The Development of an Optimized PATHWORKS Transport Interface

By Philip J. Wells

1 Abstract

Digital's Personal Computing Systems Group developed an optimized transport interface to improve the performance of the PATHWORKS for VMS version 4.0 server. The development process involved selecting a transport protocol, designing appropriate interface test scenarios, and measuring server performance for each transport interface model. The engineering team then implemented the optimized design in the server and performed benchmark testing for specified server workloads. Using an optimized transport interface improved server performance by decreasing the time required to complete the test while maintaining or decreasing the percent CPU utilization.

2 Introduction

The PATHWORKS family of network integration software products includes file servers that provide file and print services to personal computers (PC) in local area networks (LANs). Developed by the Personal Computing Systems Group (PCSG), the PATHWORKS for VMS version 4.0 server supports the Microsoft LAN Manager network operating system. This server allows PC clients transparent access to remote VMS files. With each new release of the PATHWORKS for VMS product, the PCSG engineering team improved server performance and thus accommodated an increasing number of time-critical PC applications. In version 2.0, we introduced disk services as an alternative to file services for read-only files. We included data caching in version 3.0 of our file server.

For version 4.0, our goal was to increase file server performance by optimizing the transport interface and the data buffering algorithm. To achieve this goal, we evaluated several transport interface designs and measured server performance for various server workloads. We started with the premise that using the standard buffered interface results in increased overhead for each transaction and thus decreases overall CPU availability. Figure 1 illustrates this interface design. The server copies a user data buffer in process context across the kernel service interface to a system buffer in system context, before transferring the data to the network layer.

Prior analysis of PATHWORKS server performance over the DECnet transport protocol revealed that when the file server request sizes were large, i.e., 4 to 8 kilobytes (KB), file server performance met or exceeded the performance of other vendors' transports. However, when the transfer sizes were small, i.e., less than 256 bytes, file server performance degraded significantly. Also with small request sizes, our server did not ramp well

when many clients were supported in this environment. As illustrated in Figure 2, incremental increases in server workload cause dramatic increases

in CPU utilization once a certain workload is reached, i.e., at the knees of the curves, denoted by points A and B. We wanted our server performance to approach that represented by the curve containing point B. In this way, we could support more clients at the same or less CPU utilization.

3 Server Performance Analysis

We based our analysis of PATHWORKS server performance on two initial hypotheses:

- The CPU overhead associated with a buffered interface significantly degrades server performance.
- The variable transaction response times inherent in using the standard queued I/O (QIO) interface results in inefficient server performance.

Protocol Selection

To begin our performance analysis, we needed to choose a transport protocol. We considered the DECnet and the local area system transport (LAST) protocols and selected the LAST protocol for the following reasons:

- An advanced development effort on the DOS client software showed that file and print services over the LAST protocol decrease the client memory usage by one-third.
- The PATHWORKS engineering team maintains the LAST protocol and thus, can make any required modifications.
- The VMS operating system implementation of the LAST transport protocol is called LASTDRIVER. LASTDRIVER serves our purpose because it presents a buffering model that permits the passing of multiple noncontiguous data buffers as a single, logically contiguous buffer. Figure 3 shows two physical data buffers, of sizes N and M, being passed to LASTDRIVER as a single message. The second buffer descriptor contains a zero in the next buffer descriptor pointer word. This value indicates the end of the data stream.

Test Scenarios

After selecting the LAST transport protocol, we created four test scenarios to measure server performance. The first scenario, the kernel model, required developing a VMS device driver that was layered on top of LASTDRIVER. In this model, when the driver receives request data, the data is immediately transmitted back to the client. The driver does not copy buffers and does not schedule a process. This model represents the optimum in performance, because absolutely no work is performed in relation to the

request.

The second test scenario required that we develop a user-mode test program. This model performs similarly to the kernel model in that it loops receive data directly back to the client without performing any copy operations. This model differs from the first model in that the driver schedules a VMS process to loop the data back to the client. We then developed the following variations on this test scenario to accommodate three transport interfaces to the VMS process:

- A standard VMS QIO interface model. This model uses the standard interface provided with the VMS operating system.

  The remaining two scenarios represent optimized transport interfaces with regards to two aspects of a request: the initialization and the completion.

- A model that incorporates the standard VMS QIO interface with a process wake-up completion notification. This QIO/WAKE model uses the standard QIO interface to initiate a transport request. However, the transport queues I/O completion notification directly to the receiving process by means of a shared queue and a process wake-up request. The purpose of this optimization was to avoid the standard postprocessing routines of the VMS operating system.

- A model that includes kernel mode initialization and wake-up completion notification. This CMKRNL/WAKE model uses the transport completion technique of the previously described model. However, we created an entry point into the driver for the test program to call, thereby initiating transport requests. The test program uses the change-mode-to-kernel (CMKRNL) system service to call the driver entry point. This optimization was made to avoid the standard QIO interfaces.

To support the optimized transport interfaces, the test program allocates a buffer in process context and divides it into two sections: the first contains shared queues for moving data between process context and system context; the second contains the test program's shared data buffers. The driver issues a system call to double map the shared buffer into system context. Figure 4 shows this double-mapped buffer. Since the buffer is contiguous, the difference between the start of the shared data region in process context and the start of the shared region in system context is a constant, and is used as an offset. The test program accesses the shared region by using a process virtual address (PVA); device drivers access the region by adding the offset to the PVA to compute a system virtual address (SVA), as shown in Figure 5. To accomplish completion notification, the driver inserts the data into the shared queue and issues a process wake-up request for the test program.

Performance Measurements

Our hardware platform was a VAXstation 3100 workstation. Wee measured server performance as the difference between the request arrival time and the response departure time, as observed on the Ethernet. Times were measured in milliseconds using a Network General Sniffer. Table 1 presents the test results.

Table 1: Server Performance over Various Interfaces

Interface	Server Performance (Milliseconds)
Kernel Model	0.8
Standard VMS QIO Model	2.2
QIO/WAKE Model	1.7
CMKRNL/WAKE Model	1.6

As Table 1 shows, we decreased server response time by using an optimized transport interface. The kernel model yields the best possible performance results. As we move from the standard VMS QIO interface to more optimized interfaces, there is a decrease in transaction response time which represents improved server performance.

Data collected during initial performance testing supported our decision to optimize the transport interface. Occasionally while testing the interfaces, server throughput dropped dramatically, i.e., 30 to 50 percent, for a short time interval, i.e., one to two seconds, and then resumed at its prior rate. Initially, we thought there was a problem with our code. However, the anomaly persisted throughout the development period, so we decided to investigate the cause of the dip in performance.

The VAXstation 3100 system that we used to perform the testing had a graphics controller card installed, but did not include the graphics monitor. Since the system included a graphics card, the DECwindows login process frequently tried to display the initial DECwindows login screen. This attempt failed because there was no monitor. Therefore, the process was deleted and restarted a few minutes later. We concluded that the temporary drop in server performance we had observed was the effect of the DECwindows start-up process.

The significance of this observation became apparent when we optimized the

transport interface, and the effect of this background process activity decreased to less than 10 percent. We concluded that the optimized interface was less susceptible to concurrent I/O than was the standard QIO interface.

4 Implementation

Once the initial testing of prototypes was complete, we decided to implement the double-mapped buffering algorithm with shared queues. The VAX architecture provides inherent queuing instructions that allow the sharing of data across dissimilar address spaces. It accomplishes this by storing the offset to the data, rather than the address of the data, in the queue header. This technique permits us to insert a system virtual address into a queue in system context and later remove the address in process context as a process virtual address. A second function that these instructions perform is to interlock the queue structure while modifying it. This procedure precludes concurrent access by other code and thus allows the interface to support symmetrical multiprocessing.

We modified the file server to support this new optimized transport interface. To ease the implementation, the QIO interface emulates the DECnet interface in all aspects except one. Since the client-server model is essentially a request/response model, we developed a transmit /receive (transceive) operation that allows the server to issue read buffer and write buffer requests at the same time. This variation reduces the number of system boundary crossings. When the server transmits buffers, these buffers return to the server process by way of a transmit complete queue. When the server receives a new request message, the associated buffer is transferred to the server process via a receive complete queue. To facilitate a transceive operation, we defined a work element data structure. As shown in Figure 6, a work element permits the passing of two distinct data streams: one for transmit and one for receive.

As development of the client and server software modules continued, we encountered some interesting problems. The following three sections describe several of these problems and how we addressed them.

Microsoft LAN Manager Redirector I/O Behavior

When the Microsoft LAN Manager redirector, i.e., the DOS client protocol equivalent of the VMS file server, generates a read request, it first writes the request for service to the network. The redirector then issues a read request and uses a short buffer to receive only the protocol header of the response message. After verifying that the response was successful, the redirector issues a second read request to receive the data associated with the response message.

This behavior requires lower protocol layers to buffer the response data until the redirector issues a read request to receive the data. In order to buffer the response data for the client, the transport layer needs to allocate an 8KB buffer. An alternative approach to maintaining a dedicated transport buffer is to use the inherent buffering capacity of the Ethernet data link software and the Ethernet controller card, which maintain a cache of receive buffers. This technique requires the transport layer to retain data link receive buffers while the redirector verifies the response message protocol header and posts the actual receive buffer. Once the redirector issues the second read request, the remaining data is copied and the Ethernet buffers are released.

One problem with this approach is that each vendor's Ethernet card has different buffering capacities. In some cases, the capacity is less than the size of the maximum read request. To support such inadequate buffering capability, we inserted a buffer management protocol (BMP) layer between the file server and the redirector. The resulting process is as follows:

The client module communicates its data link buffering capacity to the server module in the session connect message. When the application generates data requests, the DOS redirector packages a server message block (SMB) protocol message and passes it to the BMP layer. This layer adds a small buffer management header to the message and pass it to the transport layer to transmit to the server.

To complete the operation, the file server processes the request, formats an SMB response message, and passes it to the BMP layer. At this interface, the size of the response message is indicated by the transmit buffer descriptors, and a protocol header that describes the response packet is created. If the response message is larger than the client's data link buffering capacity, the driver software segments the response packet into smaller messages and passes these messages to the server transport

to transmit to the client. The client module copies the header to the redirector's short buffer and completes the redirector's read request. The BMP layer then waits for the second read to copy the remaining data to the redirector's buffer and releases the data link buffers. At this point, the client can request more data from the server.

Response Buffering

The LAST protocol does not acknowledge the receipt of messages because it relies on the integrity of the underlying LAN to deliver datagrams without error. Consequently, the BMP layer must buffer all response data transmitted to the client to protect against packets that are lost or discarded. In such a case, the BMP layer transmits the original response message back to the client without sending the message to the server process.

For instance, consider the two cases shown in Figures 7 and 8. In Figure 7, a client generates a read request at time T1. The server processes the request and generates a response at time T2. The response is lost due to congestion, so the client requests the same data again, as indicated at time T3. The server rereads the file and generates a new response. Since the read operation is naturally idempotent, i.e., it can be repeated without changing the result, the operation completes successfully.

In the case depicted in Figure 8, we changed the operation from a disk read to a delete file. Here, the client makes the delete request at time T1, and the server successfully deletes the file at time T2. The response message is again lost. When the client reissues the delete file request at time T3, the server fails in its attempt to perform the operation because the file no longer exists. The delete operation is not idempotent; thus, repeating the operation yields a different outcome.

We cannot determine in advance the actual idempotency of any given request. Therefore, the BMP layer must cache all response buffers. If a response message is lost, the server transmits the original response message instead of retrying the entire operation. If, as in the second example, the server is able, at time T4, to transmit the actual buffer used at time T2 to store the response message, the operation can complete successfully.

To facilitate the buffering of response data, the transport provides a transaction identifier for request and response messages. This identifier is set by the client BMP layer whenever a new request is received from the redirector. The server stores this identifier and verifies it against the identifier of the next request. If a received request has a duplicate identifier, the request must be a retransmission and the server transmits the message in the cached response buffer. If the identifier is unique, the cached buffer is returned to the server by means of the shared queues, and a new request is created. The client's single-threaded nature ensures that the transaction identifier method is successful in detecting a retransmission.

NetBIOS Emulation

The PATHWORKS transport interface implementation relies on the request /response behavior of the DOS redirector. However, the redirector uses the standard DOS network basic I/O system (NetBIOS) interface to communicate with transports, and this interface does not exhibit request/response behavior. Therefore, our implementation is not a true NetBIOS emulation and can prevent common NetBIOS applications from operating correctly.

To resolve this problem, we developed a common NetBios interface between the DECnet and LAST transports. After receiving a request, the client first tries to connect over the LAST transport. If the connection attempt fails, the request passes to the DECnet transport. Thus, standard NetBIOS application requests operate over the DECnet transport; only redirector requests are processed over the LAST transport.

5 Final Benchmarks

At the completion of the project, we performed benchmark tests to measure server performance for varied workloads and for a directory tree copy. Table 2 shows the results for varied workloads. The first column of the table describes the test performed. ALL I/O represents a raw disk I/O test in which the measured client issues read and write requests of various buffer sizes ranging from 128 bytes to 16KB. TP represents a transaction processing test that measures random read and write requests of small units of data. This test emulates a typical database application. The workload value indicates the number of client systems used in the test to produce a background workload. As one might expect, as the workloads increase, the performance of the measured client degrades.

Table 2: Final Benchmark Test Results for Varied Workloads

Test Description	LAST Protocol Elapsed Time (seconds)	LAST Protocol CPU Utilization (percent)	DECnet Protocol Elapsed Time (seconds)	DECnet Protocol CPU Utiliation (percent)
All I/O 0 Workloads	840	4	961	4
All I/O 2 Workloads	943	69	1074	75
All I/O 4 Workloads	1091	100	1434	100
TP 1 Workload	59	39	79	50
TP_5_Workloads	163	83	212	93

The entries in each row of the table are the elapsed time and percent CPU utilization for the given test. We measured server performance over the LAST protocol using our optimized interface and over the DECnet protocol using the standard VMS QIO interface. For the All I/O tests, the resultant elapsed time is the actual time it took to complete the test. For the TP tests, the performance numbers are the average of all the PCs tested. As Table 2 shows, we were able to decrease the elapsed time for each benchmark while maintaining the same or decreased CPU utilization.

The two graphs in Figures 9 and 10 illustrate these results. In the ALL I/O test, CPU utilization using the optimized interface increases steadily as the workload increases. Using the standard QIO interface, CPU utilization increases at a faster rate once a specified workload is reached. Although the TP graph in Figure 10 contains only two data points, it is evident that CPU utilization is proportionally higher for five workloads than it is for one. We performed multiple tests to verify that the results could be reproduced consistently.

The final benchmark test performed was a directory tree copy using the DOS XCOPY utility. In this test, the utility copies the directory tree first from the server to the client and then from the client to the server.

The bottleneck in this test is known to be the file creation time on the server. Therefore, we expected a more efficient transport interface to have no effect on server performance. The test results in Table 3 support our theory. The I/O rate and the elapsed time over both the DECnet protocol (using the standard transport interface) and the LAST protocol (using the optimized transport interface) are nearly the same.

Table 3: Final Benchmark Test Results for a Directory Tree Copy

Test Description	LAST Protocol Elapsed Time (seconds)	LAST Protocol I/O Rate (KB/sec)	DECnet Protocol Elapsed Time (seconds)	DECnet Protocol I/O Rate (KB/sec)
XCOPY to Client	115	39	15	39
XCOPY to Server	119	38	121	37

6 Acknowledgements

I wish to thank Jon Campbell for incorporating the interface design modifications into the file server, Alpo Kallio for developing the client software, and Alan Abrahams for designing the combined DECnet/LAST NetBIOS interface and for his encouragement and support.

7 Biography

Philip J. Wells Phil Wells is the PATHWORKS server architect and is responsible for coordinating the design and implementation of the PATHWORKS server products. In previous positions at Digital, Phil worked for Corporate Telecommunications designing Digital's internal network, the EASYNET, and helped support data centers and networks while in the Internal Software Services Group. Phil joined Digital in 1976 as a computer operator in the Corporate Data Center.

8 Trademarks