VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication

Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos Damianakis, and Kai Li
Computer Science Department, Princeton University, Princeton, NJ-08544
{dubnicki, bilas, yuqun, snd, li}@cs.princeton.edu

Abstract

The basic virtual memory-mapped communication (VMMC) model provides protected, direct communication between the sender's and receiver's virtual address spaces, but it does not support high-level connection-oriented communication APIs well. This paper presents *VMMC-2*, an extension to the basic *VMMC*. We describe the design, implementation, and evaluate the performance of three mechanisms in *VMMC-2*: (1) a user-managed TLB mechanism for address translation which enables user libraries to dynamically manage the amount of pinned space and requires only driver support from many operating systems; (2) a transfer redirection mechanism which avoids copying on the receiver's side; (3) a reliable communication protocol at the data link layer which avoids copying on the sender's side.

To validate our extensions we implemented stream sockets on top of the *VMMC-2* running on a Myrinet network of Pentium PCs. This zero-copy sockets implementation provides a maximum bandwidth of over 84 Mbytes/s and a one-way latency of 20 μs.

1 Introduction

With the arrival of fast networks such as Myrinet and ATM, the overhead of the network communication software has become a major performance bottleneck, hurting user-to-user latency and bandwidth. For example, the TCP/IP protocol overhead increases the latency by several hundreds microseconds [24] even for small messages. A copy in the network protocol can reduce the communication bandwidth to the memory copy bandwidth which is usually substantially less than the peak hardware DMA bandwidth. For example, while the Myrinet hardware is capable of delivering bandwidth of 1.28 Gbits/s [7], the end-user applications see little performance improvement compared to the 100 Mbits/s Ethernet [25].

The desire to build high-performance servers from a network of commodity computers has pushed researchers to explore ways to reduce the communication bottleneck for system area networks. During the last few years, researchers have proposed several approaches to move network protocols partially or entirely to the user level with protection in order to reduce the protocol overhead [3, 15, 18, 19, 36]. The common framework is to implement a user-level data transport protocol that integrates well with the network interface. Existing high-level protocols for shared virtual memory [21] and message-passing [1, 5, 14, 32] can then be implemented efficiently. The goal is to deliver to the user communication performance close to the hardware's limit while providing the full functionality of the high-level APIs. This paper addresses three of the important issues associated with this goal.

The first challenging issue is how to design and implement the low-level data transport API so that high-level APIs can be implemented with little overhead and with true *zero-copy* protocols. A memory-mapped communication model, such as *SHRIMP* [6] or DEC's Memory Channel [20], requires the least software overhead because data transfers avoid receiver's CPU intervention. However, our early experience has shown that using such a low-level API to implement a high-level connection-based API like stream sockets often requires a copy on the receiver side [14]. This is because the basic memory-mapped communication model requires the sender to know the destination address before it sends data. On the other hand, high-level connection-oriented APIs such as stream sockets do not have destination addresses on the sending side; a sender knows only the name of the connection. As a result, zero-copy protocols typically require a scout message or a handshake in the implementation [1, 14, 32]. In this paper, we propose, implement and evaluate a mechanism called *transfer redirection*. We show that this mechanism naturally extends the virtual memory-mapped communication model and provides support for connection-oriented APIs to achieve zero-copy without any additional messages.

The second challenging issue is how to integrate the low-level communication mechanism with standard operating system facilities such that protected, user-level data transfers can use pageable virtual memory without copying. The dilemma is that applications use an "infinite" amount of pageable virtual memory while the network interface DMA uses physical memory addresses. Our previous implementations of virtual memory-mapped communication required static pinning of all send and receive buffers. This paper describes a novel method that allows a user library to dynamically manage the amount of pinned memory used for both send and receive with transfer redirection operations. This approach requires only standard support available in most existing operating systems. As a result, one can realize protected, user-level communication without modifying operating systems and without pinning a large amount of memory for communication. Our implementation and measurements show the amortized overhead of this approach is small.

The third challenging issue is where and how to incorporate a retransmission protocol to provide low-overhead, reliable communication for system area networks. Traditionally, reliable end-to-end connections have been implemented in the TCP layer with retransmission. This is a satisfactory design for wide area networks but the overhead of TCP is significant, in particular, when it is implemented on the host computer. This is because a retransmission protocol requires a sender process not to touch its send buffer until the message is acknowledged. The sender has to either wait for the acknowledgment, copy the send buffer to a system buffer, or copy-on-write for the send buffer. Our approach is to implement the standard retransmission method at the data link level, that is, between network interfaces. This approach takes advantage of the buffering of outgoing packets on the network interface and eliminates the need for the sender to wait for acknowledgments or to copy send buffers. We show that the loss of bandwidth of this method is small and its overhead on latency is only a few microseconds.

To demonstrate these techniques and their integration, we have designed an extended virtual memory-mapped communication model *VMMC-2* and implemented it on PCs connected with Myrinet. This is a completely new implementation, not an incremental improvement of VMMC. We also implemented the stream sockets protocol on top of *VMMC-2* to demonstrate how to take advantage of the new features. Our micro-benchmarks show that the one-way latency of *VMMC-2* is 13.4 μs with transfer redirection and retransmission and the maximum bandwidth is over 90 MBytes/s. The one-way latency of the stream sockets layer is 20 μs and its bandwidth is over 84 MBytes/s. We experimented with several applications using the stream socket facility and showed that these techniques are indeed effective.

2 From *VMMC* to *VMMC-2*

2.1 The Basic *VMMC* Model

Virtual memory-mapped communication (*VMMC*) is a communication model that provides direct data transfer between the sender's and receiver's virtual address spaces.

The *VMMC* mechanism achieves protection by requiring that data transfers take place only after the receiver gives the sender permission to transfer data to any area of the receiver's address space. The receiving process expresses this permission by *exporting* areas of its address space as receive buffers. A sending process must *import* remote buffers that it will use as destinations for transferred data.

The basic data transfer operation supported by *VMMC* is an explicit request to transfer data from the sender's local virtual memory to a previously imported receive buffer. Any valid local address can be used as a send buffer. When a message arrives at its destination, it is transferred directly into the memory of the receiving process, without interrupting the receiver's CPU. Thus, *there is no explicit receive operation in VMMC.*

VMMC also provides *notifications* to support transfer of control and to notify receiving processes about external events. Attaching a notification to a message invokes a user-level handler function in the receiving process after the message has been delivered into the receiver's memory.

2.2 Lessons from *VMMC* Implementations

The basic *VMMC* model has been implemented on two platforms: PCs connected by the Intel Paragon routing network [18] and PCs connected by a Myrinet [17] network. The first platform uses a custom-designed network interface supporting virtual memory-mapped communication and the second platform uses standard Myrinet programmable network interfaces and Myrinet routing switches.

Programming with *VMMC* directly delivers communication latency and bandwidth very close to the hardware limit. We already have a set of communication libraries such as NX, Sun RPC, stream sockets and shared virtual memory (SVM). User-level buffer management allowed us to implement zero-copy protocols for SVM and NX. *VMMC* is easy to use if the senders know receive buffer addresses.

However, the basic *VMMC* model and its implementations have some drawbacks.

The basic *VMMC* model does not support zero-copy protocols for connection-oriented high-level APIs such as stream sockets and NX. The sender does not know the destination address unless the receiver sends a scout message. As a result, our previous stream sockets implementation used a one-copy protocol.

The address translation (from virtual to physical) in the Myrinet *VMMC* implementation uses a firmware-managed cache on the Myrinet network interface that interrupts the host CPU on a miss. The overhead of an interrupt is quite high. Furthermore, such a design interacts in a complex way with the OS because it requires memory management in the interrupt handler.

The previous *VMMC* implementations do not provide reliable communication.

VMMC-2, described in the next section, overcomes these drawbacks.

2.3 *VMMC-2* Overview

VMMC-2 provides three novel features: user-managed TLB (UTLB) for address translation, transfer redirection, and reliable communication at the data link layer.

The *VMMC-2* implementation works under Linux OS version 2.0.24 running on 166 MHz Pentium PCs connected with Myrinet network interfaces (M2F-PCI32B) and Myrinet switches (M2F-SW8). Each network interface has a 33 MHz processor (LANai) and 1 MByte of static RAM.

Our system supports multiple senders per node and runs on both uniprocessor and SMP nodes. Figure 1 shows the structure of the *VMMC-2* implementation. It consists of the following components:

- A user library which implements the API of the *VMMC-2* model.
- Firmware for the Myrinet network interface that works with the user library to implement the data transfer and reliable communication.
- A device driver which provides *VMMC-2* specific support. We do not require any operating system modification except for this loadable driver.

When an application exports a buffer, it makes an `ioctl` call to the driver, which in turn pins the exported buffer's memory and sets up a descriptor for this buffer in the network interface's memory. Exported receive buffers must be word aligned but they do not have to be page-aligned. On an import the *VMMC-2* user library passes the import request to the firmware running on the local network interface which communicates with the remote node to match the

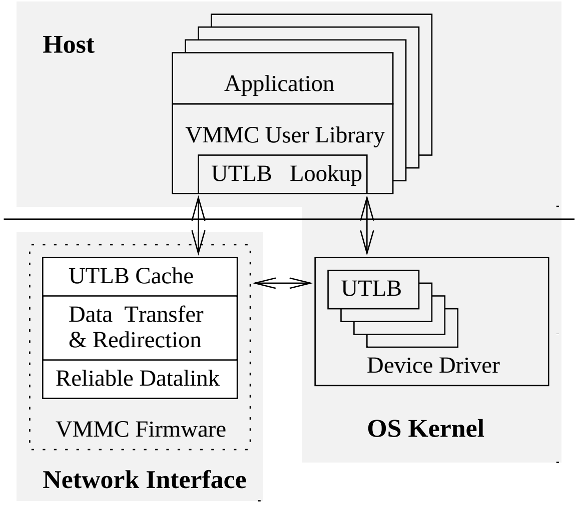


Figure 1: The architecture of the *VMMC-2* implementation on the Myrinet platform.

import with an export and to establish the export-import mapping. Unlike previous implementations, this one does not require Ethernet or any other network except Myrinet itself to establish a connection.

When an application requests a data transfer, the *VMMC-2* library uses the *VMMC-2* translation mechanism to pin the send buffer (if necessary), and to obtain the physical address of the send buffer which is passed to the network interface

The *VMMC-2* firmware uses DMA to move data, one page at time, from the host memory to the network interface outgoing buffers. Simultaneously, it transfers data from the outgoing buffers in the network interface to the network. When a message arrives, the *VMMC-2* firmware determines the destination address using the transfer redirection module; it then DMAs the incoming message from the network interface to the host memory.

VMMC-2 also implements reliable communication at the data link level using retransmission and sender-side buffering on the network interface.

The next three sections describe the three new features of *VMMC-2* and compare our work with previous approaches.

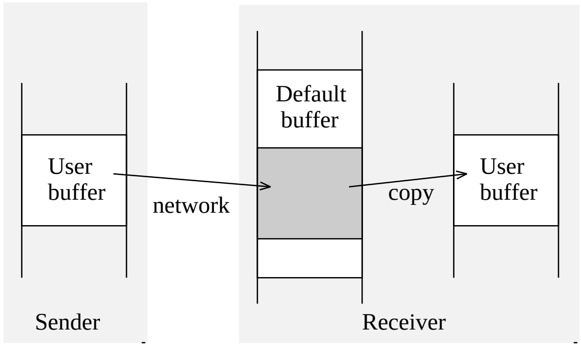
3 Transfer Redirection

3.1 Description

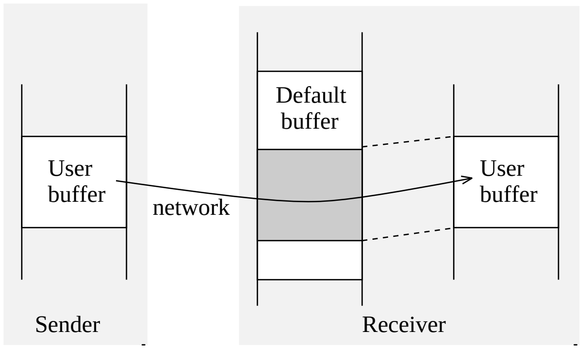
As described in Section 2.1, the basic virtual memory-mapped communication model provides protected, user-level communication to move data directly from a send buffer to a receive buffer without any copying. Since it requires the sender to know the receiver buffer address, it does not provide good support for connection-oriented high-level communication APIs.

The *VMMC-2* extends the basic *VMMC* model with a mechanism called *transfer redirection*. The basic idea is to use a default, *redirectable* receive buffer for a sender who does not know the final receive buffer addresses.

When the sender sends data, it always sends to the default buffer. When the data arrives at the receive side, the redirection mechanism checks to see whether a redirection address has been posted. If no redirection address has been posted, the data will be moved to the default buffer. Later, when the receiver posts the receive buffer address, the data will be copied from the default buffer to the receive buffer, as shown in Figure 2(a).



(a) A copy takes place when the receiver posts its buffer address too late.



(b) A transfer redirection moves data directly from network to the user buffer .

Figure 2: Transfer redirection uses a default buffer to hold data in case receiver posts no buffer address or posts it too late, and moves data directly from the network to the user buffer if the receiver posts its buffer address before the data arrives.

If the receiver posts its buffer address before the message arrives, the message will be put into the user buffer directly from the network without any copying, as shown in Figure 2(b). If the receiver posts its buffer address during message arrival, the message will be partially placed in the default buffer and partially placed in the posted receive buffer. The redirection mechanism tells the receiver exactly how much and what part of a message is redirected. When partial redirection occurs, this information allows the receiver to copy the part of the message that is placed in the default buffer.

3.2 Comparison with Other Approaches

A zero-copy protocol can be implemented with an additional network communication. In this approach the receiver informs the sender about the receive buffer, after which the sender is able to transfer data directly into it. Variants of this protocol have been used by NX implementation on Paragon [31] as well as early NX implementation on *VMMC* [1].

Active Messages(AM) [19] is a user-level communication mechanism based on fast remote execution facility. Recently, a fast sockets facility has been built with AM handlers [32]. They used a receiver-based approach called *receive posting*. The arriving data is first placed in an anonymous staging area in memory and the message handler is invoked. If a receive has been posted, the handler moves the data to the final destination bypassing the socket buffer. Similar to the redirection, this technique avoids the socket buffer copy. Unlike the redirection, receive posting requires a copy through anonymous memory but avoids the cost of pinning and unpinning user buffer pages.

Many methods have been proposed to avoid copying between I/O devices and application memory. IBM's OS/360 can specify a file I/O buffer area for applications to avoid copying. Page remapping has been used to avoid copying in several past and recent projects [26, 10, 22, 16, 28]. Brustoloni and Steenkiste proposed *emulated copy* [8] which combines page remapping and copying of sub-page data chunks. In this technique, total amount of data copied is not more than one page regardless of user buffer size and its alignment.

Lately, several groups have proposed letting applications directly access communication interface memory [11, 13, 33]. However, network interface memory is usually small and non-cacheable which limits its direct use by applications.

4 User-Managed Address Translation

4.1 Description

In the *VMMC-2* API, a user-level *send* or *redirect* request specifies an arbitrary user buffer with a virtual memory address and size. The user buffer must be pinned during the data transfer operation, after which it may be unpinned. The difficulty arises because the user process does not know physical addresses while the network interface needs physical addresses to initiate DMA. We resolve this mismatch by introducing a separate address translation mechanism called the *User-managed TLB* (UTLB). The UTLB consists of *per-process* array holding physical addresses of pages belonging to this process virtual memory regions that are pinned in the host physical memory. Each UTLB is allocated by the driver in the kernel memory (see Figure 1).

The user process identifies a buffer in its address space by a start index and count of contiguous entries in the UTLB. These entries contain the buffer's physical address translations. The network interface uses the (index, count) pair to directly obtain physical addresses for the buffer from the process' UTLB.

Although virtual-to-physical address translations are maintained at the page granularity, user buffers can start at word boundaries. A user request always includes the buffer's offset into the first page in addition to its length in words.

For protection, the *VMMC-2* library cannot read or modify the UTLB directly. Instead, the library manages the UTLB indirectly by asking the device driver to replace TLB entries and to provide physical addresses. The library maintains a lookup data structure (see Figure 1) to keep track of pages that have been pinned and whose translations are present in the UTLB. This data structure is consulted for every send and redirect request to obtain UTLB indices for the user buffer. If a lookup *misses* in the UTLB, (i.e., page translations are not present in the UTLB), the *VMMC-2* library makes an `ioctl` call to pin the required pages and add their translations to the UTLB.

The network interface can access its on-board SRAM much faster than the host physical memory. However, there is not enough SRAM to allocate multiple UTLBs that are large enough to be useful for real applications. Therefore, UTLBs are allocated in the host physical memory which can support an "unlimited" number of UTLBs. This results in a 3 μs increase in latency to read a UTLB entry. To alleviate this problem, we add a *UTLB cache* on the network interface (see Figure 1). It is a direct-mapped cache with 8192 lines, indexed by UTLB entry. Each cache line holds a UTLB entry from a process that is identified in the tag field.

4.2 Comparison with Other Approaches

Compared with existing approaches for address translation on the network interface, our UTLB mechanism can deal with arbitrary user buffers without the implementation complexity associated with an *interrupt-driven* approach.

Hamlyn [9] uses a limited number of registers to identify and keep translations of pinned user buffers. This mechanism is similar to our UTLB with two key differences: (1) it limits the number of translations to the number of registers; (2) it does not allow for buffer overlapping. By comparison, the UTLB mechanism can deal with a large number of small buffers and allow for buffer overlapping.

A typical interrupt-driven approach is used for address translation by both U-Net/MM [4] and *VMMC* [18]. This approach uses a global TLB on the network interface; when a TLB miss happens, the network interface interrupts the host processor which pins or pages-in the virtual page and returns the physical address to the network interface. Pinning pages in an interrupt context is non-standard and complicated. For example, the Linux implementation of U-Net/MM cannot use the standard kernel memory pinning routines because they do not work properly in an interrupt context. U-Net/MM also has to implement a page-in thread for user pages that are temporarily swapped out to the disk. By contrast, the UTLB approach avoids this complexity by managing the pinning and unpinning of user pages in the user-level library. The only OS service the UTLB requires is the standard memory pinning routines in the process context. These are standard services provided to device drivers by almost all existing operating systems. Therefore, UTLB can be ported to any OS platform with little effort.

Another important advantage of the UTLB is that it simplifies the design of the network interface. The network interface reads translations directly from the UTLB according to user requests, whereas the interrupt-driven approach requires non-trivial interaction between the network interface and the host processor.

Finally, our UTLB scheme facilitates zero-copy data receiving. To receive data directly we must know that receive pages are present; otherwise, if a page is swapped out we would have to buffer or drop the data. In U-Net/MM, if a receive page is not present the incoming packet is dropped. To avoid this problem, the *VMMC-2* library uses the UTLB and makes sure that receive pages are present and pinned when a receive (redirection) is posted.

5 Reliable Communication

5.1 Description

VMMC-2 deals with transient network failures and provides applications with reliable communication at the data link layer. The goal is to tolerate CRC errors, corrupted packets, and all errors related to the network fabric; links and switches can be replaced on the fly. If a transient network error becomes permanent, the remote node is declared unreachable, imported buffers from that node are invalidated, the user is notified, and all packets that cannot be delivered are dropped. In this case the user needs to reestablish the mappings in order to resume communicating with the remote node.

VMMC-2 implements a simple retransmission protocol at the data link layer–between network interfaces. Our scheme buffers packets on the sender side and retransmits when necessary. Each node maintains a retransmission queue for every node in the system. The available buffer space is managed dynamically so that no buffers need to be reserved for each node. Each packet carries a unique (for the sender-receiver pair) sequence number. Sequence numbers and retransmission information is maintained on a per-node and not per-connection basis. The receiver acknowledges packets. Each acknowledgment received by the sender, acknowledges (and frees) all previous packets up to that sequence number. There is no buffering at the receiver. If a packet is lost, all subsequent packets will be dropped. However, if a previously acknowledged packet is received again, it is acknowledged. In the current implementation, there are no negative acknowledgments for lost packets.

5.2 Comparison with Other Approaches

Most high-level communication APIs provide reliable communication to the user by implementing retransmission in one of the software layers in the protocol stack. Stream sockets, for example, are typically built on top of TCP/IP with reliable end-to-end connections implemented in the TCP layer. This design is suitable for wide area networks since network connections go through intermediate host systems and packets may need to be stored for a long time before they are acknowledged. However, this results in high overhead for TCP; highly tuned implementations add over a hundred microseconds of overhead.

We ruled out the design choice of implementing reliable communication at the library level. The main reason is that a retransmission protocol requires the sender process not to touch its send buffer until the acknowledgment is received. The sender has either to wait for the acknowledgment, copy the send buffer to a system buffer, or use copy-on-write for the send buffer. The overhead of these design choices is high.

A few low-level communication interfaces assume reliable hardware and treat each network error as a catastrophic one. Examples of such approach include Fast Messages [30] and our previous work [6, 17]. This works well as long as network errors are really extremely rare or when the packet retransmission is handled by the hardware (like S-Connect [29]).

6 *VMMC-2* Performance

6.1 The logP Model

We characterize the performance of our system using the five parameters of the logP model [12]: $(\frac{RTT}{2}, L, O_r, O_s, g)$.

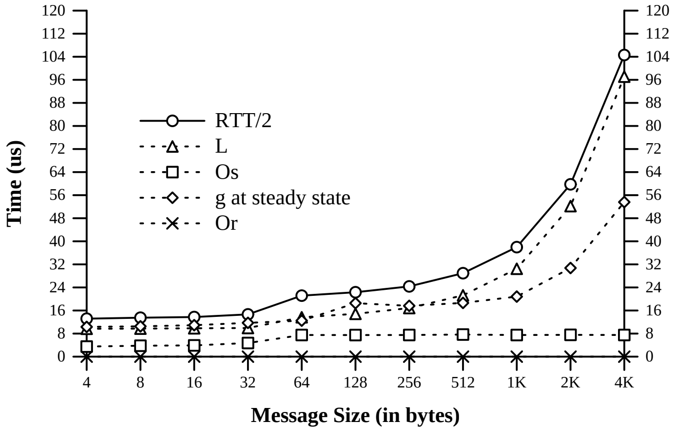


Figure 3: *VMMC-2*, logP numbers

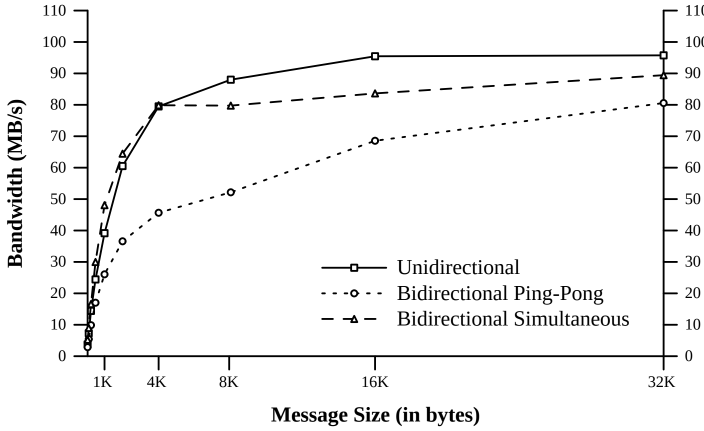


Figure 4: *VMMC-2*, Bandwidth numbers

$\frac{RTT}{2}$ is the one-way host to host latency for a ping-pong test. O_s, O_r are the host overheads of sending and receiving a message respectively. L is the one-way network latency between network interfaces. Finally g (gap) is the time between two successive sends and describes how fast messages can be pipelined through the system.

Figure 3 presents logP parameters as a function of the message size. All experiments have been performed with warmed-up UTLB cache to avoid cold misses. In *VMMC-2*, when a message is sent, data is written directly to the LANai send queue for sizes up to 32 bytes. For longer messages, LANai initiated DMA is used to transfer the data to the network interface. In both cases, the overhead to send a message is very small, below 4 μs. The one-way host-to-host latency ($\frac{RTT}{2}$) is 13.4 μs, and stays essentially flat for up to 32 bytes. The latency L between network interfaces closely tracks $\frac{RTT}{2}$. g, at steady state, is related to the bandwidth of the slowest stage in the pipeline and is affected by the complexity of the Myrinet firmware.

We also run three bandwidth tests: *Unidirectional* is the bandwidth of a one way transfer. In the *Bidirectional ping-pong* benchmark data flows in both directions in a ping-pong fashion. *Bidirectional simultaneous* is similar to the bidirectional ping-pong benchmark, only that both nodes start as senders. This benchmark can potentially use all three DMA engines of the network interface, depending on how the Myrinet firmware is written.

Figure 4 shows results for these three types of bandwidth. Bidirectional ping-pong bandwidth grows slower than the other two types because there is only one message in the network at any given time, so pipelining is less effective for short messages. For longer messages, data is transfered in 4 KBytes packets and unidirectional bandwidth catches up with the other two tests. The maximum bandwidth is in the range of 90-93 MBytes/s for all the tests. The 4 KBytes message bandwidth is about 44 MBytes/s for the bidirectional ping-pong test, and about 80 MBytes/s for the other two tests.

6.2 Cost of Transfer Redirection

Transfer redirection incurs additional overhead. This overhead includes posting redirection request by the host and computing new scatter lengths and addresses by the network interface. The cost to post a redirection request is about 6 μs provided UTLB contains already translations for pages of the redirection destination buffer. The additional work performed by the network interface takes about 2.5 μs per packet received. On our system, we can copy about 108 bytes in that time, so redirection is beneficial for all messages longer than 108 bytes provided it is posted ahead of the message arrival.

6.3 Cost of Address Translation

If the translations for a user buffer are present in the UTLB (a hit), the cost of address translation includes a *lookup* for the UTLB indices in the user-level *VMMC-2* library, and reading of UTLB entries in the SRAM by the network interface. The latter is negligible. The former is shown as the lookup cost in Table 1.

The major overhead in *VMMC-2* address translation comes from handling misses in the UTLB. There are two types of misses possible. The first, called host miss, is an expensive miss and happens when the library needs to ask the driver to pin pages and update UTLB entries. The cost of this miss includes the library overhead for allocating free UTLB entries and the device driver overhead for pinning user buffers. The second type, called NI miss, is much cheaper and involves the network interface fetching translations from an UTLB pinned in the host memory. The cost of this miss is dominated by the DMA time. All of these costs are listed as *VMMC-2* library, device driver, and NI-DMA overhead in Table 1.

	Operations	1 page (μs)	4 pages (μs)	16 pages (μs)
hit	lookup	1.5	1.5	1.5
host	VMMCII library	50	55	65
miss	device driver	25.7	29.2	50.3
NI miss	NI-DMA	3	3.1	4

Table 1: *VMMC-2* address translation cost

6.4 Cost of Reliable Communication

Table 2 shows the basic communication performance of *VMMC-2* with and without support for fault tolerance. We see that support for fault tolerance can be provided at low cost. Latency increases by about 2μs (from 11.1μs to 13.4μs) and bandwidth decreases by 2%-10%. The additional overhead comes from moving messages between queues and handling acknowledgments.

Performance Metric	RC	No RC	units
Latency	13.4	11.1	μs
Unidirectional b/w	92.63	97.61	MBytes/s
Bidirectional ping-pong b/w	90.05	99.39	MBytes/s
Bidirectional simultaneous b/w	90.77	91.87	MBytes/s

Table 2: *VMMC-2*, Basic performance numbers with and without support for reliable communication (RC)

7 Sockets Library and Its Performance

We have implemented a user-level stream sockets library to understand how well *VMMC-2* supports a reliable, high-level communication API. Our micro-benchmarks and applications show that *VMMC-2* supports the socket abstraction quite well.

7.1 Implementation

The sockets API is implemented as a user-level library using the *VMMC-2* interface. It is compatible, and seamlessly integrated with the Unix stream sockets facility [27]. We introduce a new address family, `AF_VMMC`, to support the new type of stream socket.

To take advantage of the transfer redirection mechanism in *VMMC-2* (see Figure 2), we use a circular buffer as a backup buffer for transfer redirection. In order to avoid copying, the redirection request must be issued before or during message arrival. The request will result in one of three outcomes:

- *full redirection*: all requested data was redirected because the request was issued before the message arrived.
- *partial redirection*: some contiguous part of the data was redirected because the request was issued while the message was arriving.
- *no redirection*: no data was redirected because the request was issued after the entire message arrived.

A `recv` call first checks the default buffer. If no data has arrived, it redirects the default buffer, at the appropriate offset, into the user buffer. It then spins waiting for data to arrive. Once data arrives, `recv` checks the result of redirection. In the case of full redirection `recv` completes without any copying. If the redirection request partially completes, or fails, `recv` must copy some, or all, of the message from the default buffer to the user buffer.

During our performance tuning, we found it beneficial to add an adaptive send delay (ASD) to stream sockets. This is because transfer redirection is much faster than a memory copy. We used an adaptive delay algorithm derived from a competitive, random walk algorithm [23]. The algorithm adjusts the delay threshold using feedback from the receiver. The delay varies between 0 and the time to copy the message.

The receiver gives the sender two types of feedback:

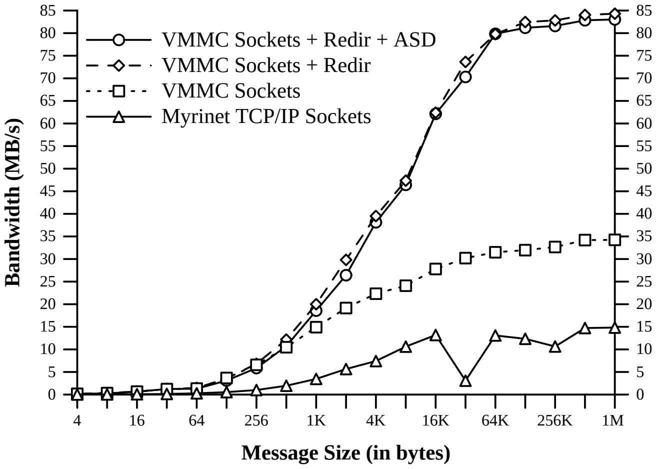


Figure 5: Sockets, Bandwidth for Bidirectional Ping-Pong

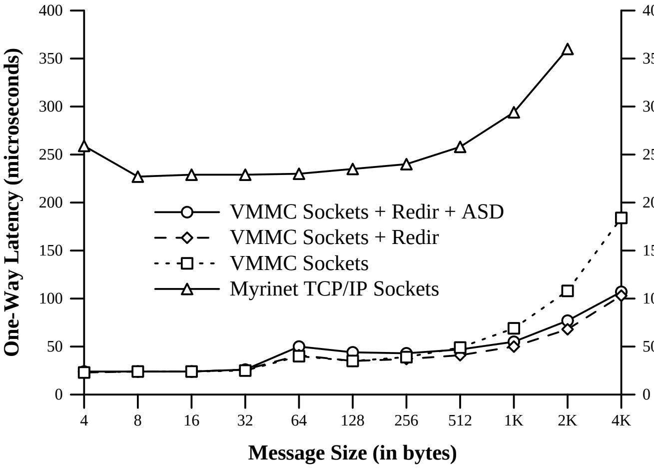


Figure 6: Sockets, One-way Latency for Bidirectional Ping-Pong

- *redirection status*: this indicates whether the receiver successfully redirected the last message. It is transfered to the sender "for free", i.e. an additional send is not needed. This data is included in the message the receiver uses to update the sender's head index. It is used by the sender to increment or decrement the delay threshold associated with the socket.
- *stop-delay*: it is possible that the sender ends up spinning when receiver has already posted the redirection request. If this happens, the receiver sends a message to stop sender from delaying.

In the absence of a stop-delay message, the sender will delay according to the current threshold.

7.2 Sockets Benchmarks

We measured the performance of our sockets implementation using micro-benchmarks that simulate two common socket communication patterns. All results were obtained using a 128 KBytes socket buffer. In addition to the micro-benchmarks, we measured the performance of our system using `ttcp`, a public domain sockets benchmark. Together with our results we present numbers obtained using the Myrinet TCP implmentation. These numbers are presented only for reference, as Myrinet supports TCP in the kernel whereas our protocol is implemented at user-level.

7.2.1 Micro-benchmarks

The bidirectional ping-pong test results are shown in Figure 5. We see the significant benefit of redirection since this communication pattern allows for full redirection of messages. For 128 KBytes message transfers, we obtained 78.8 MBytes/s with redirection, but only 34 MBytes/s without. Also, ASD neither helps nor hurts performance of this test.

Using the bidirectional ping-pong test we also measured the small message latency, given in Figure 6. VMMC sockets achieve a latency of 20 μs for short messages. Myrinet's implementation is an order of magnitude slower, primarily because they use interrupts and system calls.

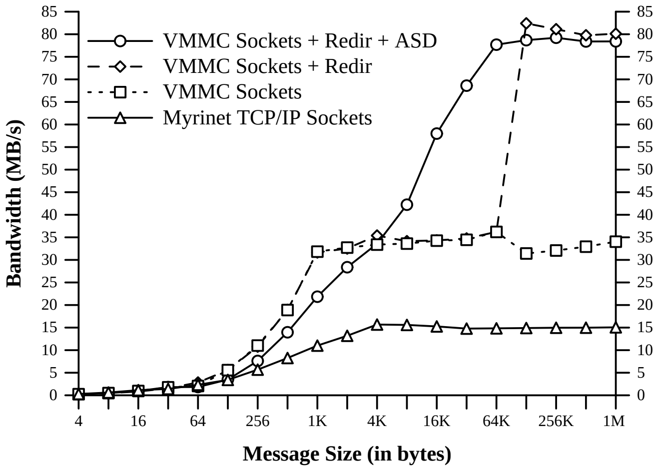


Figure 7: `ttcp`, Bandwidth numbers

7.2.2 `ttcp`

We also measured the performance of our implementation using `ttcp` version 1.12, a public domain benchmark originally written at the Army Ballistics Research Lab. `ttcp` measures network performance using a one-way communication test where the sender continuously pumps data to the receiver.

Figure 7 illustrates the performance for `ttcp`. VMMC sockets without redirection obtains a peak bandwidth of 38 MBytes/s. Adding redirection alone, does not improve performance until the message size matches the socket buffer size, at that point (128 KBytes) the socket's buffer-flow-control delays the the sender between sends. This allows the receiver to redirect incoming message which results in a peek bandwidth of 84 MBytes/s. Using redirection and ASD, we achieve superior performance for messages larger than 4 KBytes. For smaller messages, the overhead added by ASD negatively impacts overall performance. The simple solution is to disable ASD for messages smaller than 4 KBytes. Finally, `ttcp` measured a bandwidth of 15.7 MBytes/s using Myrinet-TCP/IP sockets.

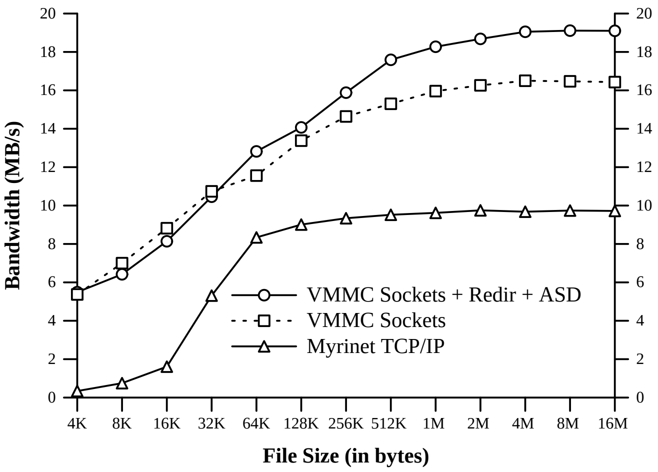


Figure 8: `tftp`, Bandwidth of `get` operation

7.3 Sockets Applications

7.3.1 `tftp`

`tftp` (Trivial File Transfer Protocol) implements only part of the functionality provided by FTP. `tftp` consists of a simple daemon and client. We run `tftp` on top of both Myrinet-TCP and *VMMC-2* sockets.

All calls to send and receive data over the network are synchronous. However, there is some pipelining between the sender, the network, and the receiver, since transfers are in progress at the same time in these stages of the data path. We set the size of the basic transfer unit to the value that gives the best results for each communication subsystem, 4 KBytes for Myrinet TCP and 64 KBytes for *VMMC-2* sockets. In our experiments we use the standard Linux file system with local disks. File system operations (reads, writes) are cached.

Figure 8 shows the `get` bandwidth for files of different sizes. The slowest phase in the pipeline is the write system call which writes a received block to the file cache. We measured the bandwidth of this operation to be 25 MBytes/s, so even with infinitely fast network the bandwidth of `tftp` would be limited to this number. The maximum bandwidth achieved by *VMMC-2* sockets is 19 MBytes/s.

7.3.2 PARFUM

The PARFUM Parallel Fault Tolerant Volume Renderer [2] is a dynamically load-balanced parallel volume renderer executing in a distributed environment. PARFUM is based on a traditional ray-casting algorithm for rendering volumetric data sets. It consists of a controller processor that implements a centralized task queue and a set of worker processors that remove tasks from the queue, process them and send the results back to the controller processor. The data set is replicated in all worker processors and is loaded at connection establishment. PARFUM is a latency sensitive application. The task size is an important parameter of the system since it determines the tradeoff between load balancing and communication. The smaller the task size the more communication incurs and the better load balance can be achieved.

Table 3 presents the elapsed time for three worker processors on Myrinet TCP and *VMMC-2* sockets for varying task sizes. We see that for this application good performance can be achieved with all communication systems, given that the task size is chosen appropriately. However, when tasks are fine grained *VMMC-2* Sockets outperform the other communication systems by far.

Task size	Myrinet TCP	*VMMC-2* Sockets
1000	9.44	8.97
250	10.79	9.16
100	14.90	9.11
50	23.27	9.32
10	120.48	10.24
5	261.92	11.76
1	1302.68	26.11

Table 3: PARFUM, Elapsed time in seconds for different numbers for 3 worker processors and different task sizes

7.3.3 Pulsar

Pulsar is a high-performance scalable parallel storage server that is implemented at user level and uses user-level communication. Pulsar avoids using the kernel for common operations, thus offering better performance than systems that use many kernel-level services.

Despite the fact that file systems are usually I/O bound, using the user-level sockets described in this paper, PULSAR's performance improves by about 200%.

8 Related Work

There is a large body of literature on improving communication performance. Most related work is in using protected, memory-mapped communication or other user-level mechanisms to support reliable, connection-oriented high-level protocols.

Spector [34] proposed a remote memory reference model to perform communication over a local area network. The implementation was programmed in a processor's microcode. This model has been revived by Thekkath et al. [35] using fast traps. They do not deal with the issue of supporting connection-oriented high-level protocols.

Druschel et al. [15] proposed the concept of application device channels which provide protected user-level access to a network interface. U-Net [3] uses a similar abstraction to support high-level protocols such as TCP/IP. Their implementation requires applications to use special, pinned buffers. Hamlyn [9] implements a sender-based communication model and uses the network interface to support user-level message passing, but it requires application programs to build headers using pinned memory. They do not address the issue of supporting connection-oriented high-level protocols.

FastSockets [32], built on top of Active Messages, achieves 30 μs one-way latency and 33 MBytes/s maximum bandwidth between two UltraSPARC 1 connected by a Myrinet network.

Fast Messages(FM) [30] is similar to AM in that each message is associated with a handler that will process the message on the receive side. FM does not allow multiple processes to share the network access. The FM sockets implementation [30] achieves one-way latency of about 34 μs and maximum bandwidth of 11 MBytes/s between two SPARCstations connected with Myrinet.

The Nectar system [11] implements TCP on their programmable network interface (16Mhz Sparc as the communication processor). Its overhead is 45 μs.

9 Conclusions

This paper describes the design, implementation, and performance of three mechanisms in the *VMMC-2* implementation: user-managed TLB for address translation, transfer redirection, and reliable data link. They are used to support protected, reliable user-level connection-oriented APIs. We implemented stream sockets using these mechanisms, and measured the performance with benchmarks and applications. Our main conclusions are that: (1) *VMMC-2* can deliver performance to the user that is quite close to the hardware limit; (2) these mechanisms support stream sockets quite well.

Transfer redirection is the key mechanism to support zero-copy connection-oriented APIs. It naturally extends the basic virtual memory-mapped communication model. The overhead of this mechanism is about 2.5 μs, compared with no redirection. Applications using stream sockets can effectively utilize the redirection mechanism. For example, over 94% of the receives in a file system using stream sockets used the redirection mechanism.

The user-managed TLB (UTLB) is an efficient mechanism to translate addresses for the network interface. It facilitates zero-copy receive and avoids data dropping caused by swapped out receive pages. A UTLB hit costs only 1.5 μs. The UTLB scheme is simple to implement, requires only device driver support and uses limited space of NI SRAM while supporting multiple processes.

We also showed that incorporating a retransmission protocol at the data link level in the network interface firmware is an effective way to deal with network failures in a system area network. This method introduces small bandwidth loss and the overhead on latency is only about 2 μs.

The overall communication performance of the *VMMC-2* implementation is quite good. Its one-way latency is 13.4 μs for a small messages and it achieves an one-way bandwidth of 80 Mbytes/s for messages of 4 Kbytes and over 90 Mbytes/s for messages larger or equal to 16 Kbytes. Stream sockets on top of *VMMC-2* also performs quite well. Its one-way latency is 20 μs and its peak bandwidth is over 84 Mbytes/s.

References

[1] R. Alpert, C. Dubnicki, E.W. Felten, and K. Li. Design and Implementation of NX Message Passing Using Shrimp Virtual Memory-Mapped Communication. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages 111–119, August 1996.

[2] J. Asplin and S. Mehus. On the Design and Performance of the PARFUM Parallel Fault Tolerant Volume Renderer. Technical Report 97-28, Univerity of Tromso, Norway, January 1997.

[3] A. Basu, Buch V, Vogels W, and von Eicken T. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, Colorado*, pages 40–53, December 1995.

[4] Anindya Basu, Matt Welsh, and Thorst von Eicken. Incorporating Memory Management into User-Level Network Interfaces. http://www2.cs.cornell.edu/U-Net/papers/unetmm.pdf, 1996.

[5] A. Bilas and E. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. *Journal of Parallel and Distributed Computing*, 14:to appear, February 1997.

[6] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

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

[8] J.C. Brustoloni and P. Steenkiste. Effects of Buffering Semantics on I/O Performance. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 1996.

[9] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 245–260, October 1996.

[10] D. R. Cheriton. The Unified Management of Memory in the V Distributed System. Technical Report STAN–CS–88–1192, Stanford University, August 1988.

[11] Eric Cooper, Peter Steenkiste, Robert Sansom, and Brian Zill. Protocol Implmentation on the Nectar Communication Processor. In *Proceedings of the ACM SIGCOMM'90 Symposium*, September 1990.

[12] D. E. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, 16(1):35–43, February 1996.

[13] C. Dalton, G.Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, 7(4):36–43, 1995.

[14] S. Damianakis, A. Bilas, C. Dubnicki, and E.W. Felten. Client-Server Computing on Shrimp. *IEEE Micro*, 17(1):8–18, February 1997.

[15] P. Druschel, B. S. Davie, and L. L. Peterson. Experiences with a High-Speed Network Adapter: A Software Perspective. In *Proceedings of SIGCOMM '94*, pages 2–13, September 1994.

[16] P. Druschel and L. L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 189–202, December 1993.

[17] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the 1997 International Parallel Processing Symposium*, pages 388–396, April 1997.

[18] C. Dubnicki, L. Iftode, E.W. Felten, and K. Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the 1996 International Parallel Processing Symposium*, pages 372–381, April 1996.

[19] T. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 256–266, May 1992.

[20] R. Gillet, M. Collins, and D. Pimm. Overview of Network Memory Channel for PCI. In *Proceedings of the IEEE COMPCON '96*, pages 244–249, 1996.

[21] L. Iftode, C. Dubnicki, E.W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, pages 14–25, February 1996.

[22] David Johnson and Willy Zwaenepoel. The Peragrine High-Performance RPC System. *Software Practice and Experience*, 23(2):201–221, February 1993.

[23] A.R. Karlin, M.S. Manasse, L. McGeoch, and S. Owicki. Competitive Randomized Algorithms for Non-Uniform Problems. In *1st Annual ACM Symposium on Discrete Algorithms*, pages 301–309, 1989.

[24] J. Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *Proceedings of the ACM Communications Architectures and Protocols Conference*, pages 259–269, September 1993.

[25] Kimberly K. Keeton, Thomas E. Anderson, and David A. Patterson. LogP: The Case for Low-Overhead Local Area Networks. In *Hot Interconnects III*, August 1995.

[26] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson, and B.L. Stumpf. The Architecture of an Integrated Local Network. *IEEE Journal on Selected Areas in Communications*, SAC-1(5), 1983.

[27] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison Wesley, 1989.

[28] J. Lumley. A High-Throughput Network Interface for A RISC Workstation. Technical Report Technical Report HPL-92-7, Hewlett-Packard Laboratories, January 1992.

[29] A.G. Nowatzyk, M.C. Browne, E.J. Kelly, and M. Perkin. S-Connect: from Networks of Workstations to Supercomputer Performance. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, pages 71–82, May 1995.

[30] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstatin Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 1997.

[31] P. Pierce. The Paragon implementation of the NX message passing interface. In *Proceedings of 9th Conference on Hypercube Concurrent Computers and Applications*, 1994.

[32] Sl H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local Area Communication With Fast Sockets. In *USENIX '97*, 1997.

[33] J.M. Smith and C.B.S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network*, 7(4):44–52, July 1993.

[34] A. Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4):260–273, April 1982.

[35] Ch. A. Thekkath and H.M. Levy. Hardware and Software Support for Efficient Exception Handling. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–121, October 1994.

[36] Ch. A. Thekkath, Thu D. Nguyen, Moy E, and D. Lazowska E. Implementing Network Protocols at User Level. In *Proceedings of the ACM SIGCOMM'93 Symposium*, September 1993.