Performance Evaluation of an Off-Host Communications Architecture

Jeffrey R. Michel, Alexander S. Waterman, Alfred C. Weaver

Technical Report No. CS-93-32
June 01, 1993

Performance Evaluation of an Off-Host Communications Architecture

Jeffrey R. Michel, Alexander S. Waterman, and Alfred C. Weaver

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

The Computer Networks Laboratory at the University of Virginia has implemented the SAFENET lightweight protocol suite on the Navy's Desktop Tactical Computer (DTC-2). The software includes the Xpress Transfer Protocol (XTP), a new transport and network layer protocol to support high-throughput, low-latency, priority-sensitive communications. One of our research questions was whether to embed XTP in the DTC-2's UNIX kernel or run XTP on an attached processor on the machine's VMEbus. We implemented both strategies and the attached processor approach proved to have somewhat higher performance. Our conclusions identify both the advantages and performance concerns inherent in off-host protocol execution.

1 Introduction

Our challenge was to provide reliable end-to-end communications using XTP [STRA92] and an FDDI network to application programs running over the SunOS (UNIX) operating system. Two approaches seemed obvious: (1) embed XTP protocol processing in the DTC-2's operating system kernel, or (2) run XTP on an attached processor on the machine's system bus. The first idea was conventional; we chose the second because it offered several potential advantages:

- Reduced host load
- Predictable application processing
- Reduced and bounded host interrupt arrivals
- Dedicated processor cycles for protocol processing
- Specialized protocol processing hardware
- Ideal operating system environment in which to run the protocol

2 Implementation Architecture

This section describes the hardware and software implementation of our off-host communications architecture.

2.1 Hardware

Figure 1 depicts our hardware and software architecture. The host platform for our work is the C3 Desktop Tactical Computer (DTC-2), a SPARC-based machine with a VME backplane bus; the bus contains our FDDI adapter and protocol processor. The FDDI board is manufactured by Network Peripherals (NP), and the processor board is a Motorola MVME-167A (167) with 8 MB of on-board memory, all of which is addressable on the VMEbus. The FDDI MAC device driver resides on the 167, which communicates with the NP board through block-mode DMA, shared memory, and VME interrupts. The system is configured such that a portion of the 167 memory is mapped into the virtual memory of the DTC-2 kernel's 32-bit address space, facilitating communication through shared memory between the 167 and DTC-2. Communication also occurs through interrupts generated on the 167 board for the DTC-2. The 167 runs the pSOS+ lightweight multitasking operating system.

2.2 Software

Our software architecture exists to provide transport services to Ada applications running as UNIX processes. It consists of a set of Ada packages, a C library, a UNIX character device driver, an implementation of XTP 3.6, and an FDDI MAC driver. The individual software components are partitioned onto our two processor platforms and run in several address spaces. On the host, the Ada package bodies and C library run in the address space of their Ada application's process, while the character device driver runs in the kernel with its own distinct address space. On the attached processor, the XTP implementation and MAC driver run as a set of pSOS+ tasks sharing a common address space.

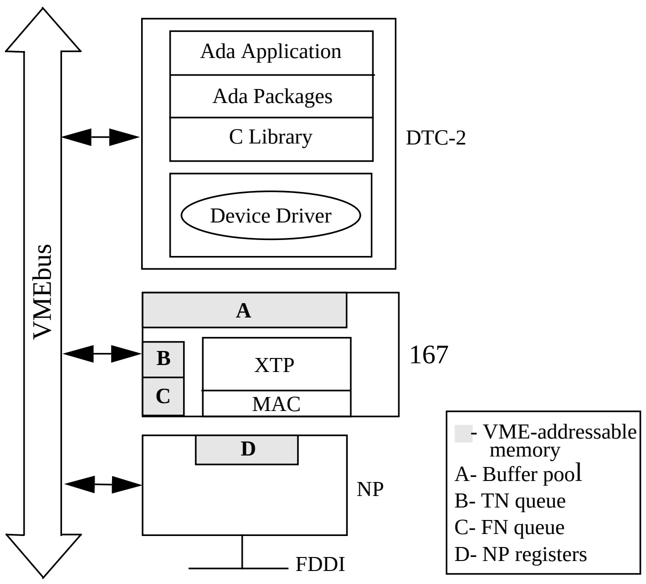

Figure 1: Communications System Architecture

An Ada application employs primitives such as `SEND_MESSAGE` and `GET_MESSAGE` from the Ada packages to perform network communication. The Ada program is linked with the library of C code which contains the user-level portion of the implementation of the transport services. The primary function of this C code is to issue control blocks to the protocol processor and receive its acknowledgments of command completion (acknowledgment blocks) through the use of the character device driver.

The C-library interface to the transport layer uses a standard UNIX character device interface. It utilizes system calls such as `read()` and `write()` to communicate with the off-host protocol processor. The communication mechanism between the host and the protocol processor is via two queues, the TN (To Network) queue and the FN (From Network) queue. When the user process issues a control block in a `write()`, this invokes the device driver, which queues the control block in the TN queue for the protocol processor and returns control to the user. When the control block command completes, the protocol processor places an acknowledgment block in the FN queue and interrupts the host. This interrupt invokes a device driver interrupt routine which in turn signals the user process. A signal handling routine in the user process performs a `read()` in order to access the acknowledgment. A 4 MB memory region on the protocol processor board is allocated as a buffer pool which holds incoming and outgoing message data.

3 Performance

In order to illustrate the overall performance of our architecture, we provide throughput and latency measurements at its MAC, transport, and user levels for the full range of message sizes available at each. We also include a profile of host processing time for a `SEND_MESSAGE` operation.

3.1 MAC Layer

The MAC layer provides the transport protocol with a raw data link service over the 100-Mbit/s FDDI network. Our results were obtained using a pair of NP and 167 boards in two stand-alone VME card cages. Figure 2a shows end-to-end latency, and Figure 2b shows throughput. Here, latency is half of the round-trip time of a frame, and throughput measures the rate at which the MAC driver can transmit frames with no receiver. A minimum latency of 91.5 ns occurs for a frame with no payload, and the maximum throughput of 56.6 Mbit/s occurs for frames carrying a payload of 4487 bytes (4500-byte maximum FDDI payload less LLC and SNAP headers).

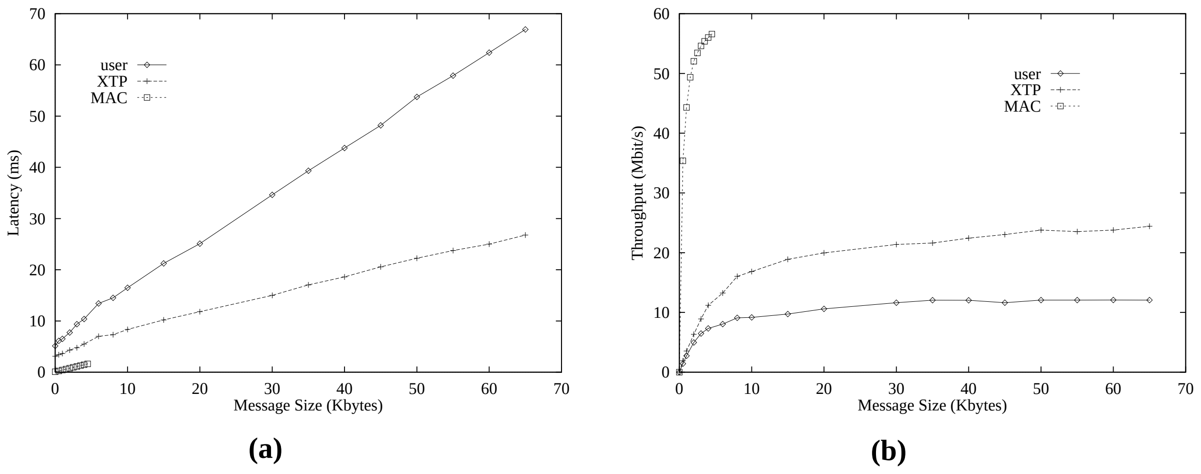

Figure 2: (a) End-to-End Latency vs. Message Size (b) Throughput vs. Message Size

3.2 Transport Layer

The transport protocol (XTP) provides reliable end-to-end delivery of memory buffers from the local memory of one protocol processor board to another. For the throughput measurements, the data transfer operations were performed in asynchronous mode, and data checksums were disabled with the XTP `NOCHECK` option. Latency measures half the round-trip time of a message, and throughput is for connection-oriented message transmission. The minimum latency of the protocol is 2.7 ms for a one-byte message, and the maximum throughput of 23.8 Mbit/s occurs for a 64-Kbyte message.

3.3 User Level

All user performance measurements were obtained with two Ada programs running on separate hosts using the connection-oriented `SEND_MESSAGE` and `GET_MESSAGE` primitives. Latency measures half the round-trip time of a message sent from one Ada application to another. For the throughput measurements, the communication primitives were performed asynchronously, and XTP's rate control features and `NOCHECK` option were used to provide maximum performance. `RATE` was set to 1.5 Mbyte/s and `BURST` was set to 10 Kbyte/burst. The minimum latency occurred at a message size of one byte and was 5.1 ms. The maximum throughput was 12.1 Mbit/s at a message size of 64 Kbytes.

3.4 Profiling

To analyze the user-level results, we profile the execution of the `SEND_MESSAGE` primitive. First we concern ourselves with the processing of short messages. Table 1a lists where the host processor spends its time during a one-byte `SEND_MESSAGE` call. The "other" category accounts for operations which consume less than 10 μs and accumulated measurement error.

The wait for completion is the amount of time that the host awaits an indication that the operation is complete. Although it accounts for most of the total time, the wait does not require host processor cycles; rather, it is a function of the transport protocol's performance. All other times in the table are incurred by various UNIX services. Most significant is the time required to perform the processing of `physio()` and `iodone()`, routines which manage the arguments of the `read()` and `write()` system calls. It is clear that UNIX overhead dominates the host processing time when the message size is small.

To study the processing overhead of long messages, we profile a `SEND_MESSAGE` of 64 Kbytes in Table 1b. As before, the total time is dominated by the wait for completion. However, with the long message size, the cost of allocating and deallocating a buffer on the local memory of the protocol processor now becomes notable. Furthermore, a startling result is the time required to perform the data `copyin()` across the VMEbus. For long messages, the time to perform this backplane transfer dwarfs all UNIX overhead and even rivals the wait for completion.

Operation	µs/call	Calls	total µs
wait for completion	3,881	1	3,881
`physio()` & `iodone()`	301	3	903
`read()` system call	81	2	162
signal delivery	143	1	143
`write()` system call	83	1	83
disable signals	25	3	75
enable signals	22	3	66
control block `copyin()`	29	1	29
ack block `copyout()`	23	1	23
other			732
total			6,097

(a)

Operation	µs/call	Calls	total µs
wait for completion	23,455	1	23,455
data `copyin()`	18,164	1	18,164
`physio()` & `iodone()`	301	3	903
`read()` system call	81	2	162
signal delivery	143	1	143
get and return buffer	109	1	109
`write()` system call	83	1	83
disable signals	25	3	75
enable signals	22	3	66
control block `copyin()`	29	1	29
ack block `copyout()`	23	1	23
other			1,636
total			44,848

(b)

Table 1: (a) Profile of a One-Byte `SEND_MESSAGE` **(b) Profile of a One-Byte** `GET_MESSAGE`

4 Conclusions

Our results lead to several conclusions. A large percentage of host processing time is spent in both the `SEND_MESSAGE` and `RECEIVE_MESSAGE` operations performing programmed I/O copies to or from the protocol processor buffer pool. Though all implementations will incur some amount of "peripheral commanding" overhead, the fact that we were unable to have the protocol processor address host memory effectively led us to use host cycles for buffer copies across the VME backplane. We believe this is a problem inherent in our host's memory architecture, not a problem inherent in off-host protocol processing and thus it should not be considered an inherent cost of an off-host processing architecture. For example, the introduction of user buffer DMA capabilities would reduce the host's VME data copy overhead to a constant DMA setup time, not dependent on data length.

The time spent for a user to command the protocol is predictable since the command block is submitted to the processor and control returns directly to the system. Control returns to the user in a deterministic fashion such that other activities can be done; the application need not wait on the completion of the transfer to regain control.

Preliminary measurements show the transmission of one 64-Kbyte message produces 26 host interrupts for a UNIX in-host implementation of XTP, compared to 1 host interrupt for our off-host implementation. This is a reduction in host-processor interrupts of 96%. The ability of the attached processor to field network interrupts and provide the MAC interface also decreases work the host must perform for communication processing. Results of a in-kernel implementation also show transport level throughput to be substantially lower than that of the off-host implementation. This shows that the lightweight operating system of the attached processor is a better match to the performance demands of communications [WATE93].

References

[STRA92] W. T. Strayer, B. J. Dempsey, A. C. Weaver, *XTP: The Xpress Transfer Protocol*, Addison-Wesley, Reading, Massachusetts, 1992.

[WATE93] A. S. Waterman, "A Comparison of Off-Host vs. In-Kernel Communications Architecture," M. S. Thesis, Department of Computer Science, University of Virginia (in preparation).