# On the Design and Semantics of User-Space Communication Subsystems

Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy
University of Karlsruhe, Dept. of Informatics*
Am Fasanengarten 5, D-76128 Karlsruhe, Germany

In: Proceedings of the International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'99), June 28th - July 2nd 1999, Las Vegas, Nevada, Vol. I, pp. 2344--2350

**Abstract**

*The problem with Gbit/s networks is to get the hardware performance into the applications. The most promising technique is a zero-copy protocol combined with a user-space communication subsystem that (a) gives the application direct access to the network interface and (b) avoids all buffering/copying.*

*In this paper we examine the design space of user-space communication subsystems, especially how send and receive operations work and which communication semantics they imply. Furthermore, we propose a technique called* page-exchange *which avoids copy operations, is well defined, and has communication semantics equivalent to those of standard programming interfaces such as UNIX sockets, PVM, or MPI.*

*Keywords:* User-Space Communication, Zero-Copy Protocols, High-Speed Interconnects, High-Performance Cluster Computing.

## 1 Introduction

Modern high-speed networks, such as Myrinet [5], allow cost effective high-performance clusters to be built from commodity PCs and workstations. Traditional communication subsystems (e.g. kernel based TCP/IP protocol stack), however, are unable to deliver anything close to the raw hardware performance. To solve this problem, several *user-space communication* subsystems have been developed. They remove the operation system from the critical communication path, giving the application direct access to the network interface and avoiding copy and buffer operations.

In terms of performance, all user-space communication subsystems are superior to the traditional approach. But some of them define a new and different semantics on send and receive operations. The classical semantics of send and receive operations is defined in standardized programming interfaces such as UNIX sockets, PVM, or MPI: Messages are buffered at the sender and the receiver (unless the programmer requests something different), such that (a) send buffers can be reused easily, (b) sending and receiving are independent, and (c) message reception is decoupled from message delivery. In contrast, several of the user-space communication subsystems define a different semantics. For example, the BIP [12] protocol requires that *"send and receives have a rendez-vous semantics where the receive needs to be posted before or at least 'not too long' after the send has begun."* This is not the only example. A similar passage can be found in the Virtual Interface Architecture Specification (VIA [14]): *"The VI Consumer on the receiving side must post a Receive Descriptor of sufficient size before the sender's data arrives."* It should be obvious that pro-

*Now: Scarasoft AG, Mühlfelder Straße 10, 82211 Herrsching, Germany. Email: {blum,warschko}@scarasoft.com

gramming with these primitives is more difficult than with the classical, buffering primitives. The alternative is to place a buffering layer on top of these primitives which, alas, reduces performance back to where we started from.

The following section explains the design space and semantics of different send and receive operations. Section 3 will discuss the semantics of 'pre-posting receive buffers' in detail, and section 4 shows that designs with classical semantics perform as well as non-standard approaches.

# 2 Sending and Receiving Messages

Figure 1 shows the major components to consider when designing send and receive operations.

## 2.1 Sending a Message

**PIO:** Programmed I/O (PIO) assumes that the network interface is mapped into the address space of an application (user-space principle). Transferring user data to the network interface is accomplished by issuing a simple copy operation together with some control information to the network interface (NI). Then the NI transmits the data to its destination. The two arrows labelled *PIO* in figure 1 describe the PIO mechanism.

**CopyDMA:**
This technique distinguishes data transfer from control transfer. Control (protocol) data is written directly to the network interface (using PIO). The user data is redirected (copied) to a DMA buffer which resides in a DMA-capable memory region. Usually this buffer resides within the kernel, but in most systems a pinned consecutive memory region is sufficient. After the copy stage the NI fetches the data using a DMA operation. CopyDMA is illustrated by the arrows labelled *ControlPIO*, *Copy*, and the *DMA* arrow in figure 1.

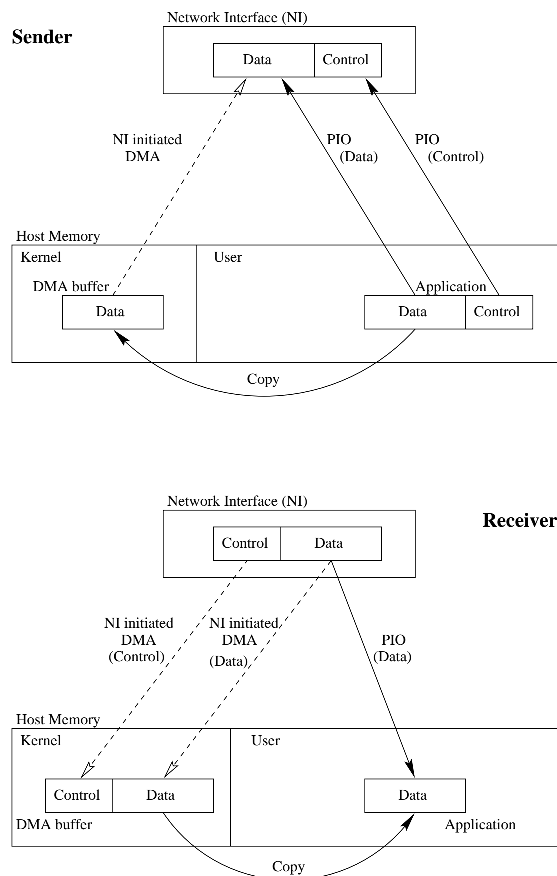


Figure 1: Data transfer during send and receive operations

**DirectDMA:** DirectDMA assumes that all data an application has to send reside in pinned, DMA-capable memory such as the DMA buffer in figure 1. By allocating a DMA buffer at application startup, the send mechanism avoids the copy operation between application and kernel memory. Thus DirectDMA uses only the *Control-PIO* and the *DMA* arrow in figure 1.

**V2PDMA:** This technique replaces the *Data-PIO* arrow in figure 1 with a DMA operation. In order to do this, a translation of virtual to physical addresses (*V2P*) is needed, which is accomplished within the operating system kernel. Furthermore, the selected pages have to be pinned down (marked unpageable) to prevent the pager process from replacing them during a DMA transaction. After the DMA operation has finished, the pages are either unpinned to free resources or may stay pinned in case of a page reuse (to avoid a second V2P translation).

**PEXDMA:** Page-exchange-DMA uses the same mechanism as V2PDMA, except that the pinned-down pages are marked as *copy-on-write*. This has the effect that the pages are duplicated if the application tries to modify the content during an ongoing DMA transaction. Thus the DMA operation always transfers correct data and PEXDMA provides the classical semantics of send operations (safe reuse of send buffers). After the DMA operation has finished, the *copy-on-write* flag is checked to detect if the pages have been duplicated. In that case, pages are unpinned and released to the system pool of free pages. Otherwise the *copy-on-write* flag is cleared and the application can reuse the pages. As within V2PDMA, reused pinned pages may or may not be unpinned.

CopyDMA is the classic technique; all communication subsystems for LAN and WAN networks are based on this principle. With a slight modification, which allows bypassing the protection boundary between kernel and user memory, it is used in high-speed communication subsystems such as Generic AM [15], PM [13], and ParaStation2 [16]. The PIO approach is also used in user-level communication subsystems, such as FM [11], LFC [2], and AM-II [6]. PIO and CopyDMA offer a message-passing model, whereas DirectDMA has a memory-mapped communication model in mind. Examples for DirectDMA are SCI [9], SHRIMP [4], VMMC [7] and Digital's MemoryChannel [10]. V2PDMA is used within BIP [12] and VMMC-II [8]. PEXDMA is being explored with ParaStation2.

Besides the different communication models – message passing vs. memory mapped communication – there is a difference in how send buffers are handled during a send operation. With PIO, CopyDMA, or PEXDMA it is safe for an application to reuse its send buffers as soon as the send operation returns. Thus, these techniques provide the 'classical' semantics of send operations. In DirectDMA and V2PDMA, however, the application has to test whether a send buffer can be reused by calling some kind of `DMA-Ready()` operation. Although DirectDMA and V2PDMA may have performance advantages, they have different send semantics, namely that the application has to be careful when reusing send buffers. In contrast, PEXDMA is likely to offer the same performance level as V2PDMA, but it provides the 'classical' send semantics of PIO and CopyDMA.

## 2.2 Receiving a Message

Receiving messages with PIO, CopyDMA, DirectDMA, and V2PDMA works similar to sending, except that the data flow is reversed (see figure 1). PIO is normally not used for receiving because of unacceptable transfer rates [1, 3].

At the sender, PEXDMA can be viewed as a refinement of V2PDMA. As receive operation, however, the PEXDMA technique is an improvement over CopyDMA. PEXDMA

avoids the copy step of the CopyDMA approach (see receiver in figure 1). The DMA operation which transfers the incoming data from the NI to the DMA buffer is unchanged. If the application issues a receive operation, PEXDMA exchanges DMA buffer pages with application memory pages whenever possible. In order to do this, a translation of virtual to physical pages as well as a modification of the application's page tables are needed. This is accomplished within the operating system kernel. Upon a page exchange each affected DMA buffer page is unpinned (because it now belongs to application memory), each affected application page is pinned down (because it is now used as part of the DMA buffer), the application's page table is modified (because the virtual address of the exchanges page has now a different physical representation), and the buffer addresses in NI memory are updated. After these modifications, the application has 'received' its data, while the size of the DMA buffer remains constant. Nevertheless, the PEXDMA mechanism also has some drawbacks. First of all there is a performance tradeoff between CopyDMA and PEXDMA (similar to that of PIO vs. DMA); for a small amount of data CopyDMA is likely to be faster than the PEXDMA mechanism (due to the necessary system call). Furthermore, a prerequisite for PEXDMA is that the receive memory in application space must be page aligned; otherwise a fallback mechanism to Copy-DMA is invoked.

The semantic differences of the five techniques become obvious when looking at the possibility of explicit or implicit receive buffering in host memory. If receive buffering is possible, the reception of messages from the network is completely decoupled from the delivery of these messages to the application. An incoming message is stored in an intermediate buffer and if the application issues a receive operation, a prior buffered message can be delivered. If receive buffering is not possible, the receiving application has to be ready to receive at the time a message arrives. There is no decoupling of message reception and message delivery any more. Receiving a message in these models is either accomplished by a rendez-vous principle between sender and receiver (the receiver has to be started before the sender will transmit the data) or receive buffers have to be pre-posted so that an incoming message can be stored in these pre-posted buffers. In addition, the size of the pre-posted buffers have to fit the size of incoming messages.

CopyDMA (AM, FM, PM, ParaStation2) and PEXDMA use receive buffering whereas Direct-DMA (SCI, VMMC, MemoryChannel) and classical V2PDMA (BIP) does not. VMMC-II [8] uses a hybrid method of CopyDMA and V2PDMA called *transfer redirection*, which buffers incoming messages in case that the application is not yet ready to receive. Otherwise the message is directly transferred to application memory. PEXDMA is being explored with ParaStation2.

## 3 Communication Setup

Communication models which omit default buffering in host memory at the receiver have to define how the user-level receive buffers have to be established. This is necessary, because the communication hardware has to know where to store incoming messages before the message arrives.

Using a memory mapped communication model, such as DirectDMA, the receive buffers are established at application startup. For each memory area which is shared with another node, an appropriate mapping is defined. So called *inbound* mappings grant other nodes write access to a specified memory area and *outbound* mappings request write access to a memory area from another node. Obviously an inbound mapping on node A has to correspond to an outbound mapping on node B. Once all the application instances are started and all corresponding mappings have been established, the startup phase completes. After that no further handling of communication relationships is needed – as long as the model does not allow dynamic process creation – and

incoming messages are delivered directly to the application.

V2PDMA instead requires that prior to each communication sufficient receive buffers of appropriate size are preposted to the communication subsystem. The problem with this demand is twofold. First, for each communication request the total amount of data packets being received as well as the size of each data packet has to be known in order to be able to prepost the necessary receive buffers. In case of regular communication patterns (e.g. ring-shift) with a fixed amount of data to transmit, calculating the necessary buffers to prepost is simple. But with irregular, data dependent communication patterns and variable amounts of transmitted data this task can be hard if not impossible. The second problem concerns the point in time when it is safe to start a communication operation. Using the same example as above (ring-shift), each sender has to be sure that it's communication partner (the next node, assuming a ring-shift) has posted the necessary receive buffers before it can start sending messages. The question is how to ensure this – other than using a synchronization primitive, which involves a communication operation for which the necessary receive buffers have to be preposted. This results in the classical *chicken and egg* situation. How to solve this problem is often left open by communication subsystems proposing V2PDMA with buffer preposting. Obviously, there are practical solutions to this problem but this does not provide a well defined semantics of preposting receive buffers in general.

In contrast to the shown difficulties of V2PDMA, receiving messages in all other communication models is well defined, much simpler and in our opinion more intuitive. Furthermore, very few existing applications are assumed to fit a model with preposting receive buffers.

# 4 Performance

Although the main focus of this article is on the semantics of different send and receive operations the major goal of all user-space communication subsystems is performance. If one method clearly outperforms another, it is hard to find arguments for the slower approach. But if two methods perform roughly the same, an application programmer is likely to prefer the one with the more common or more convenient semantics.

In our evaluation, we use a pair of Pentium II systems, running at 350 MHz and using the BX chipset (ASUS P2B-LS board). Each system is equipped with 128 MByte DRAM, 4 GByte UW-SCSI disk, FastEthernet, and a Myrinet adapter. The PCI bus runs at 33 MHz and is 32 bit wide, which results in 133 MByte/s maximum bandwidth. The operating system is Linux-2.0.35.

Figure 2 shows the performance for the different send techniques presented in section 2.1. During send analysis, only the protocol information is passed to the host code at the receiving side. This resembles an receiver with an infinite bandwidth to receive user data, and the sender performance is not limited by the chosen technique at the receiving side.

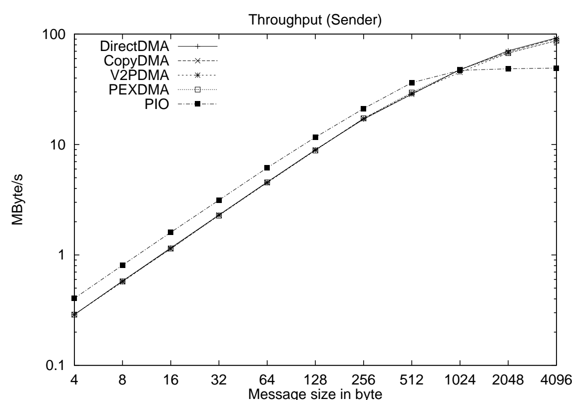


Figure 2: Performance at sending side

Surprisingly, all DMA based sending techniques perform roughly the same (within a few percent). This proves the assumption that copy operations while pipelining them with

successive DMA operations hardly affect the overall performance. Only PIO behaves differently and shows a clear advantage for small packets (up to 1024 byte), but for large packets the throughput saturates at 50 MByte/s. The maximum throughput for the DMA based techniques is 91 MByte/s (CopyDMA), 92 MByte/s (DirectDMA), and 87 MByte/s (V2PDMA and PEXDMA) respectively. The message transfer unit (MTU) was limited to 4096 Byte, because for page based approaches such as V2PDMA and PEXDMA it's hard to handle larger MTU sizes. As the figure shows, the DMA based approaches do not show a saturation effect, so increasing the MTU further will result in an increased throughput until the PCI bus saturates.

Figure 3 shows the performance for the different receive techniques presented in section 2.2. During receive analysis, only the protocol information is passed to the network interface at the receiving side. This resembles a sender with an infinite bandwidth to transmit user data, and the receiver performance is not limited by the chosen technique at the sending side.

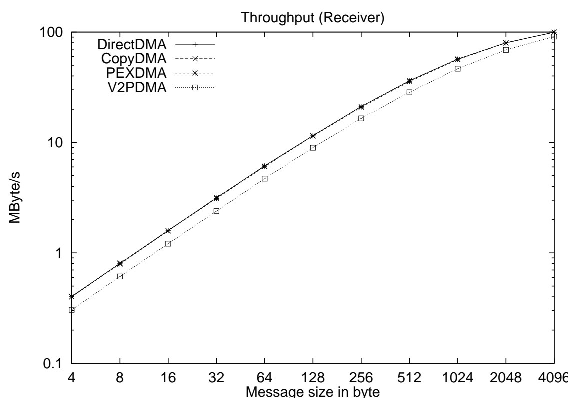


Figure 3: Performance at receiving side

Again, all techniques show an equivalent level of performance. As on the sending side, copy operations hardly affect the overall performance while pipelining them with previous DMA operations. The lower performance of V2PDMA might be due to the difficult handling of receive buffers to keep the transmission pipeline running, and we are sure that further optimizations for V2PDMA exist that will result in the same performance as the other implementations. The maximum throughput of most techniques (DirectDMA, CopyDMA, and PEXDMA) is 99 MByte/s, and 91 MByte/s for V2PDMA. These numbers are up to 12 MByte/s higher than the maximum throughput on the sending side. Thus, overall performance of a data transmission is limited by the sender and not the receiver.

As said earlier in this section, the major motivation for implementing different send and receive techniques was performance. But according to the presented results, the differences are negligible. True zero copy techniques (DirectDMA and V2PDMA) do not outperform techniques which use copy operations, as long as the copy operations are pipelined with previous or successive DMA operations. Thus, the major motivation for choosing a specific communication model is likely to depend on the semantics of that model.

# 5 Conclusion

In this paper we examine the design space of user-space communication subsystems, especially how send and receive operations work and which communication semantics they imply. We identify five different models, namely PIO, DirectDMA, CopyDMA, V2PDMA, and PEXDMA which have a different semantics at the sending and the receiving side. While sending a message, PIO, CopyDMA, and PEXDMA offer the 'classical' send semantics known from common communication subsystems such as UNIX sockets, PVM, and MPI (on default). They buffer the outgoing data so that the application is free to reuse its send buffer without any restriction. In contrast to that, DirectDMA and V2PDMA force the application to ensure that a previously used send buffer can be reused safely.

At the receiving side the semantic differences are due to explicit or implicit receive buffering in host memory. Again, CopyDMA and

PEXDMA offer the 'classical' receive semantics, because incoming messages are buffered in host memory. Within DirectDMA, receive buffers have to be established during application setup, and afterwards incoming messages are delivered directly to the application. This condition implies that the application has to know a priori which parts of its memory are shared (read or write) and which node is going to access a shared memory region. If this mapping is known at application startup, DirectDMA works fine. Otherwise, a transmission buffer has to be established and in this case DirectDMA behaves exactly like CopyDMA. In contrast to all other methods, V2PDMA requires that prior to each communication sufficient receive buffers of sufficient size are preposted to the communication subsystem. Otherwise, an additional software layer is needed which emulates a buffering communication subsystem.

Prior to the writing of this paper, there was a strong evidence that CopyDMA and PEXDMA offer the same level of performance as DirectDMA and V2PDMA, while still providing the 'classical' communication semantics. Now the evidence has turned to confidence and the conclusion is that CopyDMA or PEXDMA are superior to other models, because application programmers do not have to rewrite their code to fit a new communication semantics.

## References


[1] Soichiro Araki, Angelos Bilas, Cezary Dubnicki, Jan Edler, Koichi Konishi, and James Philbin. User-space communication: A quantitative study. In ACM, editor, *SC'98: High Performance Networking and Computing: Proceedings of the 1998 ACM/IEEE SC98 Conference: Orange County Convention Center, Orlando, Florida, USA, November 7–13, 1998.* ACM Press and IEEE Computer Society Press, November 1998.

[2] Raoul A. F. Bhoedjang, Tim Rühl, and Henri E. Bal. LFC: A Communication Substrate for Myrinet. In *Fourth Annual Conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, June 1998.

[3] Raoul A. F. Bhoedjang, Tim Rühl, and Henri E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):52–60, November 1998.

[4] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA), Chicago, Illinois*, pages 142–153, Los Alamitos, California, April 18-21, 1994. IEEE Computer Society Press.

[5] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jarov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[6] B. Chung, A. Mainwaring, and D. Culler. Virtual Network Transport Protocols for Myrinet. In *Hot Interconnects'97*, Stanford, CA, April 1997.

[7] C. Dubnicki, A. Bilas, K. Li, , and J. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *11th Int. Parallel Processing Symposium*, pages 388–396, Geneva, Switzerland, April 1997.

[8] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos N. Damianakis, and Kai Li. VMMC-2: Efficient support for reliable, connection-oriented communication. Technical Report TR-573-98, Princeton University, Computer Science Department, February 1998.

[9] IEEE. *IEEE – P1596 Draft Document. Scalable Coherence Interface Draft 2.0*, March 1992.

[10] James V. Lawton, John J. Brosnan, Morgan P. Doyle, Seosamh D. Ó Riordáin, and Timothy G. Reddin. Building a High-performance Message-passing System for MEMORY CHANNEL Clusters. *Digital Technical Journal*, 8(2):96–116, 1996.

[11] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, San Diego, California, December 3-8 1995.

[12] L. Prylli and B. Tourancheau. Protocol Design for High Performance Networking: A Myrinet Experience. Technical Report 97-22, LIP-ENS Lyon, July 1997.

[13] H. Tezuka, F. O'Carrol, A. Hori, , and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *12th International Parallel Processing Symposium*, pages 308–314, Orlando, Florida, Mar 30 - Apr 3, 1998.

[14] VIA. *The Virtual Interface Architecture Specification, Version 1.0, http://www.via.org*, December 1997.

[15] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19st Annual International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[16] Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. Design and Evaluation of ParaStation2. In *Proceedings of the International Workshop on Distributed High Performance Computing and Gigabit Wide Area Networks*, Lecture Notes in Control and Information Sciences, Essen, Germany, September 1 - 5, 1998. Springer. *To appear.*