ELSEVIER

Future Generation Computer Systems 18 (2002) 335–352

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.com/locate/future

PODOS — The design and implementation of a performance oriented Linux cluster

Sudharshan Vazhkudai [a], Jeelani Syed [a], Tobin Maginnis [b,*]

[a] *The University of Mississippi, P.O. Box 1943, University, MS 38677, USA*
[b] *Department of Computer and Information Science, The University of Mississippi, 302, Weir Hall, University, MS 38677, USA*

Abstract

PODOS is a performance oriented distributed operating system being developed to harness the performance capabilities of a cluster-computing environment. In order to address the growing demand for performance, we are designing a distributed operating system (DOS) that can utilize the computing potential of a number of systems. Earlier clustering approaches have traditionally stressed more on resource sharing or reliability and have given lesser priority to performance.

PODOS adds just four new components to the existing Linux operating system to make it distributed. These components are a Communication Manager (CM), a PODOS Distributed File System (PDFS), a Resource Manager (RM), and Global Interprocess Communication (GIPC). This paper addresses the design and implementation of the various components of the PODOS system. © 2002 Published by Elsevier Science B.V.

Keywords: Linux clusters; Distributed operating systems; Distributed file system; Communication protocol; Parallel Ethernets; IPC

1. Introduction

A distributed operating system (DOS) is basically the cooperation among a group of machines interconnected by a network such that the group of machines appears to the user as a single operating system. With DOS, users are neither aware of where their files are stored nor they are aware that remote machines may execute their programs. All resources within the network are managed in a global fashion using global mechanisms rather than local mechanisms [1].

A group of machines could cooperate for a variety of reasons. A few of them are: (1) resource sharing; (2) performance enhancement; (3) reliability; (4) fault tolerance; (5) transparency [1]. Tens of DOSs have been designed and implemented with various goals. Most distributed system designs are willing to compromise on performance. On the other hand, systems that are designed to be performance oriented make no attempt to provide a single system image. They provide a high-performance computing environment (examples of clustering systems: Condor [2], Beowulf [3], etc.; e.g., DOS: Amoeba [4]). High-performance computing environments are designed to solve one class of problems, whereas a system like PODOS is designed as a general high-performance computing solution.

With these issues in mind, we are designing a distributed system, PODOS, an experimental Linux [5] 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

* Corresponding author. Tel.: +1-601-232-5357; fax: +1-662-915-5623.
E-mail addresses: chucha@john.cs.olemiss.edu (S. Vazhkudai), jmsyed@olemiss.edu (J. Syed), ptm@pix.cs.olemiss.edu (T. Maginnis).

0167-739X/02/$ – see front matter © 2002 Published by Elsevier Science B.V.
PII: S0167-739X(01)00055-3

environment. Furthermore, we try to minimize the additions to the basic operating system [6]. Each node in the PODOS cluster is a monolithic Linux kernel. The PODOS design has a number of key performance benefits. A few of these are:

1. PODOS builds upon a highly robust and performance oriented monolithic Linux kernel.
2. PODOS adds very few components to the basic Linux operating system, thereby maintaining a simple design.
3. Each of these components is designed to achieve high performance. For example, the CM uses a custom high-speed protocol and the PDFS is tightly glued to Linux's file system to speed up remote file fetches.

PODOS comprises of the following components:

Communication Manager (*CM*). The 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 and PDFS) use the CM to talk to their peer components in the cluster [6].

Resource Manager (*RM*). The 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 [6].

Global Inter-Process Communication (*GIPC*). The 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 [6].

PODOS Distributed File System (*PDFS*). The PDFS 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 PDFS. PDFS local-to-remote requests will be carried out by simply invoking the CM [6].

Let us look at the network architecture in PODOS.

2. The PODOS network topology

PODOS has a special network topology that aids in implementing an efficient and a high-performance communication mechanism [7]. Let us look at a model of the PODOS network topology. Fig. 1 gives an overview of the PODOS network architecture.

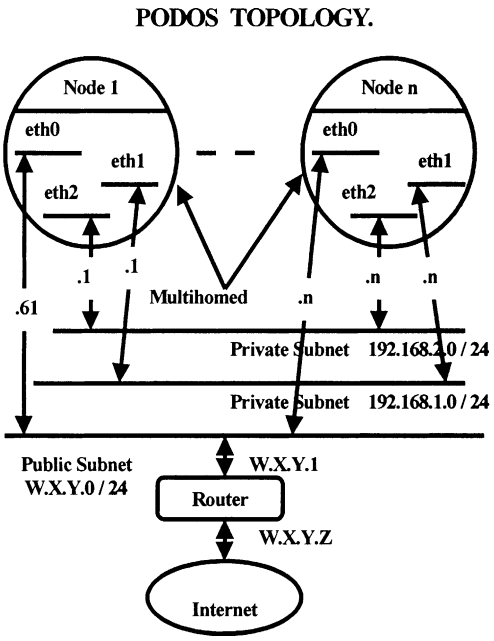

Fig. 1. PODOS network topology.

PODOS uses an Ethernet network interface as the communication media. Each node represents a machine, where node 1, node 2, etc., is machine 1, machine 2, etc., respectively. PODOS can use as many interfaces as the system supports. Our current implementation has three Ethernet interfaces, namely `eth0`, `eth1`, and `eth2`. Each node is connected to the public subnet, W.X.Y.0, through the primary interface, `eth0` and thus gets an IP address, W.X.Y.*n*, where *n* is between 1 and 254. Further to this, each node has two more interfaces, `eth1` and `eth2`, which are, connected to private subnets 192.168.1.0 and 192.168.2.0, respectively, and thus get two more IP addresses, 192.168.l.*n* and 192.168.2.*n*. Thus, each machine in PODOS is a multihomed host. Private subnets in PODOS are configured in such a way that machines can communicate only through the interface pairs `eth0`–`eth0`, `eth1`–`eth1`, and `eth2`–`eth2`. More specifically, packets on `eth1` can only go to `eth1` on another machine. This can be easily observed from Fig. 1, since the network 192.168.1.0 connects only `eth1`'s in all machines.

Similarly, the network 192.168.2.0 connects `eth2`'s. The Transmission-Groups feature uses these interfaces in a round-robin fashion [7].

In the following sections we will discuss the various components of PODOS.

3. Communication in PODOS

Communication in PODOS is handled by the CM. Higher-level DOS layers (RM, PDFS, GIPC, etc.) 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:

- PODOS-packet protocol;
- Communication Descriptor Table (CDT);
- Transmission-Groups.

Fig. 2 demonstrates the relationship among these subsystems. To the left, in Fig. 2, is the traditional network protocol stack, the Open Systems Interconnection (OSI) [8] model of networking. To the right, in Fig. 2, are the PODOS components. The communication subsystem is depicted in more detail. The CM comprises of three components, namely the CDT, the Transmission-Groups, and the PODOS-packet protocol. The CDT is an interface for higher-level PODOS layers. Higher-level layers typically make entries with the CDT. A Transmission-Group algorithm is applied to each entry in the CDT. And finally, the PODOS-packet protocol transmits the packet using

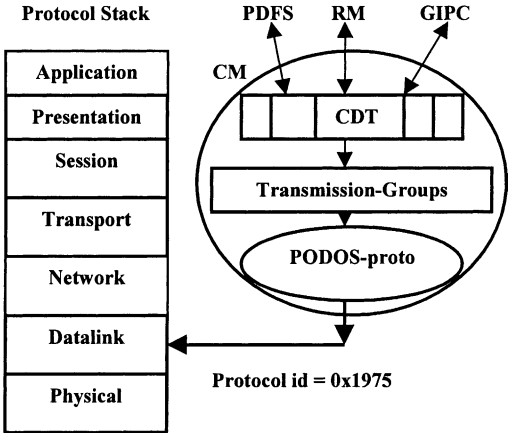

Fig. 2. Communication subsystem.

the datalink layer of the OSI model. The RM, PDFS, and GIPC use the CM to communicate with peer entities in the PODOS cluster. We will now look at the various components of the CM, depicted in Fig. 2.

3.1. PODOS-packet protocol

The PODOS-packet protocol is at the very bottom of the CM. It provides primitives for transmitting and receiving PODOS packets. The PODOS-packet protocol has evolved from a very rudimentary structure [7]. In this section, we will describe the protocol briefly.

Since our primary goal is performance and the typical DOS bottleneck is network bandwidth, we needed an efficient communication mechanism that could speed up packet transmission and reception. We needed something other than the traditional networking protocol (left side of Fig. 2). The traditional protocol consists of several layers and each layer has its own headers and error checking. However, traditional protocols also contain a lot of detail to accommodate many types of network configurations [8]. Thus, we designed and implemented a DOS packet (PODOS packet), which bypasses the traditional network protocol stack [7].

In Fig. 2, we can see how the CM resides above the datalink layer (Ethernet driver). From Fig. 2, it is also evident that how the CM has moved away from the traditional network layers and has fewer overheads. Our approach here is to have the CM interact with the datalink layer (Ethernet driver) to transmit and receive packets. In short, the CM will have to transmit and receive packets that bypassed the network protocol stack. This would mean that the CM packets would have to have a separate protocol ID [7], one that is different from IP, ICMP, etc.

3.1.1. The PODOS-packet structure

Table 1 describes the PODOS-packet structure and also illustrates the importance of each field [7].

Now let us look at the CDT.

3.2. The CDT

The CDT is the CM's interface to the other PODOS layers. The CM shields the higher-level PODOS layers by performing all the intricate details involved with packet transmission. The higher-level PODOS

Table 1
PODOS-packet structure

struct PODOS_pkt {	
char from_pid[8];	The global pid of the process from whom the packet is originating
char to_pid[8];	The global pid of the process to whom the packet is being sent
char ctrl_info;	This is a 1-byte field that is used by the CM to differentiate PDFS, RM and IPC packets. It uses the least significant 3 bits. Higher-level protocols use the most significant 5 bits. For example, the PDFS would use it to differentiate open, read, write, close, etc.
int cdt_active_ind;	The CDT index at the active end (the end that originated the connection)
int cdt_passive_ind;	The CDT index at the passive end (the end that is accepting the connection)
int wake_active_passive;	Since the CM code is the same for active and passive ends, it needs to know which process to wake up
int length;	The length of the data being sent
char data[1400];	The data
};	

protocols register their packets with the CDT. The CM picks up packets from the CDT and transmits the packets. The CM also uses the CDT to construct a virtual circuit, which helps in streamlining communication between peer components. The CM also maintains a simple timeout mechanism by which it can keep track of errors and retransmissions. Thus, the CM maintains a simple and elegant protocol for packet delivery [7].

3.2.1. The CDT structure

A fully functional CDT has marked a milestone in the evolution of the CM. The higher-level PODOS layers fill in a CDT entry and invoke the CM. The CM picks up the PODOS packet from the CDT and transmits it. A typical entry in a CDT is described in Table 2 [7].

3.2.2. Classes of CDT entries

CDT entries can be classified into two categories based on the time spent in the table. They are:

- *Ephemeral*. These entries are short duration entries and release the CDT slot once the packet has been transmitted. An example of such a request that is short-lived is the RM query. The RM periodically broadcasts system information to all the nodes in the cluster. It does not wait for the arrival of any packet. Whenever a broadcast message arrives the CM invokes the RM. Thus the RM would register its packet in the CDT and invoke the CM. Once the CM transmits the packet, it releases the CDT entry corresponding to the RM query [7].
- *Virtual Circuits* (*VC*). When a higher-level PODOS protocol wishes to communicate with its peer for a longer duration, it usually requests the CM to establish a virtual circuit. The virtual circuit is nothing but {cdt_active_index, cdt_passive_index} pair. When a higher-level PODOS layer (at the active end) requests such an entry, the CM makes a CDT entry and transmits the packet. The peer CM (at the passive end) receives the packet, makes a CDT

Table 2
CDT structure

struct comm_desc_tab {	
struct PODOS_pkt pkt;	The PODOS-packet structure is embedded in the CDT
char *to_pid[8];	Higher-level layers specify the process pid in the remote machine to which the packet is sent. This could be a group of processes to support group communication
int to_pid_cnt;	The count of the processes referencing this CDT entry
int need_reply;	Whether the CDT entry should be held for a longer time. For example, RM requests are typically short lived as compared to PDFS or GIPC requests
Char *hostname[20];	The name of the machine to which the packet is being sent. Could be list of machine names (for group communication)
int host_cnt;	The count of machine names
Struct interface if;	The interface structure (will be discussed later)
};	

entry and then invokes the appropriate layer. Henceforth, any communication between these two layers would go through this virtual circuit. This helps in streamlining the subsequent flow of packets [7].

Now let us look at how PODOS handles Transmission-Groups.

3.3. Transmission-Groups

Transmission-Groups is a suite of algorithms that multiplex packets across multiple network interfaces. Transmission-Groups exploit parallel networks connected among distributed computers and thereby match the external aggregate network bandwidth with internal memory bandwidth. The CM employs Transmission-Groups to achieve further performance gains over the PODOS-packet protocol. The PODOS cluster is configured in such a fashion that the suite of Transmission-Group algorithms can exploit the network architecture. Each node in PODOS has been configured with multiple Ethernet interfaces (Fig. 1). This increases the local area network (LAN) bandwidth and effectively utilizes the communication media. Once the higher-level PODOS layers register their packets with the CDT, the CM decides which interface the packet should go through. This decision making is Transmission-Groups [7].

Transmission-Group algorithms are applied only to PODOS packets in the cluster. PODOS packets can travel on any one of the three interface pairs (the following denotes active-end-interface–passive-end-interface: `eth0`–`eth0`, `eth1`–`eth1`, and `eth2`–`eth2`). Whereas regular network traffic (IP, ICMP, and IGMP packets) continues to go through `eth0` only. If we wish to multiplex those packets too, then we would have to employ an algorithm similar to Transmission-Group suite at a higher level in the network protocol stack.

Now let us look at Transmission-Groups in more detail.

3.3.1. Transmission-Group suite

The Transmission-Groups is a suite of algorithms, each based on a different goal. We provided a set of routines (round-robin, load based), so that one may select the best algorithm that suits their requirements. The Transmission-Group routine is held as a function pointer in the interface structure (described in Table 3) in the CDT entry. When a higher-level layer wishes to transmit a packet it makes an entry in the CDT and invokes the CM. The CM initiates the Transmission-Group algorithm by invoking the function pointer. The function pointer is set during system initialization.

The following paragraph describes a simple round-robin Transmission-Group algorithm that multiplexes virtual circuits across multiple interfaces.

- *Round-robin with VC multiplexing*. PODOS employs a simple round-robin mechanism to multiplex packets across multiple network interfaces. But since, multiplexing packets would result in ordering and sequencing issues, PODOS multiplexes virtual circuits. This implies that all packets resulting from a virtual circuit would be transmitted on a particular interface pair. In short, PODOS employs a *"1-Interface-for-all-Packets-in-a-VC"* rule. This ensures that packets reach their destination in order and saves us the trouble of ordering them. Ephemeral packets (packets resulting from an Ephemeral CDT entry, a RM packet) would be transmitted on the current interface (maintained by the Transmission-Group routine). Let us look at this algorithm in more detail.
- *Implementation*. The round-robin algorithm maintains an interface counter which it increments for every virtual circuit. This counter is reset once it reaches IF_MAX. The algorithm differentiates

Table 3
Interface structure

struct interface if {	
char active_interface[6];	The hardware address of the interface at the active end
char passive_interface[6];	The hardware address of the interface at the passive end
int (*elect_interface)(struct device*);	Transmission-Group algorithm held as a function pointer. This is the function that round-robin's packets
};	

virtual circuits from ephemeral entries by looking at the control byte of the packet. Let us discuss the implementation of VC multiplexing with reference to a PDFS remote file-write. The local PDFS wishes to write a file to another node by contacting its peer entity in the other node. This PDFS file-write request is characterized by an open call, a sequence of write calls and then a close call. This is a typical virtual circuit. Once the PDFS decides to write a file to another node, it builds a PODOS packet, makes a CDT entry and invokes the CM. Embedded in the CDT entry is the interface structure. Below is Table 3 describing the interface structure.

Having the interface structure in the CDT entry is the key to the algorithm. Each entry will have such a structure and packets can be transmitted to/from the address specified in the interface structure. Once an entry is made in the CDT, the Transmission-Group routine corresponding to that CDT entry is launched by calling cdt[*i*]->elect_interface( ), where *i* denotes the particular entry. The Transmission-Group algorithm maintains a simple round-robin strategy with which it multiplexes virtual circuits.

Now, when the higher-level layer (PDFS) registers subsequent packets in the CDT, the CM just inspects the interface structure of the CDT entry and transmits the packet to the passive_interface address on the active_interface address. Thus, with this approach, the Transmission-Group routine is invoked only once per virtual circuit (or CDT entry) and all packets belonging to one virtual circuit are transmitted on the same interface pair. Thus, the algorithm makes a basic assumption that "*virtual circuits have the potential to generate a lot of traffic on the interface*". Hence, the algorithm attempts to uniformly distribute virtual circuits across the three interfaces. Ephemeral entries do not contribute much to the interface load and thus the algorithm does not worry about such packets [7].

3.4. Performance

In this section, we analyze the performance capabilities of PODOS protocol and its variants. We further compare them with the traditional TCP/IP protocol and present the preliminary results based on this analysis.

In order to study the behavior of these two protocols, we conducted a series of experiments, each with a different objective, and measured the average round trip time (average RTT) in each case. All our experiments involved comparing variants of the PODOS protocol with the traditional networking protocol under various network loads [9]. Each experiment:

1. Computes the average RTT of packets using 10 sets of 100 packets each.
2. It discards the maximum and the minimum set and then computes the average.
3. Transmits each packet with a 1 s delay.

Let us look at each experiment in detail [7].

3.4.1. Experiment 1

In this experiment, we compare the RTT's of a simple PODOS protocol with a typical socket read/write call of the TCP/IP. The experiment transmits 10 sets of 100 packets each. Each packet is 64 bytes long and is transmitted with a 1 s delay. Fig. 3 depicts the performance gain achieved. From the graph it is evident that PODOS protocol out-performs traditional networking protocol and is almost twice as fast. The RTT difference is substantial at higher network loads. The graph depicts RTT's for loads up to 35–40%. This is because 35–40% network load is a substantial network load and the system is saturated at that load. In normal circumstances, the multiprogramming level would further increase the RTT thus making the load substantial [7]. The RTT for the PODOS protocol is given in Eq. (1).

$$\begin{aligned} T_{\text{PODOS}} &= T_{\text{build-PODOS-pkt}} + T_{\text{xmit}} + T_{\text{propagation}}, \\ T_{\text{xmit}} &= T_{\text{alloc-net-buff}} + T_{\text{build-Ethernet-pkt}} + T_{\text{drv-xmit}}, \\ T_{\text{propagation}} &= T_{\text{forward-propagation}} + T_{\text{backward-propagation}} \\ &\quad + T_{\text{filter}}. \end{aligned} \tag{1}$$

The RTT for a PODOS packet is the sum of the times taken to build a PODOS packet, build an Ethernet packet, the driver transmit time (the time taken by the driver to place the packet on the physical media after its transmit function has been invoked), and the propagation time. The propagation time includes forward and backward propagation and the filter time. The forward propagation is the time taken for the packet to reach its destination after it has been placed on the physical media. The backward propagation time includes the time taken by the passive end to filter out

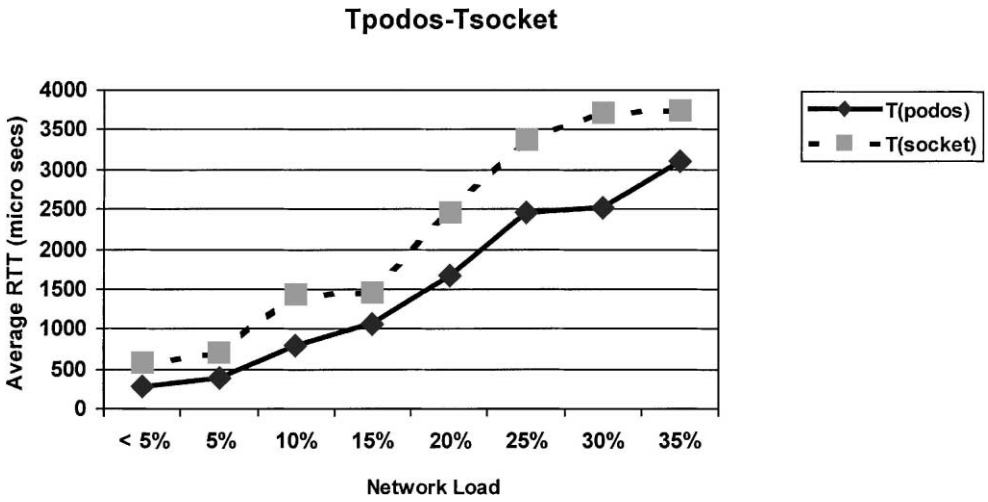

Fig. 3. Comparison between PODOS RTT and RTT using regular sockets.

the PODOS packet and reply. The filter time is the time taken by the filter, installed in the datalink layer, to extract PODOS packets. Of these, the propagation time is entirely dependent on the network load and the rest depend on the multiprogramming level of the system.

3.4.2. Experiment 2

In this experiment, we compare the RTT's of three protocols, the TCP/IP, the simple PODOS protocol, and PODOS with Transmission-Groups. The PODOS with Transmission-Groups is a variant of the PODOS protocol that transmits PODOS packets across multiple interfaces. In this case, we vary the network load only on the primary interface and maintain minimal load on the other two interfaces. The Transmission-Groups-based communication is a simple divide-and-conquer strategy:

$$T_{RTT} = T_{PODOS} + T_{tgcomm}. \quad (2)$$

It divides the tasks in hand equally among interfaces thereby sharing load and enhancing system throughput.

Fig. 4 is a graphical representation of the results of this experiment. The T(tg_minload) is the average RTT when transmitting packets across multiple interfaces, with varied load on the primary interface

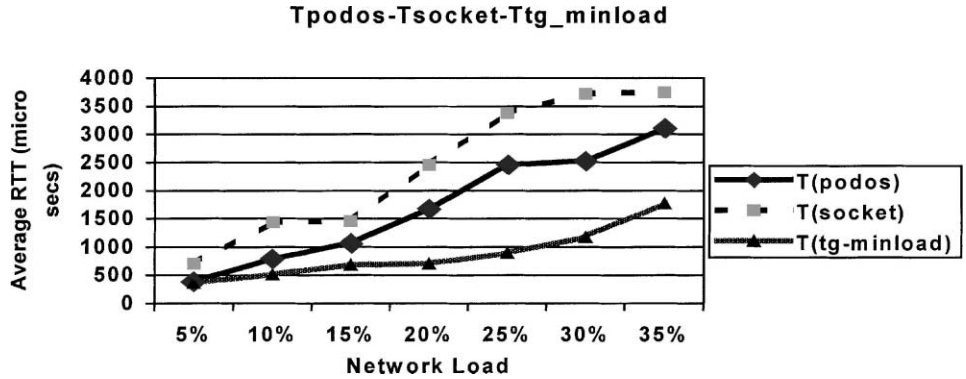

Fig. 4. Comparison between PODOS RTT and RTT using regular sockets with minimal load on secondary interfaces.

and minimal load on the other two interfaces. In Fig. 4, we can observe the drastic performance gain with Transmission-Groups. We can see how the Transmission-Groups-based PODOS protocol performs consistently at higher loads. It is almost three times faster than the regular networking protocol. Transmitting PODOS packets across multiple interfaces reduces the load on a particular interface and thus decreases the RTT of packets.

The RTT is given in Eq. (2). The RTT is the time taken to transmit a regular PODOS packet plus the time taken by the Transmission-Group algorithm. The Transmission-Group algorithm usually takes around 100–150 μs, which is very less overhead when compared to the performance gain [7].

3.4.3. Experiment 3

This experiment is similar to the previous one except in that, we compare only the PODOS variants, namely the simple PODOS, the PODOS with Transmission-Groups (minimum load on secondary and tertiary interfaces), and the PODOS with Transmission-Groups (approximately same load on all interfaces). In this case, we vary the network load on all interfaces.

We can observe the stability of the T(sameload) protocol at high loads, 20, 25, and 30%. The RTT varies by very meager amounts. Fig. 5 depicts these results. From Fig. 5, it is evident that even when the load on all interfaces is approximately the same, it is profitable to distribute packets among interfaces rather than transmitting them on a single highly loaded interface.

Let us consider the 30% load case. The primary interface was already 30% loaded and transmitting another 1000 packets resulted in an RTT of 2524 μs. This loads further a highly loaded interface. Whereas dividing the 1000 packets among three interfaces and transmitting 333 packets on each interface resulted in an RTT of 1239 μs which is less than half the RTT of a simple PODOS. Thus Transmission-Groups is beneficial even in a heavily loaded case.

From Figs. 3–5 we can derive the relation, depicted by Eq. (3), that holds under all network loads. The relation that is of more interest to us is the Transmission-Groups at approximately same load and the socket. From the above graphs, it is obvious that the RTT of Transmission-Groups would be much better than that of a socket read/write [7].

$$T_{\text{tgcomm-minload}} < T_{\text{tgcomm-sameload}} < T_{\text{PODOS}} < T_{\text{socket}}. \quad (3)$$

3.4.4. Experiment 4

This experiment is to measure the connection establishment time of the PODOS protocol. We further compare this to traditional socket connection establishment in TCP/IP. Connection establishment in PODOS implies making a CDT entry at the active end, transmitting the packet, making a CDT entry at the passive end and responding back to the active end. This signifies a virtual circuit. We further time the read and write of the first 1500 byte packet through this circuit. In traditional socket connection, this involves the following system calls: socket( ), bind( ) and

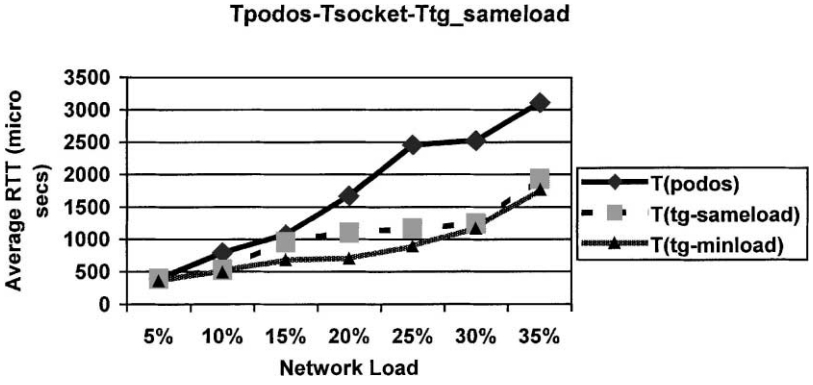

Fig. 5. Comparison between PODOS RTT and RTT using regular sockets with same load on all interfaces.

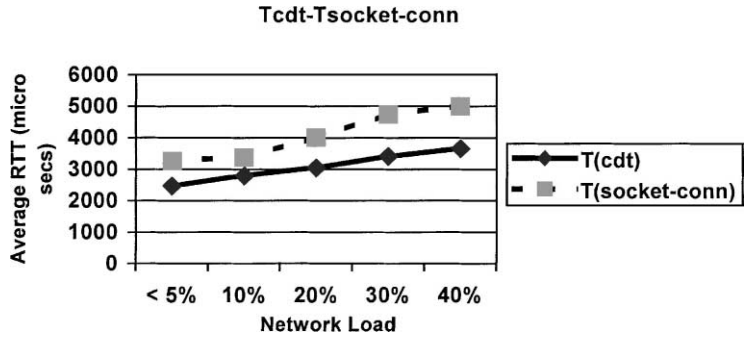

Fig. 6. Comparison between PODOS connection establishment time and socket connection establishment time.

connect( ). From Fig. 6, we can observe that T(cdt) is very stable and consistent at network loads.

The connection establishment in PODOS includes the server setup time too, whereas in typical socket connections, the server is assumed to be listening for a connection and thus server setup time is ignored. And server setup time typically takes around 500–800 μs. Further, the results above depict connection establishment without Transmission-Groups. Thus, in reality, the performance gain is much more.

The RTT is given in Eq. (4).

$$T_{RTT} = T_{PODOS} + T_{CDT}. \quad (4)$$

A typical CDT entry creation, under an optimal multiprogramming level takes around 200 μs, which is lesser than standard connection setup time.

4. The PDFS

In this section, we will discuss the implementation of the PDFS [10]. We will consider the following:

- distributed file systems;
- motivation for PDFS;
- design and overall structure;
- architecture and implementation;
- trace of a remote file fetch;
- performance.

4.1. 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 [11] and Coda [12].

NFS is a network file system by Sun Microsystems, designed for a network of computers. NFS works in both LAN and wide area network (WAN) environments. NFS uses a standard networking protocol called Remote Procedure Calls (RPCs) [8], which is built on sockets and UDP [8]. 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 [11].

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 [12].

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

4.2. Motivation for PDFS

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 are based upon the traditional TCP and UDP protocols that were primarily designed for WANs. For example, the NFS uses a networking standard called RPC, which is built upon UDP (TCP versions are available too). RPC [13] 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 [11], which would make it reliable but slower. We basically 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 LAN. This led to the evolution of the PDFS.

Let us look at the design and implementation of the PDFS. In this section, we will discuss the PDFS, its architecture and implementation.

4.3. Overall structure

Each node in the PODOS cluster is named as *linus1*, *linus2*, *linus*⟨*n*⟩. Every node has its own unique files 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 PDFS would be used and would behave [10]. They are the following:

- naming scheme;
- assumed mounts;
- lazy update semantics.

4.3.1. Naming scheme

Every distributed file system has to have a naming convention that would help to 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 [10].

4.3.2. 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.) [11].

When an access is made to a file, "/*linus4*/*miaow*", the PDFS, 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 PDFS. If RM returned a positive result, then the PDFS proceeds to fetch the file. Thus, the PDFS 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.

4.3.3. Lazy update semantics

Typical distributed systems follow one of the following file sharing semantics [12]:

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

Following the Unix semantics is very costly in a distributed scenario. 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 1 K) [10].

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

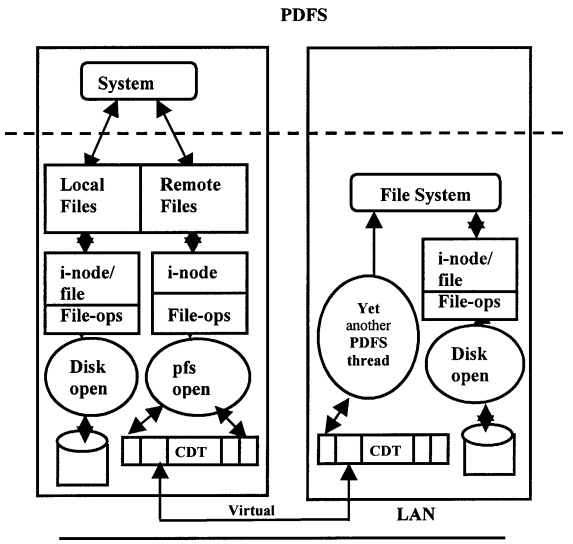

Fig. 7. PDFS architecture with kernel threads.

4.4. Architecture

In this section, we will discuss the following [10]:

- mapping i-node operations;
- kernel resident state-full threads;
- read ahead caching.

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

4.4.1. Mapping i-node operations

The PDFS is tightly integrated with the Linux file system and exploits its architecture for performance benefits (Fig. 7). PDFS uses the same file access primitives for both local and remote file operations. The PDFS exploits the Linux operating system's virtual file system (VFS) [5] 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 these 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 [5].

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 PDFS 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 PDFS uses the VFS architecture to channel remote file operation through traditional i-node structures. Thus, the advantages to this approach are the folowing:

1. speed;
2. tight Integration with traditional file system;
3. same primitives for local and remote accesses.

We will discuss more about the functionality of the above-mentioned methods in Section 4.5. Let us look at kernel threads [10].

4.4.2. State-full kernel threads

A key design decision of the PDFS is the use of kernel resident state-full threads. These threads are specialized kernel-level worker-bees of the PDFS, 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 initiates the PDFS. The PDFS spawns a thread (called "*Yet Another PDFS*") 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 PDFS 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 [10].

4.4.3. Read-ahead caching

The PDFS employs read-ahead caching to further improve the performance. It is traditional in distributed file systems to perform read-aheads. The PDFS fetches data from the remote system in terms of 1 K blocks and buffers it in the CDT. As long as the length of the data requested is less than the length of the data in the CDT, the PDFS avoids making a remote fetch. The moment the length requested is greater than the length of the buffered data in the CDT, a 1 K 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.

4.5. 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 PDFS and their interaction with the CM and the RM. The protocol used by PDFS for peer-to-peer communication (PDFS–PDFS) is as follows [10]:

- 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 PDFS 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) (Table 5) 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 PDFS 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 [5]:

```
struct file {
        ......................
        struct file *f_next, *f_prev;
        struct inode *f_inode;
        struct file_operations *f_op;
        void *private_data; /* needed for tty
driver, and maybe others */
};
```

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

- The PDFS 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.

- Passive end's response to open
  - At the passive end, the CM makes a CDT entry, establishes a virtual circuit and then invokes the PDFS.
  - The PDFS 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 GPID (address of the node + local PID) (Fig. 10) along with the message (also returns the first block of data if it is a read operation) and suspends itself.
- 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 PDFS. 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.
  - Writes are typically stored in the CDT and flushed to the remote end after the CDT buffer reaches 1 K. In the case where the requested length is greater than the CDT buffer length, the PDFS builds a packet (requesting for a 1 K block), registers it with the CDT, and invokes the CM for transmission and blocks for the arrival of the data.
- 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.
- 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 PDFS.

4.6. Performance

In this section, we present the preliminary performance results obtained by analyzing the PDFS against NFS, network file system. All experiments were conducted with the following setup [10]:

1. Using Pentium 133 MHZ machines, with 32 MB of RAM, interconnected with Ethernet [9] adapter cards (16 and 8 bit cards).
2. With default NFS block size of 1 K.
3. In the case of NFS, the time taken to fetch the file, the first time is considered.
4. All time values presented are averaged milliseconds.

Let us look at the experiments conducted [10].

4.6.1. Experiment 1

In this experiment, we compare the average time taken by NFS and PDFS to fetch files of different sizes. We considered standard file sizes of 10 K, 50 K, l00 K, 500 K, and 1 M. The graph depicts the results. The graph (Fig. 8) uses a logarithmic scale for the *y*-axis, to clearly show the performance gain achieved for all file sizes. The PDFS consistently performs better than NFS for all file sizes. Usually, in file systems, most files accessed are less than or equal to 10 K [14]. PDFS has a considerable performance gain of around 30–40 ms 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

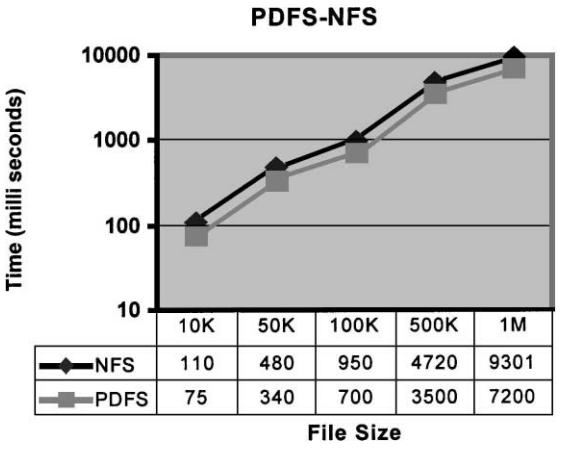

Fig. 8. Comparison between file fetch times in PDFS and NFS.

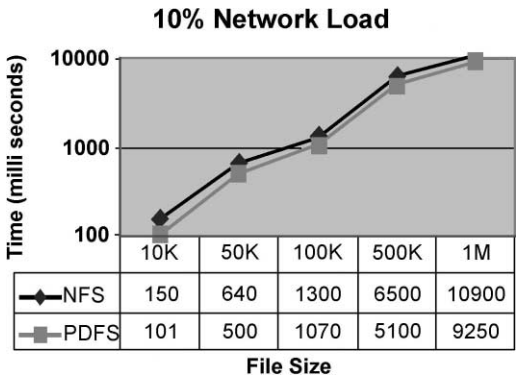

Fig. 9. Comparison between file fetch times in PDFS and NFS at 10% network load.

propagation time and the thread processing time. The dispatch time is the 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 200–300 μs. 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 a usually around 400 μs. 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.

4.6.2. Experiment 2

The next couple of experiments were designed to test the effective utilization of the high-speed communication bed, by the PDFS. In the previous experiment all file fetches were made on the same network interface. In this experiment, the PDFS makes use of the Transmission-Group feature of the CM, wherein virtual circuits are multiplexed across multiple network interfaces [7]. 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 interface (immaterial of the load). The result under 10% network load is depicted in Fig. 9. Since the CM distributes virtual circuits uniformly using a round-robin mechanism, the overall PDFS performance is improved when compared against NFS. At all loads, the PDFS consistently has a 30–50 ms gain over NFS for file sizes around 10 K, a 100–200 ms gain for file sizes around 50 K, a couple of hundred microseconds gain for file sizes around 100 K. When clustering activities tend to be intense these are considerable performance gains, which would improve overall system throughput.

Listed below (Table 4) are the average times, in milliseconds, for the system calls: open, read and close. These were computed by studying the above results [10].

5. Global IPC

GIPC is the basis for remote program execution in PODOS. This section discusses the following:

- GPID;
- design;
- implementation;

Table 4
PDFS seek times

	NFS	PDFS
Open	2.4	1.5
Close	0.052	0.03
Read	10	6

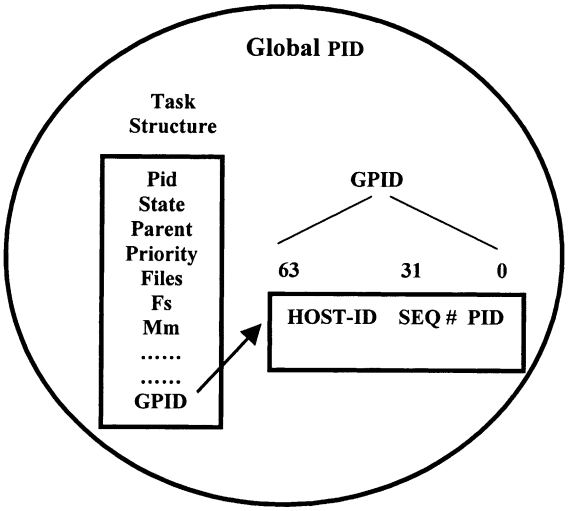

Fig. 10. GPID structure.

- remote process execution;
- performance.

5.1. Global process identifiers

A GPID is a unique identifier for a process in the entire cluster. When a process from one host communicates with a process on another host or a group of processes from different hosts in the cluster, there should exist a mechanism to uniquely identify the process among group of processes. The PID of a process is unique only in its local host. Thus, we need a GPID (Fig. 10).

GPID in PODOS is 64 bits (8 bytes) long, of which the least significant 2 bytes are for the local PID; the next 2 bytes are for a sequence number; first 4 bytes are for the host ID. The GPID of a process is also part of its Task_Structure in Linux kernel, thus whenever a process is created in PODOS it receives a GPID. The sequence number is a global number that will be maintained by the kernel and incremented every time the kernel boots up. This avoids the sequence number mismatch and renders GPID unique in spite of system crashes, or reboots.

5.2. Design

In this section, we will look at some of the key design issues involved in the GIPC component.

5.2.1. A file system interface

A primary concern while designing a distributed system is to limit the number of services specific to remote requests. As mentioned before, no new system call or library functions are added in order to provide an interface to PODOS services. This has been the principle behind the design of GIPC.

The GIPC component, similar to PDFS, uses file manipulation functions as its interface, although differing in its signature and the semantics of arguments passed. The system call used to establish a GIPC connection between the active end and passive end service thread is the OPEN call. We will discuss further about these in the implementation section. Below are a few reasons for using a file system interface for GIPC.

- *Preserve distributed system standards*. In traditional distributed systems, inter-process communication is achieved using data structures called ports. When user requests a new port, a descriptor to the port structure is returned. PODOS follows a similar strategy, although replacing ports with CDT entries. The index to the CDT entry is abstracted within the file descriptor returned by OPEN system-call.
- *Exploit i-node structure*. GIPC uses the i-node structure of Linux, similar to PDFS. We will discuss the subtle differences in the implementation section.

5.3. Implementation

As mentioned above, the main goal of GIPC is to provide user programs with the capability of being able to execute processes on remote hosts. At the same time the entire mechanism is made transparent to user.

Let us look at the implementation issues involved in achieving this.

5.3.1. Mapping i-node operations

The GIPC implementation is similar to that of PDFS, except that the protocols are slightly different. Both GIPC and PDFS use the OPEN system call to access the kernel space, to establish a virtual circuit, and also to start the service thread at the passive end. The OPEN system call accepts three arguments, namely: file name, mode and flags. The difference in the two protocols lies in the semantics of the signature.

For a GIPC service the arguments of the OPEN call map to a remote hostname (as a filename), and a

constant O_GIPC or 4 (as the mode). The flag argument is used for group communication and differentiating between local and remote files at the passive end. For example, invoking *OPEN*("/*linus4*" *O_GIPC*) from the host "linus3" will open a virtual circuit from "linus3" to the host "linus4". If the mode is O_GIPC, then the open call sets the "*open* function pointer" in file_operations field of i-node to *gipc_open* and invokes it. In function gipc_open the file_operations field is replaced by the *gipcfop*, enabling further read and write calls in the user program to invoke *gipc_read* and *gipc_write* functions. The definition of gipcfop is as follows:

struct file_operations gipcfop = {*NULL*, &*gipc_read*, &*gipc_write*, *NULL*, *NULL*, *NULL*, *NULL*, *NULL*, *NULL*, *NULL*, *NULL*, *NULL*, *NULL*};

This facilitates faster access to functions that perform remote actions and also provides a clean interface to the user.

5.3.2. Kernel threads

Similar to PDFS, GIPC uses kernel threads at the passive end to handle remote requests. The functionality of the thread in this case is slightly different, in that it *execs* the program specified in the gipc_write operation. This process of using kernel threads expedites the remote execution and increases the performance of PODOS.

5.4. Remote process execution

In this section, we will discuss the following:

- tracing a GIPC request;
- local and remote programs;
- input, error, and results.

5.4.1. Tracing a GIPC request

The protocol for GIPC is as follows:

1. The active end initiates the service using OPEN system call passing the remote host name as argument.
2. The remote request for exec service goes through the kernel and reaches GIPC, which will allocate a CDT entry and dispatch the message to the remote host using CM. The OPEN system-call returns after sending the message and before a reply from its remote host.
3. The remote GIPC receives the service request and it invokes the service thread that will send an acknowledgement back to active end, sending its CDT index and GPID through the message.
4. The active end CM then receives a reply packet containing the GPID of the service thread and the CDT index of the passive end, thereby establishing a virtual circuit.
5. The active end process sends the name of the program to be executed, the program arguments and also the program environment using the WRITE system call.
6. Again, this request reaches the GIPC, which checks whether an acknowledgement for the OPEN was received. If not, it blocks the WRITE system call until an acknowledgement arrives or until a timeout value. If the virtual circuit has already been established, it dispatches the write request to remote GIPC.
7. On write request the remote service thread, which was initiated by the open call, sets up an environment for the user program and execs it.
8. Once the program terminates, the passive end dispatches the results and the active end closes the virtual circuit.

5.4.2. Local and remote programs

A detail that needs to be mentioned is the location of the program to be executed. There exist two possibilities: the program could already reside in the remote machine, in which case a simple exec would suffice. The more complex scenario is when the program has to be dispatched to the remote end. These two cases are handled by incorporating a flag in the OPEN call that would differentiate them.

The following piece of code illustrates the first case where the "a.out" program resides in the remote end. The OPEN call needs no special flag; just the regular O_GIPC would suffice.

$f = open(\text{"}/linus4\text{"}, O\_GIPC);$
$n = write(f, \text{"}a.out\text{"}, 7);$
read(f, buff, 80);

The results are obtained by performing a read on the file descriptor that maps to the virtual circuit between the two ends.

The other case is when the program to be executed has to be fetched from the active end.

$f = open(\text{“}/linus4\text{”}, O\_GIPC, REMOTE);$
$n = write(f, \text{“}/linus3/a.out\text{”}, 14);$

This flag indicates to the passive end that the file specified by the subsequent WRITE call has to be fetched from a remote machine. The machine on which it resides precedes the file name. Thus, in our case, before executing the program it is fetched from “linus3” by the PDFS component at the passive end. This would initiate a new virtual circuit for the file fetch and follows the protocol explained in PDFS.

The code fragments shown above would eventually become part of a scheduler that automatically dispatches jobs based on the load on a particular machine.

5.4.3. Input, error, and results

In order to obtain the output of remotely executed programs, applications in PODOS are linked to a special library that handles remote I/O. These libraries are preinstalled in all PODOS machines. The library functions hide the remote I/O handling mechanism from the application.

We saw earlier that the CDT index of the virtual circuit is passed as an environment variable to the program. Library functions access the environment to obtain the index and thus write and read from the CDT buffers. This way, results are communicated back to the active end.

5.5. Performance

The preliminary performance comparison results against “rsh” [5] and “REXEC” [15], from Berkeley, are quite promising, providing at least 30–40 ms performance gain. Launching a null program and obtaining results with GIPC typically costs around l30 ms, whereas REXEC results in 160 ms.

6. Resource manager

The resource manager maintains the system state information of the PODOS cluster. The RM component in every node has a SST that maintains information about every other node in the cluster.

Table 5
System state table

Struct system_state {	
Char hostname[20];	The name of the node
Struct interface_stats if_stats[3];	Hardware addresses
Int nr_running;	Number of processes running
Int nr_tasks;	Number of processes
Float load_avg[4];	Load average
Int idle_time;	Idle time
Unsigned long mem_free;	Total memory free
}	

The information exchange is achieved using a periodic update mechanism, wherein a timer in the RM expires after certain duration invoking an update. A fragment of the information exchanged is given in Table 5. The RM in a particular node builds a PODOS packet of the information and uses the CM to transmit it. While having to receive information about other nodes in the cluster, the CM filters it out as a packet that is destined for the RM. The RM packets can be classified under the ephemeral category of CDT entries [7]. A key attribute in RM is the update duration, which can only be decided based on the system performance. Current duration is around 5 s.

We have seen in the previous sections as to how this information is used by PDFS and GIPC to check if a host is valid and alive. The information would be further used efficiently by automatic scheduling strategies for load sharing.

7. Conclusions

In this paper, we have dealt with the design and implementation of PODOS. We have described the architecture of various components of PODOS. These include: a custom communication protocol that employs a round-robin Transmission-Groups mechanism to multiplex packets across multiple network interfaces and also provides a CDT as an interface for other PODOS layers such as PDFS, RM, and GIPC; a distributed file system, PDFS that builds an efficient file-sharing environment on top of the high-speed communication subsystem; a GIPC mechanism that allows remote program execution.

References

[1] A. Goscinski, Distributed Operating Systems: The Logical Design, Addison-Wesley, Reading, MA, 1991.
[2] M.J. Litzkow, M. Livny, M.W. Mutka, Condor — a hunter of idle workstations, in: Proceedings of the Eighth International Conference on Distributed Computing Systems, June 1988, pp. 104–111.
[3] P. Merkey, Beowulf Project at CESDIS, 1994. http://beowulf.gsfc.nasa.gov.
[4] A.S. Tanenbaum, Distributed Operating Systems, Prentice-Hall, Englewood Cliffs, NJ, 1995.
[5] The Linux Documentation Project, 1998. http://sunsite.unc.edu/LDP.
[6] P.T. Maginnis, Design considerations for the transformation of MINIX into a distributed operating system, in: Proceedings of the 1988 ACM 16th Annual Computer Science Conference, 1988, pp. 608–615.
[7] S. Vazhkudai, P.T. Maginnis, A high performance communication subsystem for PODOS, in: Proceedings of the First IEEE International Workshop on Cluster Computing, December 1999, pp. 81–91.
[8] A.S. Tanenbaum, Network protocols, ACM Comput. Surv. 13 (4) (1981) 453–489.
[9] D.R. Boggs, J.C. Mogul, C.A. Kent, Measured capacity of an Ethernet: myths and reality, in: Proceedings of ACM SlGCOMM'88 Symposium, Vol. 18, No. 4, August 1988, pp. 222–234.
[10] S. Vazhkudai, P.T. Maginnis, The PODOS file system — exploiting the high-speed communication subsystem, in: Proceedings of the IEEE International Workshop on Cluster Computing — Technologies, Environments, and Applications (CC-TEA'2000) (PDPTA2000), Las Vegas, Nevada, June 2000.
[11] S.M. Inc., NFS: Network File system Protocol Specification, Vol. RFC 1094, SRI Network Information Center, March 1989.
[12] P. Braam, The Venus Kernel Interface, March 1998. http://www.coda.cs.cmu.edu.
[13] A.D. Birrell, B.J. Nelson, Implementing remote procedure calls, ACM Trans. Comput. Syst. 2 (1984) 39–59.
[14] M. Satyanarayanan, A study of file sizes and functional lifetimes, in: Proceedings of the Eighth Symposium On Operating Systems Principles, ACM, 1984, pp. 96–108.
[15] B.N. Chun, D.E. Culler, REXEC: A decentralized, secure remote execution environment for clusters, in: Proceedings of the Fourth Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, Toulouse, France, January 2000.

Sudharshan Vazhkudai is a doctoral candidate in the Computer and Information Sciences Department at the University of Mississippi. He received his BE and MS in Computer Science from Karnataka University, India, and The University of Mississippi in 1996 and 1998, respectively. His research interests are primarily concerned with the design and implementation of resource management strategies for distributed systems.

Jeelani Syed received his BE and MS in Computer Science from Andhra University, India, and The University of Mississippi in 1997 and 1999, respectively. His research interests are in network routing and distributed systems. He is currently working for Unisphere Solutions Inc., working on network routing related issues.

Tobin Maginnis is an Associate Professor in the Computer and Information Sciences Department at the University of Mississippi. His research interests are in distributed operating systems, networks, mobile IP and multimedia. He is also the President and founder of Sair Inc., a leading Linux certification company.