The PODOS File System – Exploiting the High-Speed Communication Subsystem

Sudharshan Vazhkudai Tobin Maginnis

Department of Computer and Information Science
University of Mississippi
302, Weir Hall, University, MS 38677, USA

Abstract

Performance Oriented Distributed Operating System (PODOS) is a clustering environment, being built on a monolithic Linux kernel. PODOS augments very few components to the Linux kernel in order to make it distributed. These minimal components are the Communication Manager (CM), the Resource Manager (RM), the PODOS File System (PFS) and the Global IPC (GIPC). Each one of these components are designed and implemented with key performance benefits. They are designed to exploit the basic Linux operating system in numerous ways.

In this paper we discuss the design, implementation and performance of the PODOS File System (PFS). We discuss a combination of the overall design and implementation strategies and their implications on the distributed file system usage and performance. These include a ***Hybrid Naming Scheme*** *that strikes a balance between transparency and performance; an* ***Assumed-mount****s strategy which obviates the need for expensive remote file systems setup; a* ***Lazy-update*** *mechanism which modifies the Unix file sharing semantics by delaying updates to remote nodes; the use* ***of state-full kernel threads*** *as against state-less servers; the tweaking of the Linux VFS layer to* ***map traditional i-node operations*** *onto remote file access requests, thereby tightly integrating the PFS with the Linux file system; and the use of* ***read-aheads*** *for improved performance.*

The paper further discusses how the PFS, in conjunction with the underlying high-speed communication subsystem, that short-circuits the network protocol stack and further multiplexes virtual-circuits across multiple network interfaces, provides an efficient clustering environment for file sharing.

Keywords: *Distributed File Systems, Clustering, Distributed Operating Systems.*

1.0 Introduction

A variety of clustering systems exists, each with its own design goals and problem domain. In general, clustering systems tend to attack issues like performance, resource sharing, reliability, transparency, etc. Most of these design goals are conflicting and can never be accomplished in one system without some level of compromise [1]. Systems like Condor [2], Beowulf [3] provide high-performance environments while systems like Amoeba (distributed operating system) [4], tend to provide a good resource-sharing environment.

With these issues in mind, we are designing a distributed system, PODOS [5], an experimental Linux [6] cluster (being developed at the University of Mississippi). The primary intent is to explore the performance capabilities of a clustering system, but at the same time provide a good resource-sharing environment. Furthermore, we try to minimize the additions to the basic operating system [7]. Each node in the PODOS cluster is a monolithic Linux kernel comprising of the following components:

The *Communication Manager (CM)* handles remote communication in the PODOS by using a custom protocol to interact with peer CMs in the cluster. Higher-level layers (RM, GIPC, PFS) use the CM to talk to their peer components in the cluster [7].

The *Resource Manager (RM)* in each node maintains global system state information, i.e., information about each node in the cluster. The RM makes use of the CM to transmit and receive system information in a broadcast or a piggybacked fashion among its peers [7].

The *Global Inter Process Communication (GIPC)* provides a mechanism with which processes can communicate in PODOS, by allocating a global PID (GPID) for every process in PODOS so that processes can be uniquely identified. GIPC further provides communication primitives for processes to communicate among themselves [7].

The *PODOS File System (PFS)* extends the basic operating system file capabilities to support distributed file access. Processes will be able to recognize non-local file names and invoke the PFS. PFS local-to-remote requests will be carried out by simply invoking the CM [7].

In the following sections, we will present an argument for the purpose of yet another distributed file

system, by discussing the features of popularly used distributed file systems, their drawbacks, and how the PFS attempts to minimize these drawbacks. We then explain the overall structure and architecture of the PFS and finally present a few performance results.

2.0 Distributed File Systems

There are a number of distributed and network file systems, each with its own design goals. In this section, we will briefly look at two of them, the NFS [8] and Coda [9].

NFS is a network file system by Sun Microsystems, designed for a network of computers. NFS works in both LAN and WAN environments. NFS uses a standard networking protocol called Remote Procedure Calls (RPC) [10], which is built on sockets and UDP [11]. NFS is based on the virtual i-node architecture, wherein a virtual i-node is constructed for each remote file. NFS uses state-less servers that increase their reliability but make them slower. NFS performs read-aheads and entire file caching [8].

Coda is a distributed file system from CMU. It uses the standard TCP/IP protocol. Coda is primarily intended to be a wide area distributed file system and implements an extensive caching mechanism for mobile and disconnected operation [9].

Before discussing the PFS let us briefly look at the communication protocol it uses, thereby justifying the need for yet another distributed file system.

3.0 Communication in PODOS

Communication in PODOS is handled by the CM. Higher-level PODOS layers (RM, PFS, GIPC) rely on the CM for packet transmission and reception. The CM in each node uses a specialized protocol to talk to its neighbors. The CM comprises of the following components [12]:

The *Communication Descriptor Table (CDT)* is the CM's interface to the other PODOS layers. The higher-level PODOS protocols register their packets with the CDT. The CM performs all the intricate details involved with packet transmission and timeouts. The CM also uses the CDT to construct a virtual-circuit (a connection for a longer duration), which helps in streamlining communication between peer components [12].

Nodes in PODOS have been configured with multiple Ethernet interfaces to increase the LAN bandwidth. The CM exploits this architecture by multiplexing virtual-circuits (open-read-write-read…-close calls) across these interfaces using a round-robin strategy. This is referred as *Transmission-Group* [12].

The *PODOS-packet protocol* short-circuits the traditional network protocol stack (the OSI model [11]) to send and receive packets. The CM directly interacts with the datalink layer (the Ethernet [13] driver) of the network stack, thereby bypassing all the other layers. Thus, the PODOS packets have a unique Ethernet protocol ID [12].

The above mentioned features of the CM provide an excellent high-speed communication environment upon which higher-level layers could be built [12].

4.0 Motivation for PFS

Since PODOS employs a high-speed communication mechanism, we needed a file system that could exploit this feature of PODOS. The network file systems discussed in section 2.0 are based upon the traditional TCP and UDP protocols that were primarily designed for wide area networks. For example, the NFS uses a networking standard called RPC, which is built upon UDP (TCP versions are available too.). RPC is another layer over traditional networking protocol, which definitely makes programming simpler and the system more reliable, but also increases the latency time. Added to this, NFS uses state-less servers [8], which would make it reliable but slower. We needed to minimize the layer overhead and thus needed a file system that could function in such an environment. Further, we realized that an efficient file system could be designed and implemented with high performance benefits. We required a file system for a cluster in a local area network. This led to the evolution of the PODOS File System (PFS).

Let us look at the design and implementation of the PFS.

5.0 The PFS

In this section, we will discuss the PODOS File System, its architecture and implementation.

5.1 Overall Structure

Each node in the PODOS cluster is named as *linus1*, *linus2*, *linus<n>*. Every node has its own unique file system, i.e., PODOS does not strive to provide a unified file system, but tries to provide a high-speed and efficient environment for sharing resources in other nodes. The following sections discuss a few important design decisions, which would dictate the manner in which the PFS would be used and would behave. They are:

- Naming Scheme
- Assumed Mounts
- Lazy Update Semantics

5.1.1 Naming Scheme

Every distributed file system has to have a naming convention that would help resolve local and remote file names. Traditionally, distributed systems have followed three approaches [4]:

1. Machine name + path name of the file.
2. Mounting remote file systems onto local file hierarchy.
3. A single unified name space.

In PODOS, we have adopted a hybrid approach between options 1 and 2. Remote file names have to be specified along with their machine names, but these machine names are tightly integrated into the local file system hierarchy as directories in order to facilitate easy access to remote files using traditional file system structure. For example, if "*miaow*" is a file that resides in the root directory of the node, "linus4", then this file can be accessed from any other node by simply specifying, "*/linus4/miaow*", where "*linus4"* is a directory under "*/"* in the local file system hierarchy. Typically, this directory could reside anywhere in the local file hierarchy. Hiding the machine names simply involves another layer of mapping (mapping machine names to local directories). It is a design tradeoff between performance and transparency.

5.1.2 Assumed Mounts

Every node in PODOS can access files and directories in every other node by specifying the machine name, followed by the path. Every node in the cluster is assumed mounted in every other node. No explicit mounting is necessary as required in NFS. NFS spends a lot of time creating virtual i-nodes (vnodes) for each remote file. This is essential for the design goals of NFS (stresses on reliability, supports wide-area mounts, etc.,) [8].

When an access is made to a file, "*/linus4/miaow*", the PFS, contacts the RM to check the validity of the node, "*linus4"*. The RM in turn contacts the SST to check if "*linus4"* is a valid host and if it is alive and finally returns its results to PFS. If RM returned a positive result, then the PFS proceeds to fetch the file. Thus, the PFS attempts to blend-in the remote file systems into the local file system hierarchy by treating machine names as implicit directories, and interpreting them appropriately. The entries *"/linus4"*, *"/linus5"*, etc., are created as directories in each node at startup.

5.1.3 Lazy Update Semantics

Typical distributed systems follow one of the following file sharing semantics [4]:

1. Unix Semantics, where updates are visible immediately.
2. Session Semantics, where updates are visible only after the file is closed.

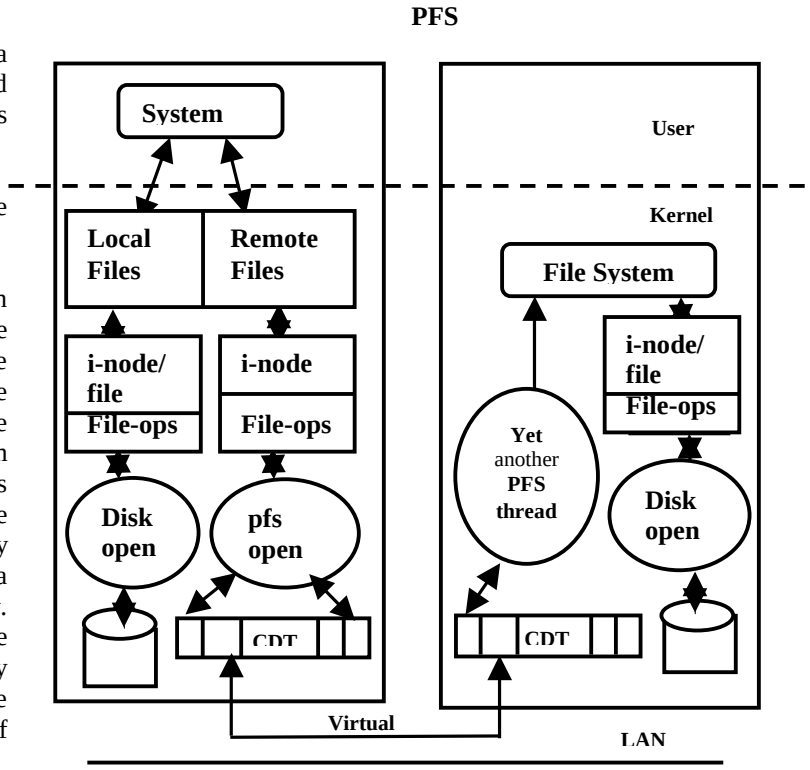


Figure 1

Following the Unix semantics is very costly in a distributed scenario and session semantics leads to consistency problems. Thus, we employ a Lazy Update Semantics, which is a slightly modified version of the Unix semantics, wherein updates are made visible after a certain size (typically 1K).

Let us look at the architecture and implementation of the PFS.

5.2 Architecture

In this section, we will discuss the following:

- Mapping i-node operations
- Kernel resident state-full threads
- Read ahead caching

These features help in making the PFS robust and highly tuned towards performance.

5.2.1 Mapping i-node Operations

The PFS is tightly integrated with the Linux file system and exploits its architecture for performance benefits (Figure 1). PFS uses the same file access primitives for both local and remote file operations. The PFS exploits the Linux operating system's virtual file system (VFS) [6] interface to implement remote file access. The VFS is an elegant mechanism that is responsible for Linux file system's flexibility and performance. The Linux operating system supports multiple file systems using the VFS

interface. The VFS layer provides hooks and generic interfaces which specific file systems could use. Specific file system implementations (ext2, MSDOS, etc.,) implement these interfaces and can be plugged into the hooks. We have modified the VFS layer to incorporate support for remote file fetches. This is achieved through the i-node operations mapping technique. The i-node operations mapping, done at the active end, is a technique with which the file operations of an i-node are mapped to methods that implement remote file accesses. Let us look at how this is accomplished.

Typically, in local file accesses, the VFS layer obtains the i-node of the file being fetched. This i-node is of type, *struct inode* *****. This structure contains all necessary information about the file, its creation time, modification time, etc. It also contains the VFS element, *i_op*, of type *struct inode_operations* *****. It contains an element, *default_file_ops*, which is of type, *struct file_operations* *****. This structure contains function pointers, which hold the addresses of the file system specific functions [6].

Once the VFS obtains the i-node, it simply makes a call *inode->i_op->default_file_ops->open(inode, f)* which invokes the file system specific open call. Inside the VFS layer, we map these i-node operations to functions that perform remote opens, reads and writes. The remote operations in the PFS are performed by *pfs_open()*, *pfs_read()*, *pfs_write()* and *pfs_close()*. The mapping is accomplished by: *inode->i_op->default_file_ops-open=&pfs_open;* this statement sets the function pointer to hold the address of the *pfs_open()* function. Read and write pointers are set as above (seeks could be done similarly). In short, the PFS uses the VFS architecture to channel remote file operation through traditional i-node structures. Thus, the advantages to this approach are:

- Speed
- Tight Integration with traditional file system.
- Same primitives for local and remote accesses.

We will discuss more about the functionality of the above mentioned methods in section 6.0. Let us look at kernel threads.

5.2.2 State-full Kernel Threads

A key design decision of the PFS is the use of kernel resident state-full threads. These threads are specialized kernel-level worker-bees of the PFS, started for each remote file request. These state-full threads reside at the passive end. State-full threads maintain the state information of the open files, like the file descriptor position. The use of state-full threads provides better performance and shorter messages over NFS that uses a state-less server. State-less servers are typically more fault-tolerant but are slower and result in longer messages (as filename and position in the file has to be sent with each read/write request).

The CM, upon receipt of a file fetch request, directs it to the PFS. The PFS spawns a thread (called *"Yet Another PFS"*) and keeps it in the kernel space. The reason behind having a kernel thread is to improve performance by minimizing the number of context switches that would result otherwise. The main PFS thread goes back to listening for further requests after creating the thread. Thus, it behaves like a concurrent server in processing remote requests. From then on, the kernel thread takes the responsibility of processing further requests with reference to the file. Once its properties are set, it opens the file using the local system's file-primitives. After opening the file the kernel thread returns an acknowledgement (and the first block if read operation) and suspends itself. It wakes up whenever further requests to that particular file arrive, processes it and goes back to suspended state. The kernel thread remains alive until the file is closed at the active end. When the file is finally closed at the active end, the kernel thread closes the file locally and performs an exit.

5.2.3 Read-ahead Caching

The PFS employs read-ahead caching to further improve the performance. It is traditional in distributed file systems to perform read-aheads. The PFS fetches data from the remote system in terms of 1K blocks and buffers it in the Communication Descriptor Table. As long as the length of the data requested is less than the length of the data in the CDT, the PFS avoids making a remote fetch. The moment the length requested is greater than the length of the buffered data in the CDT, a 1K block is fetched from the remote end. Read-aheads are typical in file systems (even etx2 performs read-ahead; NFS performs read-aheads and entire file caching). In general, read-aheads work on the principle that most file accesses are sequential, and drastically improve the file system performance [14].

Let us look at the implementation in detail by analyzing a remote file fetch.

6.0 Tracing a Remote File Fetch

In this section, we will trace the execution of a remote file fetch, thereby discussing the various components of the PFS and their interaction with the CM and the RM. The protocol used by PFS for peer-to-peer communication (PFS-PFS) is as follows:

1. **The open call at the active end**

- The active end initiates the remote request using an open system call in which it specifies the

filename along the remote host name. For example, *"/linus4/miaow"*.

- The PFS component in the VFS layer (*pfs_open*) contacts the RM, to check to see if *"linus4"* is a valid host in the cluster. The RM contacts the System State Table (SST) to verify details about *"linus4"*.

The fd-cdt map

- If *"linus4"* is a valid host in the cluster, the RM returns a TRUE value. The PFS component in the VFS layer then initializes a *remf* structure, of type, *struct remote_file* *****. The remote_file structure contains the following:

```
struct remote_file {
        ……..
        char *host;
        char *filename;
                /* the CDT index through which the remote
                file is accessed */
        int cdt_active_ind;
};
```

This structure is stuffed into the *private_data* field of the *file* structure. All open files in the Linux operating system are maintained in a doubly linked list, each node of which is of type *struct file* *****. The structure contains the following [6]:

```
struct file {
    ……..……..……..……..…
    struct file *f_next, *f_prev;
    struct inode * f_inode;
    struct file_operations * f_op;
    void *private_data;
};
```

This structure is typically allocated by the open call and subsequently used by read and write calls to refer to the specific file.

- The PFS builds a packet and then invokes the CM to make a CDT entry and transmit the packet to the remote end. Once it makes an entry, it copies the *cdt_active_ind* to *remf->cdt_active_ind* and returns. This index is important because, it saves the search time for subsequent read and write calls, i.e., read and write calls can directly map onto the CDT entry to refer to the file and not search through the table. In short, the file descriptor is directly mapped onto the CDT entry. The structure of the CDT is as follows [12]:

```
struct comm_desc_tab {
        struct DOS_pkt pkt;
        ……..……..……..……..
        struct interface if_vc;
        char read_ahead_buff[1024];
};
```

2. **Passive end's Response to open**

- At the passive end, the CM makes a CDT entry, establishes a virtual circuit and then invokes the PFS. The PFS spawns a kernel thread to process the file request and returns to listen to further requests. The thread spawned opens the file and returns the CDT index and its Global PID (address of the node + local PID) [5] along with the message (also returns the first block of data if it is a read operation) and suspends itself.

3. **read/write calls at the active end**

- Subsequent read/write calls at the active end go through the *pfs_read* and *pfs_write* methods in the PFS. The read/write calls directly map onto the CDT entry corresponding to the file based on the *cdt_active_ind* in the *remote_file* structure. The pfs_read and pfs_write access the CDT entries to fetch and write data. The pfs_read checks to see if the data in the *read_ahead_buff* (in the CDT) is greater than the requested length. If so, it copies the data from the CDT to the USER space into the read buffer specified along with the read call. In the case where the requested length is greater than the CDT buffer length, the PFS builds a packet (requesting for a 1K block), registers it with the CDT, and invokes the CM for transmission and blocks for the arrival of the data. Writes are typically stored in the CDT and flushed to the remote end after the CDT buffer reaches 1K.

4. **Passive end's response to read/write**

- When the message arrives at the passive end, the CM realizes that the packet is on its way to the specific kernel thread and wakes it up. The thread reads or writes data, builds a packet and registers it with the CDT and invokes the CM to transmit the message. It then suspends itself waiting for further requests and remains alive until the file is closed.

5. **Active end closes the file**

- After the active end is done processing, it closes the file by making a *close* call, which is translated into a *pfs_close* call, which sends a message to the kernel thread at the remote end directing it to exit.

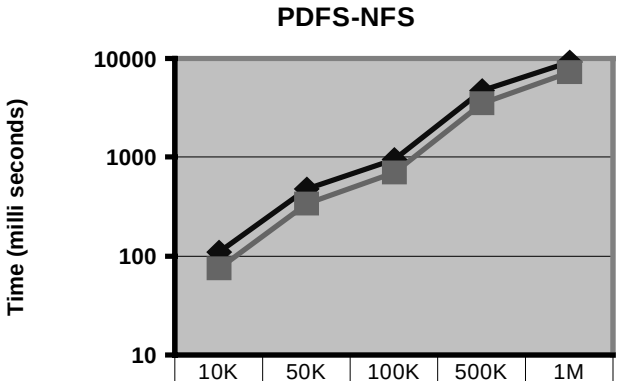
In the following section, we will discuss the performance results of the PFS.

7.0 Performance

In this section, we present the preliminary performance results obtained by analyzing the PFS against NFS, network file system. All experiments were conducted with the following setup:

- Using Pentium 133MHZ machines, with 32MB of RAM, interconnected with Ethernet [13] adapter cards (16 and 8 bit cards).
- With default NFS block size of 1K.

- In the case of NFS, the time taken to fetch the file, the first time is considered.
- All time values presented are averaged milliseconds.

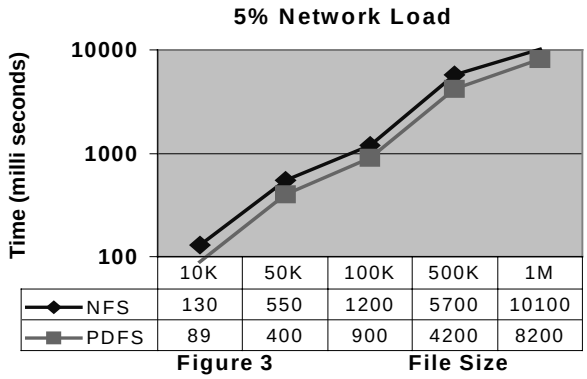


Let us look at the experiments conducted.

Experiment #1

In this experiment, we compare the average time taken, by NFS and PFS, to fetch files of different sizes. We considered standard file sizes of 10K, 50K, 100K, 500K and 1M. The graph depicts the results. The graph (Figure 2) uses a logarithmic scale for the y-axis, to clearly show the performance gain achieved for all file sizes. The PFS consistently performs better than NFS for all file sizes. Usually, in file systems, most files accessed are less than or equal to 10K [14]. PFS has a considerable performance gain of around 30 to 40 msecs over NFS for this file size.

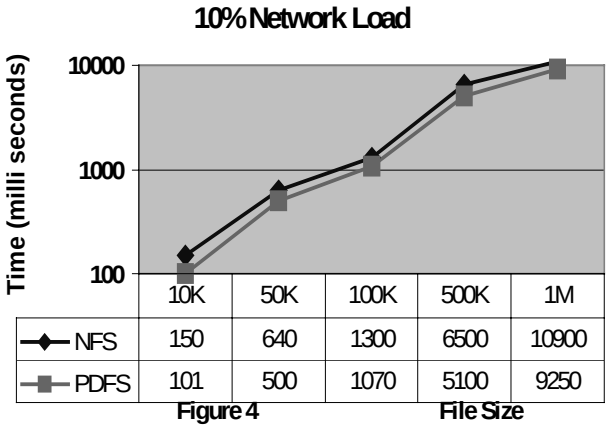
The time taken to fetch files are calculated as follows: In general, the time taken, by an active end, to read or write a block is the sum of the time spent by the kernel in the VFS layer and the suspend time. The suspend time is the sums of the dispatch time, the propagation time and the thread processing time. The dispatch time is sums of the time taken to build a PODOS packet, the time consumed to make a CDT entry and the time taken to transmit the packet and is usually around 200 to 300 microseconds. The propagation time is the sums of the forward propagation, the backward propagation and the time taken to filter out PODOS packets at both active and passive ends. The propagation time varies with network load. The transmit time is the time taken by the driver to allocate network buffers, the time taken to build Ethernet packets and the time taken to place the packet on the media. This is usually around 400 microseconds. The thread processing time is the time taken to wake-up both the active end process (that made the request) and the passive end thread, the time



taken by the thread to make a local read/write and its dispatch time. This is a couple of hundred microseconds.

Experiment #2

The next couple of experiments were designed to test the effective utilization of the high-speed communication bed, by the PFS. In the previous experiment, all file fetches were made on the same network interface. In this experiment, the PFS makes use of the "Transmission-Group" feature of the CM, wherein virtual-circuits are multiplexed across multiple network interfaces [12]. To test this further, we studied the performance of this setup under varied network loads. These results were compared against NFS, which makes all file fetches on a single network



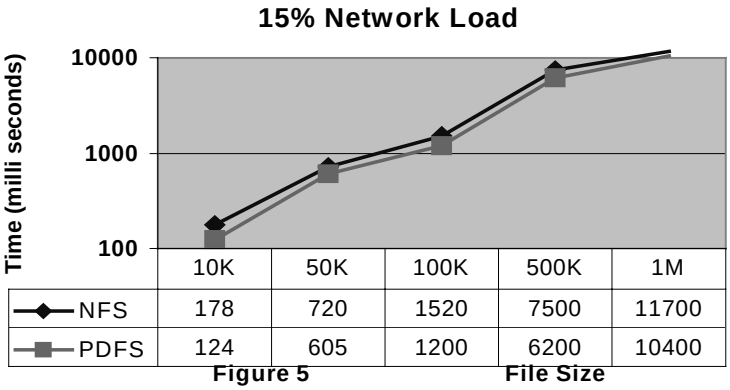


Figure 5

interface (immaterial of the load). The results under 5%, 10%, 15% and 20% network loads are depicted in Figure3, Figure4, Figure5 and Figure6. Since the CM distributes virtual circuits uniformly using a round-robin mechanism, the overall PFS performance is improved when compared against NFS. At all loads, the PFS consistently has a 30 to 50 msecs gain over NFS for file sizes around 10K, a 100 to 200 msecs gain for file sizes around 50K, and a couple of hundred msecs gain for file sizes around 100K. When clustering activities tend to be intense, these are considerable performance gains, which would improve overall system throughput.

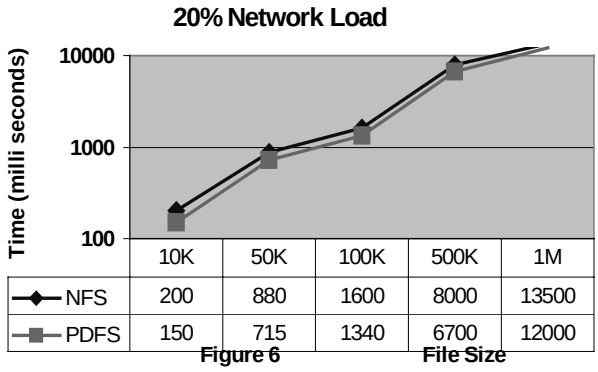


Figure 6

Listed below are the average times, in milliseconds, for the system calls: open, read and close. These were computed by studying the above results.

	NFS	PFS
open	2.4	1.5
close	0.052	0.03
read	10	6

8.0 Conclusion

We have presented PFS, a distributed file system for a cluster of workstations. We have described how PFS builds an efficient file-sharing environment on top of the high-speed communication subsystem. We then described the design and implementation of PFS, discussing its key features. A few design features discussed were: a hybrid naming scheme, Lazy updates and Assumed-mounts. Implementation features include: i-node operations mapping, read-aheads and state-full kernel threads. We then presented a performance analysis of the PFS comparing it with NFS, thereby showing the performance gains achieved.

References

[1] A. Goscinski, "Distributed Operating Systems The Logical Design," Addison-Wesley Publishing Company, 1991.

[2] M.J. Litzkow, M. Livny, and M.W. Mutka, "Condor-A Hunter of Idle Workstations," *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988, pp. 104-111.

[3] P. Merkey, "Beowulf Project at CESDIS," *http://beowulf.gsfc.nasa.gov*, 1994.

[4] A.S. Tanenbaum, "Distributed Operating Systems," *Prentice Hall*, 1995.

[5] S. Vazhkudai, P.T. Maginnis, "Performance Oriented Distributed Operating System (PODOS)," *Technical report Computer Science Department, University of Mississippi*, May 99.

[6] L. Torvalds, "Linux Kernel Site," *http://www.kernel.org*, May 1993.

[7] P.T. Maginnis, "Design Considerations for the Transformation of MINIX into a Distributed Operating System," *Proceedings of the 1988 ACM 16th Annual Computer Science Conference*, 1988, pp. 608-615.

[8] S.M. Inc., "NFS: Network File system Protocol Specification," *SRI Network Information Center*, vol. RFC 1094, Mar. 1989.

[9] P. Braam, "The Venus Kernel Interface," *http://www.coda.cs.cmu.edu*, Mar. 1998.

[10] A.D. Birrell, B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. On Computer Systems*, Feb 1984, vol. 2, pp. 39-59.

[11] A.S. Tanenbaum, "Network Protocols," *ACM Computing Surveys*, vol. 13, no. 4, Dec. 1981, pp. 453-489.

[12] S. Vazhkudai, P.T. Maginnis, "A High Performance Communication Subsystem for PODOS," *Proceedings of the 1st IEEE International Workshop on Cluster Computing*, December 99, pp. 81-91.

[13] D.R. Boggs, J.C. Mogul, C.A. Kent, "Measured Capacity of an Ethernet: Myths and Reality," *ACM SIGCOMM'88 Symposium*, vol. 18, no. 4, Aug. 1988, pp. 222-234.

[14] M. Satyanarayanan, "A Study of File Sizes and Functional Lifetimes," *Proc. 8th Symp. On Operating Systems principles*, ACM, 1984, pp. 96-108.