The latency of user-to-user, kernel-to-kernel and interrupt-to-interrupt level communication

John Markus Bjørndalen[1], Otto J. Anshus[1], Brian Vinter[2], Tore Larsen[1]

[1] Department of Computer Science
University of Tromsø

[2] Department of Mathematics and Computer Science
University of Southern Denmark

Abstract

Communication performance depends on the bit rate of the networks physical channel, the type and number of routers traversed, and on the computers ability to process the incoming bits. The latter depends on factors such as the protocol stack, bus architecture and the operating system.

To see how these factors impact the roundtrip latency, we experiment with four different protocols, an optimization of one of the protocols, and with moving the communicating endpoints from user-level processes into the kernel, and then further on into the interrupt handler of two communicating computers.

We then add a processor-bound workload to the receiving computer to simulate a server which also do computations in addition to network I/O. We report how this workload affects the roundtrip latency of the different protocols and endpoints, and also how the different benchmarks influence the execution time of the workload.

The largest reduction in latency can be achieved by using less complex protocols specialized for local networks. A less complex protocol also has less variation in the roundtrip latency when the workload is running.

For small messages, interrupt-level communication and the TCP/IP based communication both increase the workload running time the least. However, the interrupt-level communication will transmit 5.3 times more messages.

1 Introduction

The importance of distributed computing is increasing as clusters of workstations or PCs becomes more and more important. Intranets are becoming important mechanisms to manage information flow in organizations. We are seeing an increasing number of services that need to perform significant computations, and the importance and role of application and compute facilities are becoming larger.

Processor, memory and bus performance, and more and more communication performance are important factors in determining the time an application needs to complete.

The communication performance depends on the bit rate of the networks physical channel and the type and number of routers traversed, and it depends on the hosts ability to process the incoming bits. The latter depends on factors like the protocol stack, the bus architecture, and the operating system. When a message arrives or departs, many activities take place including data copying, context switching, buffer management, and interrupt handling.

In particular, low latency and low perturbation of the workload is important for scientific computations, which typically combine high computational load with a need for tight, low-latency communication.

To learn more about where the overheads are located and how large they are, we have done performance measurements to find out how much the latency of sending a message between two hosts using blocking message passing improves when increasingly less complex communication protocols are used from the user level. To establish a lower bound on the achievable latency, we also measured the latency when sending a message between two operating system kernels without going to user level. We did this in two steps, first we measured the latency when running the message passing threads in the kernel, and then by running a roundtrip benchmark directly from the interrupt handler for the network card in use. The ratios between the user level protocols and the kernel and interrupt level communication latencies will provide us with data to evaluate if there are benefits from moving, say, a distributed shared memory server closer to the network card by locating it at kernel or even interrupt level. This is work in progress.

There is overhead in allocating, copying and queueing buffers for later processing by a layer further up. If the requested service (say, send/receive operation, or read/write to distributed shared memory) takes less time than the overhead it could be better to execute the operation directly from the interrupt handler. In an earlier paper[2], we compared the performance of the read and write operations of the PastSet distributed shared memory [1][6] when using TCP/IP and two implementations of VIA (Virtual Interface Architecture). One of the VIA implementations had direct hardware support from the Gigabit network card, and the other was a software implementation, M-VIA [5], using a 100Mbit network. The results showed that both VIA implementations were significantly faster than TCP/IP. This was as expected. However, M-VIA using a 100Mbit network for small messages had comparable performance to the hardware supported VIA using a Gigabit network when blocking message passing was used between two user level processes.

We believe that the reason for this is that the majority of the overhead in both the hardware supported VIA implementation and M-VIA comes from interrupt handling and scheduling the user-level threads.

Furthermore, we observed that some of the factors contributing to the low latency of the M-VIA implementation were:

- Modified versions of the network device drivers were used which provided hooks for the M-VIA subsystem
- A localnet/Ethernet-optimized protocol was used
- Checksums were computed directly on the network interface card
- A low-overhead API with statically allocated buffers in user space was used
- Fast kernel traps were used instead of using the ordinary system call or device call mechanisms

One of the more important factors was that the implementation made use of a low-overhead Ethernet-optimized protocol. As such, it avoided a lot of the processing that internet-protocols such as TCP/IP do. However, M-VIA was also able to queue packets directly to the network interface card, and provide hooks which allows the interrupt handler to dispatch incoming VIA packets to the M-VIA system instead of sending them to the ordinary Ethernet layer in the kernel.

2 The Low-Latency Protocol

M-VIA uses a special kernel trap mechanism to speed up kernel access. A simpler (but slightly less efficient) way to provide a quick access to a device driver is to use the Linux /proc filesystem, which is a mechanism for exporting interfaces to device drivers (or any other Linux kernel module).

The /proc filesystem is typically used to export interface files which can be used for tasks such as inspection of devices (by reading files) and setting of options (by writing to specific files). To provide such interfaces, a device driver exports a file in a subdirectory of /proc and binds functions of its own choice to the read and write operations of those files.

To implement the Low Latency Protocol (LLP), we use the /proc interface to provide four different interfaces (described below) for sending and receiving Ethernet frames with a protocol ID which separate them from other protocols transmitted on the cable. The interfaces were exported by an extension to the device driver for the 100Mbit Ethernet cards we used (TrendNet NICs with DEC Tulip based chipsets).

The LLP protocol uses raw Ethernet frames with only a few addressing headers added.

Two of the interfaces were used to control the kernel-based and interrupt handler based benchmarks.

The four interfaces are:

basic This interface allows a user level process to read and write raw Ethernet frames directly to and from the network card using only a simple packet processing layer in the driver.

When a user writes a packet to the interface file, an sk_buf (a network buffer) is allocated, the packet is copied into the buffer and the buffer is queued directly with the device driver.

We used a unique ethernet protocol ID, which allowed us to recognize our own protocol directly in the device drivers interrupt handler. This was used to dispatch incoming packets to the LLP subsystem in the interrupt handler instead of going through the Ethernet packet processing in the Linux kernel.

Incoming packets are written to a buffer in the LLP subsystem. If a user-level process is blocked waiting for a packet, the process is unblocked.

basic static-skb Allocating and freeing network buffers introduces overhead in the protocol. Instead of dynamically allocating buffers for each packet, we statically allocate an sk_buf when the interface file is opened.

This buffer is reused every time the client wants to send a packet.

The interface is identical to the one above, and uses the same read routine.

kernel-based roundtrip To measure the roundtrip latency between two kernel level processes, we provide an interface file where the write operation starts a benchmark loop in the kernel.

An iteration count is provided when writing to the file, and timing data is written back to the client by modifying the write buffer.

The remote end of the benchmark starts by invoking a read operation on the corresponding interface file on that host.

The benchmark loop is run by the Linux user level processes which invoked the write and read operations. Thus, the benchmark is not run by spawning Linux kernel threads.

interupt-handler roundtrip When a packet arrives from the network for the kernel level server, the interrupt handler in the device driver still needs to queue up (or register) the packet and wake up the server thread in the benchmark.

This involves some synchronization between the interrupt handler and kernel process, as well as scheduling and running the server thread.

To avoid this, we allowed the interrupt handler to process the packet directly and queue up a reply packet for transmission with the NIC. This allows us to keep the bechmark entirely in the interrupt handlers.

To start the interrupt handler benchmark, a write operation is invoked on the corresponding interface file. An iteration count and a timestamp is stored in the packet, and the packet is immediately queued with the device driver.

The iteration count is decreased every time the packet passes trough an interrupt handler. A second timestamp is stored in the packet when the iteration count reaches 0, and the packet is returned up to the thread which invoked the write operation.

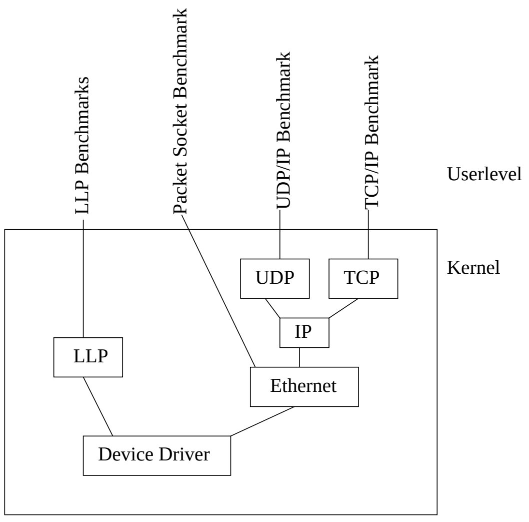

Figure 1. Protocols used in benchmarks, and location of protocol implementations.

3 Experiments

We have measured the round-trip latency of communication between two user-level processes, each on a different computer, using four different protocols. For the LLP protocol, we also measured the round-trip latency between two threads located in the operating system kernel, each on a different computer. Finally, we measured the round-trip latency of communication between two interrupt handler device drivers, each on a different computer.

The experiments measure the time it takes sending a message from one computer to another, sending a reply back, and receiving it on the original computer. We call this the round-trip latency.

For each experiment, we measure the time it took to send and receive 1500 round-trip messages, and compute the average round-trip latency. We then repeat each experiment five times, and compute the minimum, maximum, and average of the averages.

The following experiments were run:

- TCP/IP protocol, user-level to userlevel

- UDP/IP protocol, user-level to user-level
- Packet socket, transmitting raw Ethernet frames, user-level to user-level
- LLP, using the /proc filesystem, user-level to user-level
- LLP, using the /proc filesystem, statically allocated network buffers, userspace to userspace
- LLP, kernel-level to kernel-level
- LLP, interrupt handler to interrupt handler

The whole set of experiments were repeated with a work load on the receiving computer. This scenario emulates a typical client/server model where a client requests a service from a server with has other work loads than servicing remote requests. We measured both the round-trip latencies and the running time of the work load in each case. As a baseline we measured the running time of the work load without any communication.

As the work load we used a tiled matrix multiplication with good cache utilization.

The experiments were run on two HP Netserver LX Pros (4-Way 166 MHz Pentium Pros with 128 MB RAM), with Trendnet TE100-PCIA 100MBit network cards (DEC Tulip chipsets) connected to a hub. The computers were run with uniprocessor Linux kernels to simplify the experiments.

All of the protocols in the experiments are implemented at kernel level. Figure 1 shows the protocols and some of the subsystems used in the various experiments. The TCP/IP and UDP/IP benchmarks use blocking send and receive on standard sockets. The packet socket benchmark uses the PF_PACKET socket type to send and receive raw Ethernet packets on the network. Even if this saves us from processing the packet through the IP and UDP/TCP layers, the Ethernet frames are still processed by a packet processing layer in the kernel. This introduces extra processing overhead which, for instance, the M-VIA[5] implementation of the VIA communication API avoids.

To avoid perturbations with IP (and other protocols), we used Ethernet protocol ID 0x6008, which is, as far as we know, not allocated. This means that we will only receive traffic meant for this benchmark on the sockets.

4 Results

4.1 Roundtrip latency

Benchmark	min	max	avg
TCP/IP, userlevel	187	189	188
UDP/IP, userlevel	141	142	141
Packet socket, userlevel	114	115	115
LLP userlevel	87	88	88
LLP userlevel static buf	82	83	82
LLP kernel level	67	67	67
LLP interrupt handler	54	54	54

Table 1. Roundtrip latency in microseconds, 32 byte message, without workload

In table 1, we show the roundtrip latency for each benchmark with a message size of 32 bytes. The computers have no workload apart from the benchmarks. We have added a divider in the table to make it easier to see where we introduce kernel level benchmarks.

Most of the latency reduction (106 microseconds) comes from choosing a simpler protocol. Moving the benchmark into the interrupt handlers only reduce the latency by another 28 microseconds compared to the best user-level protocol.

Benchmark	min	max	avg
TCP/IP, userlevel	231	441	286
UDP/IP, userlevel	145	291	202
Packet socket, userlevel	118	178	140
LLP userlevel	90	130	106
LLP userlevel static buf	84	131	102
LLP kernel level	68	69	68
LLP interrupt handler	54	54	54

Table 2. Roundtrip latency in microseconds, 32 byte message, with workload

Table 2 shows the roundtrip latency for the benchmarks when we add a matrix multiplication workload to the receiving computer.

The less complex user-level protocols are less influenced by the workload than the more complex protocols. The kernel-level benchmark is hardly influenced by the workload, while the interrupt-level benchmark is not influenced by the workload at all.

This is interesting, since the kernel-level benchmark is implemented with user-level threads that enter the kernel and run the benchmark code. We assume that the kernel-level benchmark is scheduled the same way as any other userlevel process.

Benchmark	min	max	avg
TCP/IP, userlevel	384	385	384
UDP/IP, userlevel	339	340	339
Packet socket, userlevel	311	312	311
LLP userlevel	283	286	284
LLP userlevel static buf	278	278	278
LLP kernel level	251	253	252
LLP interrupt handler	215	215	215

Table 3. Roundtrip latency in microseconds, 1024 byte message, without workload

Table 3 shows the benchmarks with no workload on any of the hosts, and with the message size increased to 1KB.

Compared to the latencies in table 1, the increase in average latency is 196 microseconds for all user-level protocols, except from the UDP benchmark which has an increase of 198 microseconds.

The amount of processing per packet for any one of the protocols should be the same both for 32-byte and a 1024-byte message since both packet sizes fit within an Ethernet frame. Thus, the extra overhead of sending a 1024-byte message should depend on the extra time spent in the network cable, in the network adapter, time spent being copied between the network adapter to the host memory as well as time spent copying the packet within the host memory.

Observing that the added overhead for a 1KB packet compared to a 32 byte packet is nearly constant for all user-level protocols suggest that the attention paid in the TCP/IP and UDP/IP stacks to avoid extra copying of data packets has paid off and brought the number of copies down the same number as the other userlevel protocols.

For the kernel-level benchmark, the difference between the latency for 32-byte and 1024-byte payloads is further reduced to 185 microseconds.

A roundtrip message between two processes at user-level is copied 4 times between user-level and kernel-level per roundtrip. We have measured that copying a 1KB buffer takes about 2.6 microseconds on the computers we used. This explains most of the extra overhead for the user-level protocols compared to the kernel-level protocol.

For the interrupt-handler roundtrip messages, the difference between 32-byte and 1KB packets is even less at 161 microseconds. Apart from the copying done by the Ethernet card over the PCI bus to and from memory, the interrupt handler benchmark does not copy the contents of the message. Instead, it modifies the headers of the incoming message and inserts the outgoing message directly in the output queue for the network card.

Benchmark	min	max	avg
TCP/IP, userlevel	387	467	427
UDP/IP, userlevel	423	921	598
Packet socket, userlevel	316	396	346
LLP userlevel	287	288	287
LLP userlevel static buf	281	287	282
LLP kernel level	254	255	255
LLP interrupt handler	215	215	215

Table 4. Roundtrip latency in microseconds, 1024 byte payload, with workload

Table 4 shows the roundtrip latency of 1KB messages when the receiving computer runs the the matrix multiplication workload.

As in table 2, we see that the less complex protocols are less influenced by the workload than the more complex protocols.

4.2 Implications of the roundtrip benchmarks on the workload

When no benchmarks are run, the average execution time for multiplying two 512 by 512 matrices is 13.4 seconds. In tables 5 and 6, we show the impact on the execution time of the matrix multiplication when running the benchmarks. We also show the average number of roundtrip messages per second while the benchmarks execute.

Table 5 shows that the benchmarks increase the execution time of the matrix multiplication by 1.6 to 2.3 times.

The benchmark which influences the matrix multiplication the most is the packet socket benchmark. We currently do not have an explanation why this benchmark performs as bad as it does, but observe that the benchmark sends more than twice as many roundtrip messages than the TCP/IP benchmark. A likely place to look for an explanation is the amount of code executed by the protocol combined with the time that the matrix multiplication is allowed to compute before being interrupted by another packet from the client.

The two benchmarks which influence the matrix multiplication the least are the TCP/IP benchmark and the interrupt handler benchmark.

We believe that one of the reasons the workload is not influenced more by the TCP/IP benchmark is that the higher overhead in the TCP/IP protocols means that it takes longer for the client host to send the next message to the host with the workload. The matrix multiplication application is thus allowed to compute more before being interrupted by another request from the client end.

Another observation is that there is a factor 5.3 difference in the number of roundtrip messages between the two best benchmarks in this experiment, implying that a server in the kernel could support a much higher network load without disturbing the workload on a host more than a TCP/IP server which serves a smaller load.

Table 6 shows us a similar pattern to table 5. The packet socket benchmark still comes out worst when comparing the influence on the workload execution time, while the least disturbing benchmarks are the TCP/IP and interrupt handler benchmarks.

5 Related Works

In [4] it was documented that CPU performance is not the only factor affecting network throughput. Even though we have not used different computers, we have also shown that the network latency is dependent upon several other factors than CPU performance, and we have detailed some of these.

In [3] it is shown how the cost of user-level communication can be reduced by reducing the cost of servicing interrupts,

Benchmark	min	max	avg	messages/s
TCP/IP, userlevel	19	23	21	3494
UDP/IP, userlevel	23	26	24	4950
Packet socket, userlevel	25	37	30	7120
LLP userlevel	23	27	26	9406
LLP userlevel static buf	22	27	24	9801
LLP kernel level	23	24	24	14621
LLP interrupt handler	21	21	21	18501

Table 5. Impact of the benchmarks on the work load execution time, 32 bytes message

Benchmark	min	max	avg	messages/s
TCP/IP, userlevel	16	16	16	2341
UDP/IP, userlevel	16	16	16	1671
Packet socket, userlevel	20	20	20	2893
LLP userlevel	17	18	18	3482
LLP userlevel static buf	18	18	18	3540
LLP kernel level	16	16	16	3923
LLP interrupt handler	15	15	15	4642

Table 6. Impact of roundtrip benchmarks on matrix multiplication execution time, 1024 bytes message

and by controlling when the system uses interrupts and when it uses polling. They showed that blocking communication was about an order of magnitude more expensive than spinning between two computers in the Shrimp multicomputer.

Active messages [7] invoke a receiver-side handler whenever a message arrives. Control information at the head of each message species the address of a user-level routine that is responsible for extracting the message from the network. This approach often requires servicing an interrupt for each message received.

6 Conclusion

We have observed that the largest impact on roundtrip latency is the complexity and implementation of the protocol used, not whether the endpoints of the communication are in userspace or in kernel space. Reducing the overhead by choosing simpler protocols also resulted in less variation in the roundtrip latency.

This suggests that a significant reduction in communication latency can be achieved without modifications to the operating system kernel. However, to achieve the best possible latency, a combination of interrupt level handling of the communication with an efficient activation of user-level processes are necessary. The interrupt level latency effectively gives a practical lower bound on how low the latency can be.

Moving data between kernel and user-level by copying also introduces extra overhead which is visible in the roundtrip latency. This suggests that we can benefit from moving communication handling to kernel space if the data can be stored in the kernel[1]. This can be of use for distributed

[1]The extra copying could also be avoided if we used shared buffers between kernel and userspace, as VIA does

shared memory systems using servers on each computer.

The interrupt-level benchmark which had the lowest latency, was least influenced by the workload (not visible in the tables). This benchmark also disturbed the workload the least. Again, this suggests that there may be performance benefits from integrating, say, a distributed shared memory server with the interrupt handler.

7 Acknowledgements

We thank Lars Ailo Bongo for reading the paper and suggesting improvements.

References

[1] ANSHUS, O. J., AND LARSEN, T. Macroscope: The abstractions of a distributed operating system. *Norsk Informatikk Konferanse* (October 1992).

[2] BJØRNDALEN, J. M., ANSHUS, O., VINTER, B., AND LARSEN, T. Comparing the performance of the pastset distributed shared memory system using tcp/ip and m-via. In *Proceedings of WSDSM'00, Santa Fe, New Mexico* (May 2000).

[3] DAMIANAKIS, S. N., CHEN, Y., AND FELTEN, E. Reducing waiting costs in user-level communication. In *11th International Parallel Processing Symposium (IPPS '97)* (April 1997).

[4] MALY, K. J., GUPTA, A. K., AND MYNAM, S. Btu: A host communication benchmark. Computer, pp. 66-74.

[5] http://www.nersc.gov/research/ftg/via/.

[6] VINTER, B. *PastSet a Structured Distributed Shared Memory System.* PhD thesis, Tromsø University, 1999.

[7] VON EICKEN, T., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. Active messages: A mechanism for integrated communication and computation. In Proceedings of 19th International Symposium on Computer Architecture, pages 256-266, May 1992.