduced a negotiation phase accommodating different versions of the protocol as well as protocol parameter negotiation.

The final key design point is the organization of all local and remote resources into a dynamic private name space. A name space is a mapping of system and application resources to names within a file system hierarchy. Manipulation of the location of the elements within a name space can be used to configure which services to use, to interpose stackable layers onto service interfaces, and to create restricted "sandbox" environments.

Under Plan 9 and Inferno, the name space of each process is unique and dynamic. A name space can be manipulated through mount(1) and bind(1) commands. Mount operations allow a client to add new interfaces and resources to their name space. These resources can be provided by the operating system, by a synthetic file server, or from a remote server. Bind commands allow reorganization of the existing name space, allowing certain services to be "bound" to well-known locations. Bind operations can also be used to substitute one resource for another, for example by binding a remote device over a local one. Binding can also be used to create stackable layers, by interposing one interface over another. Such interposable interfaces are particularly useful for debugging and statistics gathering as in the Plan 9 application iostat(4).

Processes inherit an initial name space from their parent, but changes made to the client's name space are not typically reflected in the parent's. This allows each process to have a context-specific name space. The

system makes extensive use of this facility to provide context-sensitive file interfaces. For example, in Plan 9, the file /dev/cons refers to the current process' standard input and output file descriptors. In a similar fashion /dev/user reports the user name, and /dev/pid reports the current process id. This same facility is used by the windowing system to provide information and control over the process' window.

Handcrafted name spaces can be used to create secure sandboxes which give users access to very specific system resources. This can be used in much the same way as the UNIX chroot facility - except that the chroot name spaces under Plan 9 can be completely synthetic - with specific executables and interfaces "bound" into place instead of copied to a sub-hierarchy.

The recent open-sourcing of Plan 9 and many of its applications has created the opportunity to leverage some of its unique concepts in other open source operating systems. Alex Viro incorporated the private name space concept into the 2.5 Linux kernel [Love03] and Russ Cox has ported a number of Plan 9 applications and libraries (including libraries allowing the creation of synthetic file server applications) to Linux, BSD, and OSX [plan9ports]. This paper describes the final missing piece, a native 9P protocol for Linux.

## The 9P2000 Protocol

As mentioned earlier, 9P2000 is the most recent version of 9P, the Plan 9 distributed resource protocol. It is a typical client/server protocol with request/response semantics for each operation (or transaction). 9P can be used over any reliable, in-order transport. While the most common usage is over pipes on the same machine or over TCP/IP to remote machines, it has been used on a variety of different mediums and encapsulated in several different protocols. The Internet Link (IL) protocol [PrWi95] was a lightweight encapsulation designed specifically for 9P.

9P has 12 basic operations, all of which are initiated by the clients. Each request (or T-message) is satisfied by a single associated response (or R-message). In the case of an error, a special response (R-error) is returned to the client containing a variable length string error message. It is important to note that there are no special operations for directories or links as in VFS [Love03] because these elements are just treated as ordinary files. The operations summarized in the following table fall into three categories: session management, file operations, and meta-data operations.

| ***class*** | ***op-code*** | ***description*** |
|---|---|---|
| session | version | parameter negotiation |
| | auth | security authentication |
| | attach | establish a connection |
| | flush | abort a request |
| | error | return an error |
| file | walk | lookup pathname |
| | open | access a file |
| | create | create & access a file |
| | read | transfer data from a file |
| | write | transfer data to a file |
| | clunk | release a file |
| metadata | stat | read file attributes |
| | wstat | modify file attributes |

Table 1: 9P2000 Operations

9P is best understood by seeing what messages are transmitted for some standard file system operations. One can do this by using the UNIX 9P server, u9fs, and turning debug mode on. The debug output (which goes to /tmp/u9fs.log by default) displays a human-readable transaction log. What follows in italics is an example session of a Plan 9 client contacting a Unix 9P2000 server and writing to a new file (/tmp/usr/testfile). Explanations of the various operations follow each request/response pair. Messages marked with (→) are from the client to the server, and (←) represent the responses. Note that you will see slightly different transaction sequences from a UNIX client due to the nature of the mapping of VFS operations.

→ *Tversion tag -1 msize 8216 version '9P2000'*

← *Rversion tag -1 msize 8216 version '9P2000'*

The *version* operation initiates the protocol session. The *tag* accompanies all protocol messages and is used to multiplex operations on a single connection. The client selects a unique *tag* for each outbound operation. The *tag* for *version* operations, however, is always set to -1. The next field, *msize* negotiates the maximum packet size with the server including any headers - the server may respond with any number less than or equal to the requested size. The *version* field is a variable length string representing the requested version of the protocol to use. The server may respond with an earlier version, or with an error if there is no earlier version that it can support.

→ *Tauth tag 5 afid 291 uname 'bootes' aname ''*

← *Rerror tag 5 ename 'u9fs authnone: no authentication required'*

The *auth* operation is used to negotiate authentication information. The *afid* represents a special authentication handle, the *uname* (bootes) is the user name attempting the connection and the *aname*, (which in this case is blank), is the mount point the user is trying to authenticate against. A blank *aname* specifies that the root of the file server's hierarchy is to be mounted. In this case, the Plan 9 client is attempting to connect to a Unix server which does not require authentication, so instead of returning an *Rauth* operation validating the authentication, the server returns *Rerror*, and in a variable length strength in the field *ename*, the server returns the reason for the error.

→ *Tattach tag 5 fid 291 afid -1 uname 'bootes' aname ''*

← *Rattach tag 5 qid (0902 1097266316 d)*

The *attach* operation is used to establish a connection with the file server. A *fid* unique identifier is selected by the client to be used as a file handle. A *Fid* is used as the point of reference for almost all 9P operations. They operate much like a UNIX file descriptor, except that they can reference a position in a file hierarchy as well as referencing open files. In this case, the *fid* returned references the root of the server's hierarchy. The *afid* is an authentication handle; in this case it is set to -1 because no authentication was used. *Uname* and *aname* serve the same purposes as described before in the *auth* operation.

The response to the *attach* includes a *qid*, which is a tuple representing the server's unique identifier for the file. The first number in the tuple represents the *qid.path*, which can be thought of as an inode number representing the file. Each file or directory in a file server's hierarchy has exactly one *qid.path*. The second number represents the *qid.version*, which is used to provide a revision for the file in question. Synthetic files by convention have a *qid.version* of 0. *Qid.version* numbers from UNIX file servers are typically a hash of the file's modification time. The final field, *qid.type*, encodes the type of the file. Valid types include directories, append only files (logs), exclusive files (only one client can open at a time), mount points (pipes), authentication files, and normal files.

→ *Twalk tag 5 fid 291 newfid 308 nwname 0*

→ *Rwalk tag 5 nwqid 0*

*Walk* operations serve two purposes: directory traversal and *fid* cloning. This *walk* demonstrates the latter. Before any operation can proceed, the root file handle (or *fid*) must be cloned. A clone operation can be thought of as a dup, in that it makes a copy of an existing file handle - both of which initially point to the same location in the file hierarchy. The cloned file handle can then be used to traverse the file tree or to perform various operations. In this case the root *fid* (291) is cloned to a new *fid* (308). Note that the client always selects the *fid* numbers. The last field in the request transaction, *nwname*, is used for traversal operations. In this case, no traversal was requested, so it is set to 0. The *nwqid* field in the response is for traversals and is discussed in the next segment.

→ *Twalk tag 5 fid 308 newfid 296 nwname 2 0:tmp 1:usr*

← *Rwalk tag 5 nwqid 2 0:(0034901 1093689656 d) 1: (0074cdd0 1096825323 d)*

Here we see a traversal request walk operation. All traversals also contain a clone operation. The *fid* and *newfid* fields serve the same purpose as described previously. *Nwname* specifies the number of path segments which are attempting to be traversed (in this case 2). The rest of the operands are numbered variable length strings representing the path segments - in this case, traversing to /tmp/usr. The *nwqid* in the response returns the qids for each segment traversed, and should have a qid for each requested path segment in the request. Note that in this case there are two pathname components: the path name is walked at the server, not the client, which is a real performance improvement over systems such as NFS which walk pathnames one component at a time.

→ *Tcreate tag 5 fid 296 perm --rw-rw-rw- mode 1 name 'testfile'*

← *Rcreate tag 5 qid (074cdd4 1097874034 ) iounit 0*

The *create* operation both creates a new file and opens it. The *open* operation has similar arguments, but doesn't include the *name* or *perm* fields. The *name* field is a variable length string representing the file name to be created. The *perm* field specifies the user, group, and other permissions on the file (read, write, and execute). These are similar to the core permissions on a unix system. The *mode* bit represents the mode with which you want to open the resulting file (read, write, and/or execute). The response contains the *qid* of the newly created (or opened) file and the *iounit*, which specifies the maximum number of bytes which may be read or written before the transaction is split into multiple 9P messages. In this case, a response of 0 indicates that the file's maximum message size

matches the session's maximum message size (as specified in the *version* operation).

→ *Tclunk tag 5 fid 308*

← *Rclunk tag 5*

The *clunk* operation is sent to release a file handle. In this case it is releasing the cloned handle to the root of the tree. You'll often see transient *fid*s used for traversals and then discarded. This is even more extreme in the UNIX clients as they only traverse a single path segment at a time, generating a new *fid* for each path segment. These transient *fid*s are a likely candidate for optimization, and may be vestigial from the older 9P specification which had a separate clone operation and didn't allow multiple segment walks.

→ *Twrite tag 5 fid 296 offset 0 count 8 'test'*

← *Rwrite tag 5 count 8*

We finally come to an actual I/O operation, a write operation that writes the string 'test' into the new file. *Write* and *read* operands are very similar and straightforward. The *offset* field specifies the offset into the file to perform the operation. There is no separate seek operation in 9P. The *count* represents the number of bytes to read or write, and the variable length string ('test') is the value to be written. The response *count* reports the number of bytes successfully written. In a *read* operation the response would also contain a variable length string of *count* size with the data read.

→ *Tclunk tag 5 fid 296*

← *Rclunk tag 5*

This final clunk releases the *fid* handle to the file -- approximating a close operation. You'll note that the only *fid* remaining open is the root *fid* which remains until the file system is unmounted. Several operations were not covered in this transaction summary. *Flush* is almost never used by clients in normal operation, and is typically used to recover from error cases. The *stat* operation, similar to its UNIX counterpart, is used to retrieve file metadata. *Twstat* is used to set file metadata, and is also used to rename files (file names are considered part of the metadata).

## 9P2000 Unix Extensions

Many modern UNIX systems, including Linux use a virtual file system (VFS) layer as a basic level of abstraction for accessing underlying implementations. Implementing 9P2000 under Linux is a matter of mapping VFS operations to their associated 9P operations. The problem, however, is that 9P2000 was designed for a non-UNIX system so there are several fundamental differences in the functional semantics provided by 9P.

Under Plan 9, user names as well as groups are represented by strings, while on Unix they are represented by unique numbers. This is complicated by Linux making it exceedingly difficult to map these numeric identifiers to their string values in the kernel. Many of the available UNIX network file systems avoid this issue and simply use numeric identifiers over the wire, hoping they map to the remote system. NFSv4 [NFS4] has provisions for sending string group and user info over the wire and then contacting a user-space daemon which attempts to provide a valid mapping.

We use two different name mapping approaches based on server system type. When contacting a 9P server on another UNIX system we use numeric identifiers. When contacting a Plan 9 file server we map all numeric ids to the numeric id of the mounting user. While this yields less than accurate uid/gid information in directory listings, permissions checking (which is done on the remote server) is still valid. A potential future piece of work would be to provide a user-space daemon similar to NFSv4 to provide uid/gid mapping services or perhaps somehow leverage the existing NFSv4 service.

One of the unique aspects of the Plan 9 name space is that it is dynamic. Users are able to bind files and directories over each other in stackable layers similar to union file systems. This aspect of Plan 9 name spaces has obviated the need for symbolic or hard links. Symlinks on a remote UNIX file server will be traversed transparently as part of a walk - there is no native ability within Plan 9 to create symlinks. This breaks many assumptions for Linux file-systems and many existing applications (for example the kernel build creates a symlink in the include directory as part of the make process).

To preserve compatibility with these existing applications we implemented a transparent extension to the file system semantics which doesn't effect the protocol syntax. Files requiring this extension have their filenames prefixed with a marker character, '/', which is normally illegal in file system operations. A string following the '/' provides details of the type of extension and is followed by a numeric mode extension. For symlinks and hard links the file's content contains the link information. These extensions are essentially ignored by the Plan 9 file server, but interpreted correctly by UNIX clients and servers.

The same extension can be used to provide support for additional mode bits and other file types not provided under Plan 9. The extension codes are documented in the table below:

| ***extension*** | ***code*** |
|---|---|
| symbolic link | /syml*x* |
| hard link | /link*x* |
| character device | /char*x* |
| block device | /blck*x* |
| pipe | /pipe*x* |
| extra mode | /mode*x* |
| ***mode*** | ***code(x)*** |
| none | 0 |
| set-user-id | 1 |
| sticky bit | 2 |
| directory exec (X) | 4 |

Table 2: 9P2000.u Extension Codes

Since these are all extensions to the 9P2000 protocol, they must be negotiated during the version stage of establishing the connection. The clients register interest in the extension by appending a *.u* to the version string (e.g. 9P2000.u). If the server is capable of providing the UNIX extensions, it will respond with a 9P2000.u in the response message. If the server does not wish (or cannot) provide the UNIX extensions, it will respond without the .u (e.g. 9P2000).

An operation which does not currently have any support in the 9P2000 protocol or UNIX extensions is ioctl. Fortunately, in the Linux world, ioctl is being deprecated in favor of sysfs-based mechanisms. For interfaces still requiring ioctl operations (such as sockets), gateway synthetic file server applications can map ioctl functionality to synthetic control files.

## Details of the Linux Implementation

For the most part, 9P2000 can be directly mapped to the Linux VFS interface. There are several semantic differences beyond the syntactic differences mentioned in the above section which are worth noting.

One difference is the model by which a file system is mounted. On UNIX systems this is done either at boot time, by the super user, or through an automounter (which must be configured for particular resources by the super user). Additionally, once a file system is mounted it is visible (and accessible) to all users. By contrast, under Plan 9 file systems are mounted by individual users into their private name space environment. The connection to the file server is authenticated by the credentials of the individual user at mount time, and the file system is mounted in the user's private name space - so it is not directly visible to any other user. Even diskless terminals work this way under Plan 9 - a user must first log in before the terminal mounts its root file system from the file server.

The Linux 9P2000 driver can accommodate both systems. It can mount the file server as root (or some other special user) into the global Linux name space or it can operate with the user initiating the mount and it being authenticated against his or her user ID. The two complications are that, by default, Linux has a global name space for all users, and ordinary users don't typically have permission to mount arbitrary systems on arbitrary mount points.

This can be solved by having a special set-uid version of mount which allows certain users (or users in a particular group) to mount from certain systems matching a list of regular expressions to particular mount points matching a regular expressions. A special version of mount is required in order to do DNS resolution of the server names as NFS support for DNS resolution is a special case implementation in the standard mount application. Users can spawn off a private name space by using a simple wrapper utility (as described in the *Application* section).

A third alternative, which is currently the most secure model, is for the user to ssh to the remote file server and start his own private u9fs instance which uses the ssh connection as a transport. On the client system a set-uid mount application can attach to the i/o stream and a wrapper can be used to establish a private name space if desired. This still requires a properly configured set-uid mount application on the client and is only useful after the client system is fully booted. In fact, this is the way that Plan 9 users commonly mount UNIX file systems using u9fs when they don't have root permissions on the file server.

File creation under Plan 9 is atomic. That is to say the file is created and opened in a single 9P operation. Under VFS, creation semantics are entirely separate from open, and therefore not atomic. The Linux 9P2000 implementation seeks to preserve some of the atomicity of the original Plan 9 semantics by caching the fid in the inode structure - however, this does not guarantee the same atomic semantic.

Cache policy was a major concern when implementing the 9P2000 driver. Linux file systems routinely use two caches: the dcache provides caching of directory lookup information effectively caching inodes and metadata so they don't have to be re-read from the

disk, while the page cache provides a data cache. Both create problems as 9P2000 is intended to be a non-cached synchronous protocol since it does more than just provide a transport to share files.

The dcache creates problems because it caches inodes, effectively caching both metadata and file/directory lookup. The problem is that synthetic file system semantics may be attached to metadata operations (stat/twstat) and file system hierarchy movement (walk). This case is particularly important to consider due to the dynamic nature of synthetic file systems. For example, since a file server assumes it will see walks every time a directory is traversed, it could use such traversal as a locking mechanism. If the results of the walk is cached and reused, it would violate the lock semantics. The other problem with the dcache is that *fids* stay open (attached to the inode in the dcache) which would otherwise be clunked. Clunks are another operation which commonly have associated semantics within synthetic file systems (such as unlocking).

One solution is to always report dcache entries as invalid, forcing the clunk of their associated *fids* and requiring a rewalk from the mount-point to their location. This is unfortunate as the directory cache provides a nice performance improvement, particularly in cases where a deep directory structure is being repeatedly traversed. An alternative solution lies in the version field of Plan 9 *qid*s. By convention, static file versions start at 1 and synthetic files always have version 0. So the dcache solution is to automatically invalidate any dcache entry with a *qid.version* equal to 0, otherwise validate the dentry in a normal fashion via a stat transaction and comparing the version numbers.

The same basic policy can be applied to caching file system data in the page cache. Since synthetic file systems typically do not have version or even modification date information, there is no way to validate the cached inode, so you must assume it is invalid. Typically it is not desirable to cache synthetic file systems anyways, it doesn't make sense to think about caching the dynamic data in the /proc file system.

Loose consistency models present another particularly difficult problem. For example, by default NFS (v2 & v3) employs a timeout based invalidation strategy for its page cache and also implements a time-delayed write back in an attempt to coalesce operations. This results in admirable write performance optimization but undermines any synchronous operation of the protocol. This could be particularly damaging when you are using writes to the file system to control a remote service or manage locks.

Because of the drastically different semantics, dcache, page cache, and delayed write-back caching have been removed from the Linux 9P2000 driver. Plan 9 itself uses a stackable caching file system to provide a similar level of cache optimization. Unlike the standard Linux page cache, the Plan 9 cache file system can even use a disk as a backing store. This is particularly effective for workstations that have a disk but get their root file systems from a server. David Howell's CacheFS [CFS] shows promise as providing a similar stackable service for Linux. Ultimately, it is our intent to provide some level of cache for 9P via integration with a stackable cache system while preserving correct synthetic file system semantics.

## Applications

9P2000 support provides a nice alternative to NFS for distributing static files. It can be configured in such a way that users can export and mount their own file system resources. Its synchronous mode of operation and optional caching make it ideal for situations where the cache models and delayed writes of other file systems cause problems. Many utilities, such as CVS and various e-mail servers, discourage the use of NFS repositories due to concerns with data corruption resulting from bad cache behavior and lack of transaction semantics.

However, as mentioned several times before - distributed file service is not the only benefit of 9P2000. It can be used to share networking stacks and block and character devices between members of a 9P2000 cluster. It can be used to manipulate Linux synthetic file systems, such as /proc and /sys providing distributed control and management. It also can be used as an infrastructure to implement distributed applications.

A key to effective use of 9P2000 is the recently added private name space extensions in the 2.4.19 and later kernels. In order to use these, a special flag (*CLONE_NEWNS*) needs to set when the *clone* system call is used to create a child process. When the flag is used, Linux will create a copy of the parents name space instead of sharing the same copy. Modifications to the child's name space will not be visible to the parent and vice-versa. This can be used by normal users by writing a simple wrapper application for the standard shell which creates a new name space each time a new shell is invoked. An example wrapper application is included in the v9fs distribution.

Another possible use of the 9P2000 Linux support is with Russ Cox's UNIX ports of the Plan 9 applications. Several of the applications, including ACME [Pike94] and plumbing [Pike00], export synthetic file system interfaces to application structures. The plan9ports package includes wrapper applications which can be used with synthetic file systems without mounting them, but, having a native 9P file system, make access and manipulation much more easy and intuitive.

Similar types of synthetic file systems can now be written for Linux applications. An example would be a port of the task bag file system. The task bag file system is a synthetic file system that presents users with three directories: questions, working, and answers. Programs needing work done create exclusive-open files in the questions directory and write the questions to them, one question to each file. Once the file is closed, workers can pick up work from the questions directory (by opening a file in that directory). Files that are opened a second time for writing (i.e. by a worker) will disappear from the questions directory and appear in the working directory. If the file is closed without a write, it means the worker died or gave up; it reappears in the questions directory. When the file is closed, if there was data written to it, the file name will now appear in the answers directory.

This file system models an earlier system built using SunRPC. In that system, programs that wished to use the taskbag need to be built with SunRPC. In the taskbag file system, it is possible to use shell scripts to create, acquire, and get the answers for tasks. Status can be determined with 'ls' and 'cat'. The system works transparently over a Grid [9grid]. All in all, the task bag file system is far more capable than the SunRPC taskbag system that it replaces.

Private name spaces have also been integrated into the Clustermatic software suite [Clus]. Users can specify a set of mount points that are to be activated when their process or its children are run on a cluster node. When the process is migrated to the cluster node, the mount points are set up by the kernel before the process starts and are removed by the kernel after the process and any children exit. Note that this is not really an automounter, though it behaves in a similar way. The mount points are private and they are part of the context of the process, not defined a file in /etc. We have tested this system on the 1024-node Pink cluster at LANL. 1024 mounts take 20 seconds to set up, but once the system is running it is faster than NFS and far more stable.

## Performance

As mentioned earlier, 9P is a unifying protocol - it combines methods for name space organization, resource sharing, distributed applications, and file service. We focus our performance characterization on the distributed file service since it has easily identified comparison points and a host of generally available benchmarks. We chose to compare our implementation to NFS, because it seems to be the most widely used and understood distributed file system protocol.

Our test environment is a cluster of identically configured dual-processor 866 MHz Pentium III servers, each configured with 256 MB of memory, a 36GB IBM DDYS-T36950N SCSI drive formated with an ext2 file system, an Alteon 1 GB Ethernet card, and running Linux 2.6.8. We will show evaluations with caches disabled within 9P2000 and will test against NFSv3 (both tcp and udp). We compared the protocols with 8k and 32k buffer sizes.

We used two evaluation benchmarks to characterize the performance of our 9P2000 driver and compare it to NFS. The first is the Bonnie [Berry90] suite of file system benchmarks which test overall throughput through operations on a single large file. The second is the PostMark [Katch97] benchmark from Network Appliance; it performs a large number of transactions with smaller block operations and many files.

The Bonnie benchmark performs a series of tests on a single file of a specified size. The default is a 100 MB file, but we followed the instruction's suggested guideline of twice the size of available DRAM (512MB). This limits the effectiveness of cache operations and exposes the performance of the underlying file operations. It performs sequential writes and reads of the file, both a character at a time and in blocks. It also performs a read, modify, write sequence (rewrite), and random seeks followed by reads and writes of small chunks of data.

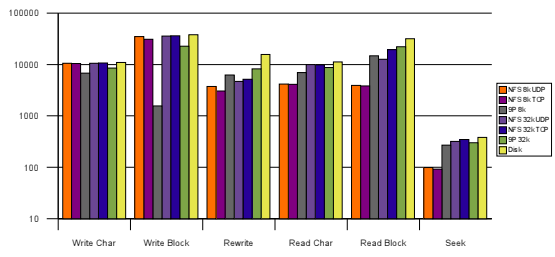


*Illustration 1: Bonnie Results (KB/sec)*

The results show comparable or better performance for all operations except write operations with small block sizes. On write benchmarks, 9P2000 does poorly due to the fact that async NFS delays write-back to coalesce transactions and does a better job of pipelining write operations. The effect of the operation coalescing is somewhat mitigated with larger block sizes, but the pipelining of write requests still gives NFS a distinct advantage.

For other operations 9P2000 shows slight advantages over NFS due to the lower complexity of the protocol and the fact this it lets the server handle metadata update instead of issuing separate transactions. 9P's lower complexity can be seen in its implementation with only 12 operations and just under 3500 lines of code, while NFS has 17 operations and is implemented in 9357 lines of code (source code counts based on David Wheeler's SLOCCount).

With the larger working set there is significantly less memory for caches, reducing the effectiveness of NFS's loose consistency model. Runs which had smaller footprints showed dramatic advantages for NFS, particularly for read block operations. Caches clearly have advantages and need to be considered for distributing static file systems.

Analysis of protocol traffic during the benchmark (using NFSDump [LISA03] and 9P server logs) shows approximately 50% fewer operations for NFS with 32k buffer sizes. This dramatic drop in the number of operations (primarily read and write operations) shows the effectiveness of caches for operation coalescing (in the write-char and read-char portions of the benchmark). With 8k buffer sizes, NFS retains an advantage in fewer write operations, but has roughly the same number of read operations as 9P. The fact that this huge reduction in the number of operations over the wire doesn't have a more dramatic effect on the performance seems to indicate the overhead added by NFS coherence and the page cache are detrimental to performance of workloads with poor locality.

PostMark is a benchmark developed under contract to Network Appliance. Its intent is to measure the performance of a system used to support a busy email service or the execution of a large number of CGI scripts processing forms. It creates the specified number of files randomly distributing the range of file sizes: it is typically used with very large numbers of relatively small files. It then runs transactions on these files, randomly creating and deleting them or reading and appending to them, and then it deletes the files. We ran our PostMark configuration with 500 files and 50000 transactions. This seemed sufficient to differentiate between the file systems and had a tractable run-time.

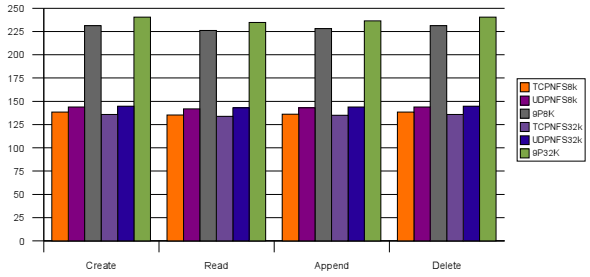


*Illustration 2: PostMark Operations (op/sec)*

PostMark paints a very different picture than the Bonnie benchmark, since the files and operations are much smaller - low latency handling of requests is much more important than overall throughput. 9P2000 demonstrates approximately double the performance of TCP and UDP NFSv3 (which effectively scored the same for the PostMark benchmark). Analysis of the protocol traffic shows 9P2000 with half the operations of NFS for 8k and 32k buffer sizes. NFS still shows an advantage on the number of read and write operations (in fact never sending a read operation over the wire due to the effectiveness of the page cache). However, NFS sends more than twice as many lookup operations and sends access messages almost every time it touches the files. The fact that so many synchronous operations are required to access a file really inhibits performance for NFS. By contrast, 9P does all permissions checking on the server and requires only a single operation to create and open a file.

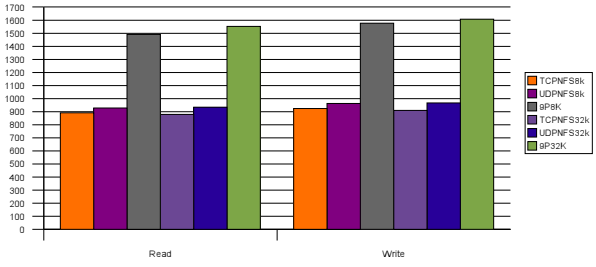


*Illustration 3: PostMark Bandwidth (KB/sec)*

## Related Work

9P is nominally just a remote file system. However, due to the unique way in which it is used to distribute not just files but other system and application

resources, it has advantages over other file system protocols for certain applications.

The Linux kernel supports several different network file systems. There is experimental support for the Andrew File System [HOW88], Coda [SATY90], Microsoft SMB and Common Internet File System [CIFS], and Netware's NCP [Majo94]. The most commonly used is the NFS protocol [Sun89]. Linux supports NFSv3, experimental support for NFSv4 [NFS4] and NFS over TCP. All of these are targeted purely for distributing static files and don't implement the correct cache behavior or transaction semantics to support more complicated distribution of devices, system resources, or an easy mechanism for applications to export synthetic file systems. Most of these protocols are tied to either TCP/IP or UDP/IP. While some support other transports, such as RDMA interfaces, none of these variants are in common use today.

The Linux network block device [NBD00], and its cousin the Ethernet block device [EBD02], and iSCSI [RFC3720] provide remote access to block and SCSI drivers. However, they provide no solution for various other system elements and don't support an organizational name space or a mechanism for coherent multi-user access.

As an application interface, 9P has many contemporaries. Remote Procedure Call (RPC) [Birr94] provides the closest contemporary providing both application level servers and a foundation layer for NFS -- however, it doesn't provide any of the semantics necessary to structure a functional name space.

Another interesting area to evaluate 9P is within cluster resource distribution and management. 9P provides a single protocol which can enable sharing of devices, file systems, and compute resources while also providing a convenience name space to organize these resources. While many other infrastructures attempt to provide these facilities [Fos96][Litz88], none provide the transparent ubiquity of Plan 9's model or an unified solution to resource distribution and management in a single protocol.

## Future Work

The largest missing piece from the Linux 9P2000 implementation is security. There is some rudimentary security provided by u9fs - either by using a rhosts file to specify what hosts are allowed to connect (similar to a stripped down exports file), or through a simple insecure challenge-response password system. Those concerned with security that want to use the existing implementation must tunnel through an ssh system which gives both user-authentication and data security.

However, Plan 9 has a rich set of security mechanisms which neither the Linux client driver or server use. One piece of future work would be to integrate these security mechanisms, plus the ability to encrypt and digest messages into the Linux drivers. Alternatively, or perhaps additionally, one could integrate Linux-style authentication schemes such as Kerberos and/or LDAP.

Another piece of future work would be to enable and tune 9P2000 to run on other transports beyond TCP/IP and pipes. Particularly interesting would be a port of 9P2000 to an RDMA [Mogu04] interface without an IP encapsulation. There is already an effort underway to port NFS to RDMA [Call02] and implement special RDMA file systems [Talp03].

NFS has the advantage of kernel-mode server. A port of the u9fs server application into the kernel with best efforts towards a zero-copy infrastructure should significantly increase the performance of the 9P2000 Linux implementation.

With the 9P2000 infrastructure now in place for Linux, a whole host of applications and synthetic file system gateways can be developed to provide Plan 9-style services and resource sharing. Gateway applications and drivers need to be developed for the IP stack, graphics systems, standard I/O console and other resources not currently exported to the Linux file system. Proxy drivers could be written to gateway character, block, and network driver operations across 9P2000 to devices on remote servers. Helper applications, like 9fs(1) and cpu(1) in Plan 9, need to be ported to Linux to allow easy access and use of the cluster resources made available with 9P2000. Finally, existing Plan 9 application ports need to be updated to use the native 9P file system support.

## Conclusions

Plan 9 was developed for distributed systems with three design principles in mind: represent all system and application resources as files, distribute those files using a simple protocol, and organize these distributed resources in a dynamic per-process private name space. With support for private name spaces and the implementation of 9P2000 now in the Linux 2.6 kernel, we can explore Plan 9 inspired distributed resource sharing and infrastructure within a commercial operating system.

The PostMark benchmark shows that 9P2000 performance for typical real-world workloads is superior to NFS, while the Bonnie benchmark shows NFS gets a significant benefit from time-delay write-back and loose read consistency on large static files. While performance for static files isn't a prime motivation for the 9P2000 protocol, we are confident that a loose consistency cache-layer similar to NFS could be implemented which would yield similar, if not better performance results. The performance results suggest that even without this cache layer, 9P2000 is a superior protocol for performing synchronous distributed file operations and demonstrates better performance for smaller files.

However, performance isn't the main advantage of 9P2000. The important thing to understand is the new paradigm of unified system resource sharing and distributed application design that it enables. The qualitative advantages of such a system have been documented in both the Plan 9 and Inferno technical papers. They can be seen, to a limited extent, in the synthetic file systems currently available under Linux (e.g. /proc and /sys). It is our hope that the availability of the paradigm in a widely available commercial operating system such as Linux will inspire more developers to experiment with this approach to distributed computing. The source code is available from http://v9fs.sourceforge.net under the GPL license.

## Acknowledgments

The 2.6 port of V9FS and performance analysis was supported in part by the Defense Advance Research Projects Agency under Contract No. NBCH30390004. The original V9FS research work by Ron Minnich was supported by DARPA Contract #F30602-96-C-0297.

Research conducted in the Cluster Research Lab at the Los Alamos National Labs was funded in part by the Mathematical Information and Computer Sciences (MICS) Program of the DOE Office of Science and the Los Alamos Computer Science Institute (ASCI Institutes). Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. Los Alamos, NM 87545 LANL LA-UR-04-7478.

## References

[9FAQ] "Plan 9 from Bell Labs FAQ", http://ask.km.ru/3p/plan9faq.html.

[9grid] A. Mirtchovski, R. Simmonds, R. Minnich, "Plan 9 -- an Integrated Approach to Grid Computing" International Parallel and Distributed Processing Symposium April 2004.

[9man] Plan 9 Programmer's Manual, Volume 1, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[Berry90] M. Berry, "Bonnie Source Code" http://www.textuality.com/bonnie/intro.html ,1990.

[Birr94] A. D. Birrell, B. J. Nelson, "Implementing Remote Procedure Calls", Proceedings of the ACM Symposium on Operating System Principles 1984.

[Call02] B. Callaghan, "NFS over RDMA" Proceedings of FAST 2002.

[CFS] David Howell "CacheFS" http://www.redhat.com/archives/linux-cachefs ,October 2004.

[CIFS] C. Hertel, "Implementing CIFS: The Common Internet File System", Prentice Hall PTR September 2003.

[Clus] "Clustermatic: A complete cluster solution", http://www.clustermatic.org

[EBD02] E. Van Hensbergen, F. Rawson, "Revisiting Link-Layer Storage Networking", IBM Technical Report #RC22602 2002.

[Fos96] I. Foster, C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit" The International Journal of Supercomputer Applications and High Performance Computing, 1996.

[HOW88] J.H. Howard, "An Overview of the Andrew FIle System", Proceedings of the USENIX Winter Technical Conference, Feb 1988.

[INF1] S.M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey, and P. Winterbottom, "The Inferno Operating System", Bell Labs Technical Journal Vol. 2, No. 1, Winter 1997.

[Katch97] J. Katcher, "PostMark: a New Filesystem Benchmark." Technical Report TR3022, Network Appliance 1997.

[Kill84] Tom Killian, "Processes as Files", USENIX Summer Conf. Proc. , Salt Lake City June, 1984.

[LISA03] D. Ellard and M. Seltzer, "New NFS Tracing Tools and Techniques for System Analysis" Large Installation System Administration Conference, 2003.

[Litz88] M. Litzkow, M. Livny, M. Mutka, "Condor: A Hunter of Idle Workstations" Proceedings of the 8th

International Conference of Distributed Computing Systems", 1988.

[Love03] Robert Love, “Linux Kernel Development”, Sams Publishing, 800 E. 96th Street, Indianapolis, Indiana 46240 August 2003.

[LPL] “Lucent Public License Version 1.02”, http://cm.bell-labs.com/plan9dist/license.html

[Majo94] D. Major, G. Minshall, and K. Powell, “An Overview of the NetWare Operating System”, Proceedings of the 1994 WInter USENIX Pages 355-72, January 1994.

[Mogu04] J. Mogul, “TCP offload is a dumb idea whos time has come” Proceedings of HotOS IX 2004.

[NBD00] P. T. Breuer, A. Marin Lopez, and A. G. Ares, “The Network Block Device”, Linux Journal Issue 73 May 2000.

[NFS4] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, et. al, “The NFS Version 4 Protocol” Proceedings on the 2nd international system administration and networking conference, 2000.

[OSI] “Open Source Initiative”, http://www.opensource.org"

[P903] “Plan 9 From Bell Labs Fourth Release Notes”, http://plan9.bell-labs.com/sys/doc/release4.html , June 2003.

[Pike00] Rob Pike, “Plumbing and Other Utilities”, Plan 9 Progreammer's Manual Vol 2. pp 219-234.

[Pike04] R. Pike, “Rob Pike Reponds” Slashdot Interview October 18, 2004.

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, “Plan 9 from Bell Labs”, UKUUG Proc. of the Summer 1990 Conf., London, England, 1990.

[Pike91] Rob Pike, “8½, the Plan 9 Window System”, USENIX Summer Conf. Proc., Nashville, June, 1991, pp. 257-265

[Pike94] Rob Pike, “Acme: A User Interface for Programmers”, USENIX Proc. of the Winter 1994 Conf., San Francisco, CA.

[plan9port] Russ Cox, “Plan 9 from User Space”, http://swtch.com/plan9port

[PPTTW93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, “The Use of Name Spaces in Plan 9”, Op. Sys. Rev., Vol. 27, No. 2, April 1993, pp. 72-76.

[PrWi95] Dave Presotto and Phil Winterbottom, “The IL Protocol”, Plan 9 Programmer's Manual, Volume 2, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[RFC793] RFC793, Transmission Control Protocol, DARPA Internet Program Protocol Specification, September 1981.

[RFC1151] RFC1151, Reliable User Datagram Protocol (version 2), Internet Engineering Task Force, April 1990.

[RFC1331] RFC1331, The Point-to-Point Protocol, Internet Engineering Task Force, May 1992.

[RFC3720] RFC3720, Internet Small Computer Systems Interface, Internet Engineering Task Force, April 2004.

[SATY90] M. Satyanarayanan, “Coda: A Highly Available File System for a Distributed Workstation Environment” IEEE Transactions on Computers, 1990.

[Sun89] Sun Microsystems, “NFS: Network file system protocol specification”, RFC 1094, Network Information Center, SRI International, March, 1989.

[Talp03] T. Talpey, “The Direct Access File System” Proceedings of FAST 2003.

[thomo95] Ken Thompson, “The Plan 9 File Server”, Plan 9 Programmer's Manual, Volume 2, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[VITA] “Vita Nuova Home Page”, http://www.vitanuova.com

[v9fs] Ron Minnich, “Plan 9-style File System for Linux/BSD”, http://sourceforge.net/projects/v9fs