RZ 3520 (# 99304) 08/11/2003
Computer Science 10 pages

Research Report

The Performance of Software Multicast-Reflector Implementations for Multi-Player Online Games

Daniel Bauer and Sean Rooney

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

{dnb,sro}@zurich.ibm.com

The Performance of Software Multicast-Reflector Implementations for Multi-Player Online Games

Daniel Bauer and Sean Rooney

<dnb,sro>@zurich.ibm.com
IBM Research, Zurich Laboratory
Säumerstrasse 4
8803 Rüschlikon, Switzerland

Abstract. Massive multi-player online games are large distributed applications where thousands of participants exchange data. Existing solutions based on central servers face scalability problems. We study a hybrid solution between the peer-to-peer and central server models that divides a large game into several federated small games. The central component of this architecture is a multicast reflector. We present two efficient software implementations that have been developed as Linux kernel extensions and compare them with our user-space implementation. The comparison is based on performance measurements done on actual implementations.

1 Introduction

The term Massive Multi-Player Online Game (MMPOG) has come to denote games that have a large number of participants and are played over the Internet. Current MMPOGs use a central server approach. The number of simultaneous participants a game can host is dependent on the amount of resources available on the server farm, in particular the server I/O, network bandwidth, processing cycles and memory. Abdelkhalek et al. [1] report that processor cycles are the main bottleneck and that network bandwidth is less of a concern. Furthermore, the study of Abdelkhalek et al. revealed that compute time is equally divided between game logic and network protocol stack processing.

Unlike web servers, game servers are proprietary and dedicated to one specific game, e.g. Sony's EverQuest. Game providers cannot know in advance the number of participants that their game will attract meaning that it is difficult for them to dimension correctly their server farms a priori, while the proprietary nature of the games means that it is not easy to simply rent additional resources dynamically.

A better solution is one in which the total available computation to the game increases with the number of participants. Peer-to-peer games have this property; however, they only work for a small number of participants — as the number of packets sent grows as a square of the number of participants — or require communication over a broadcast medium, e.g. a LAN. It is also difficult to envisage how a pure peer-to-peer system can be made reliable and secure.

Our approach is to use a hybrid of the peer-to-peer and central-server models to obtain scalable but reliable MMPOGs. We achieve this by

- dividing a large game into several federated small games, each of a size that can be handled using a peer-to-peer model;
- using fast dedicated multicast reflectors that handle the communication for one or several of the small games;
- separating the control and data planes, such that the multicast reflectors are responsible for fast data forwarding, while servers and clients continue to perform the control and management functions. This point is not discussed further in this paper.

In this paper we consider the means by which the multicast reflector is implemented. There are two alternatives a pure or programmable hardware approach in which the multicast reflector is instrumented using a Network Processor (NP) or an ASIC and a approach in which it is implemented entirely in software either in user-space or as an extension to the OS's kernel. The hardware approach is efficient but inflexible. The software approach is more flexible and does not require specialized hardware but performance is a concern. This paper reports the performance of some alternative software implementations and outlines the characteristics of the games for which the simple software approach is adequate.

2 Related Work

The work described in the overview paper of military simulation [5] is similar in nature to the architecture described here. For example, in the Navel Postgraduate School Net (NPSNET), the space is divided into cells in which information is multicast. One member of the cell is responsible for adding and removing others as well as giving a new member the current state of the cell. The difference between military simulations and games is rather of context than form; simulations are performed in a highly controlled network where the participants in the simulations and their activities are known in advance. Consequently the simulation designers can design the infrastructure with this in mind. There are more unknowns in running commercial games over the public Internet; for example, no player can be trusted to be a group leader as in NPSNET. Exact details of the architecture, for example how multicast is supported are not in the public domain.

In the commercial games space OpenSkies supports large games using a federation of multicast entities [6]. In contrast to the multicast reflector described here, they use a purely user-space implementation for group communication. A network stress test [7] on computers with a 500 Mhz CPU revealed that this implementation saturates when forwarding at a rate of less than 20 Mb/s.

The Mercury system is an example of a research game architecture based on a publish-subscribe mechanism [2]. While the approach creates an evenly balanced system, the simulation results also show that the delay scales linearly with the number of nodes and therefore limits scalability.

3 Overview of the Federated Game Architecture

Our approach divides a large game into multiple federations of smaller games. The participants of each of the smaller games belong to the same multicast group and constantly exchange data. Clients use their game-state to decide which multicast group they register in. This decision function is, in general, game specific. A simple decision function is to map the virtual location of the game characters to a multicast group. At the start of a game session, a client obtains the mapping between game's virtual locations and multicast groups from the control server. A multicast group is given by the IP address and port number of a multicast reflector. The client then adds itself to one or more of the multicast groups corresponding to these virtual locations by sending a join message to the corresponding multicast reflector.

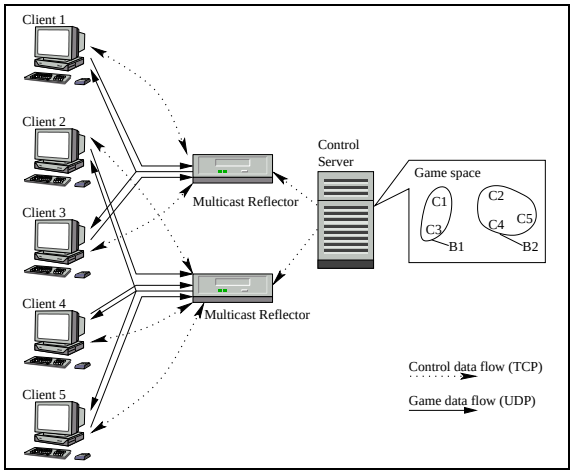


Fig. 1. Hybrid Game Architecture

As the players move between distinct virtual locations in the game, the client removes itself from the old multicast group and adds itself to the new one. The multicast reflector periodical checkpoints the current game state by requesting it from a client. As in principle all clients have the same game state, it should not matter which one. The multicast reflector may choose an arbitrary client or the one that as measured by delay is the closest, or it may use other criteria. Clients entering a new location obtain the current state of the game at that location. In addition, clients whose state has become desynchronized from the game because of packet loss may use the check-pointed state to resynchronize. In order to ensure a quick transition between virtual locations the application software writer may choose to be a member of not only the virtual location at which the client is resident but also of all those around it. The architecture is shown in Figure 1.

A multicast reflector maintains the list of clients resident at a given virtual location. The client sends data packets to the multicast reflector, which then forwards them to all co-located clients. The client does not know the addresses of the other clients, nor does it communicate with them directly. Multicast reflectors may all be located at the providers server farm, or scattered throughout the network. Whereas the first scenario is easier to install and manage, distributing multicast reflectors throughout the network allow better scalability — because not all data traffic is channeled through the same network access point — and is more resilient.

The multicast reflector implements the multicast function without requiring the network to support IP multicast. The clients send unicast UDP packets to the addressable multicast reflector, which in turn generates many unicast UDP packets to send to the members of the peer group. The multicast reflector in effect acts as a broadcast domain for the virtual location. A single multicast reflector supports multiple groups simultaneously, the groups are addressed using UDP port numbers.

4 Multicast Reflector Implementation

A key feature of the multicast reflector is the ability to send UDP datagrams to a set of receivers very efficiently. Efficiency is measured by both a high throughput and a low CPU load. In this section, three different multicast reflector implementations on Linux are described. As an introduction to these implementations, we provide a brief, schematic overview of the Linux network stack for UDP. We assume that packets are sent on an Ethernet device.

Figure 2 shows the different layers of the Linux network stack. UDP and IP are shown as a single layer as they are rather closely integrated. If a UDP datagram is sent, the following steps are carried out:

1. The application writes the datagram on the socket.
2. The UDP layer checks the validity of the parameters. It then creates both the UDP header and a pseudo UDP header [9]. A routing table lookup is also done to obtain the IP source address needed for the pseudo UDP header.
3. On the IP layer, a socket buffer is created and the IP header is filled in. The payload is then copied from the user space into the socket buffer[1]. The checksum is computed while copying, and therefore does not consume many additional CPU cycles. Before forwarding the packet to the device level, the MAC header is filled in.
4. On the device level, the packet is queued for transmission. Linux maintains a transmit queue for each physical device. Queuing discipline and scheduling are configurable; the default is a FIFO queue with tail dropping. If the queue is full, an error message is returned to the IP layer. Otherwise, the packet is enqueued, and the packet scheduler is started. The scheduler checks whether the device is accepting packets for transmission. If not, the scheduler stops and will be restarted as soon as the device is ready again. This is signaled through an interrupt. Scheduled packets are handed over to the device driver's transmission routine.
5. The device driver copies the contents of the packet to the buffer on the device. This is typically done using DMA transfer. If the device supports scatter/gather, then the packet does not have to be available in a single linear buffer but can be scattered across multiple buffers.

4.1 User-Space Implementation

The user-space implementation does not require any changes in the kernel. It uses the BSD socket API for sending UDP datagrams. The multicast group members are maintained in a list of IP address/port tuples. Multicasting a datagram is done by traversing the list and sending the datagram to each group member.

While this approach is very simple, easy and portable, it is not optimum. For each destination address, the payload data is copied from the user space to the kernel space even though the payload itself does not change. The other problem, which is more severe, is the fact that the device queue (see Figure 2) is very easily overflowed even on Gigabit Ethernet devices, as modern systems have memory transfer rates higher than 30 Gb/s [3]. The bottleneck is either the device or the PCI bus. Consequently packets are dropped, and the kernel returns the error `ENOBUFS` to the application[2]. The application has no other

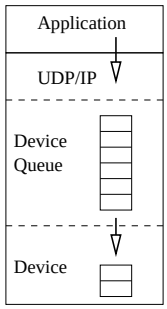


Fig. 2. Linux Network Stack Schematic Overview

[1] This copy function is part of the UDP layer but accessed by a function upcall from the IP layer.

[2] Per default, UDP datagrams are silently dropped and no error message is returned. Setting the `IP_RECVERR` option on the socket enables extended error messages, including ICMP error messages.

possibility than to resend the dropped packet. As the `ENOBUFS` error also indicates that the device queue is full, the application could sleep for some period of time before resending the packet, allowing the queue to empty itself. Unfortunately, the time granularity of Linux on Intel processors is 10 ms, which is too coarse for this purpose. For example, a 100 Mb/s fast Ethernet device sends 125 KB data in 10 ms. If we assume small packets of a few 100 B, as is typical for games, then several hundred packets can be send in 10 ms. The queue length (`txqueuelen`) is typically 100 packets; sleeping for 10 ms therefore dramatically reduces the throughput as the device will be idle for most of the time. The only viable choice for the application is to resend packets as fast as possible, even though this introduces a busy-waiting loop that consumes unnecessary CPU cycles.

4.2 Kernel-Space Implementation Using Scatter-Gather

A more efficient implementation can be achieved by extending the UDP layer in the Linux kernel. The principle is to maintain a list of address/port tuples per socket in the kernel. A datagram sent on the socket is then sent to each entry in the list.

In order to achieve a performance benefit, the payload is copied and checksummed only once per multicast operation instead of once per destination. The Linux buffer-handling routines provide mechanisms to implement such a scheme. The main data structure used for handling memory in the network layer is the `sk_buff` [4]. The `sk_buff` is a control structure that contains buffer space for holding packet data. Normally, a packet including all headers is stored in a single linear `sk_buff`. In order to handle IP fragments efficiently, the `sk_buff` contains a set of pointers to memory pages where additional packet data can be stored. These pointers are maintained in the `frag_list` array of the `sk_buff`. This structure allows the following implementation of a solution:

1. Data sent on the socket is not, as usual, copied into a `sk_buff` structure but into one or more memory pages. The memory pages contain the payload without any protocol headers. During the copy operation, the checksum is computed.
2. For each entry in the address list, a `sk_buff` structure is allocated with sufficient buffer for holding the ethernet, IP and UDP headers. Entries in the `frag_list` of the `sk_buff` are generated that reference the memory pages containing the payload. The UDP checksum can be computed efficiently based on the previously computed checksum of the payload, the source/destination IP addresses and the UDP header.
3. After the protocol headers are filled in, the `sk_buff` is enqueued in the device queue for transmission. At a given point in time, multiple `sk_buffs` that reference the same payload exist in the queue. This is shown in Figure 3. If the enqueue operation fails, then the `sk_buff` is freed by the device layer. In this case, a new `sk_buff` structure is allocated and initialized.
4. After the last `sk_buff` has been enqueued, the control of the memory pages that hold the payload are returned to the kernel's memory manager. These pages will be freed after the last `sk_buff` has been deallocated.

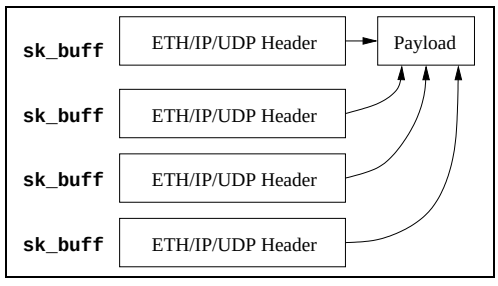


Fig. 3. Several sk_buffs sharing the same payload.

The separation of headers and payload into separate buffers is efficient for network devices that support scatter-gather I/O, i.e. devices that can copy frames from non contiguous kernel memory blocks. Most of the modern network cards support this feature.

The problem of device queue overflows can be handled more elegantly in the kernel than in user space. If a packet was dropped owing to queue overflow, then the process is suspended (put to sleep) until the queue is almost empty. If the queue is drained below a low-water mark, then all processes that have been put to sleep are woken up and continue queuing packets. This is implemented using the kernel's wait queues. We have extended each device-control structure with a wait queue. If a network device can no longer accept packets, it tells the network layer to stop any transmission. The packet scheduler then stops, and the device queue eventually fills up until it overflows. If a packet is dropped, then the sending process is put to sleep on the device's wait queue. As soon as the network device is ready for transmission

again, it raises an interrupt, and the packet scheduler is restarted. As soon as the queue is drained below the low-water mark, which is currently set to one eighth of the queue size, then all the sleeping processes are woken up and continue to queue packets.

This solution is easy to implement if all receivers can be reached through a single network device. If multiple network devices are used, unnecessary blocking on a device queue needs to be prevented. This is done by first performing a routing-table lookup for all destination addresses. The routing-table lookup provides, among other things, the outgoing device and allows the grouping of packets per device. The process is then only put to sleep if all device queues for which packets are available are full. A heuristic based on queue size and device capacity (device speed) is used to selected the device that is likely to drain the fastest. The process is then put to sleep on this device's wait queue.

4.3 Kernel-Space Implementation Using Linear Buffers

Not all devices support scatter-gather or they do not support it efficiently. For these cases, an alternate implementation using linear buffers is required. This solution works as follows:

1. Data sent on a socket is copied into a `sk_buff` structure such that there is enough headroom for all required headers. While the data is being copied, the checksum is computed.
2. For each entry in the address list, the original `sk_buff` is copied to a newly allocated `sk_buff`. The UDP header is computed using the previously computed payload checksum.
3. After the protocol headers have been filled in, the `sk_buff` is enqueued in the device queue for transmission. If the enqueue operation fails, the `sk_buff` is freed by the device layer and has to be copied again.
4. After all the `sk_buffs` have been enqueued, the original `sk_buff` is freed again.[3]

4.4 Maintaining the Address List in Kernel Space

The kernel maintains an address list per UDP socket. The list-head and other control information are stored in the socket's transport-protocol-specific fields (`union tp_pinfo`). The socket's memory footprint is not increased, as `tp_pinfo` is typically used to hold TCP-related control information that is not needed for UDP sockets.

When a UDP socket is created, the address list is empty, and the semantics of `write`, `send` and `sendto` operations are not changed. Applications administer the address list using two new socket options at the UDP level: `UDP_MCAST_ADD` is used to add a new IP address and port number to the list, whereas `UDP_MCAST_DEL` is used to remove a previously added entry.

The semantics of the `send` operation changes as soon as at least one entry exists in the address list. In this case, the datagram will be sent to all addresses in the list.

If the socket is closed, then the address list is freed again.

5 Results

In order to evaluate the different implementations, several measurements on various computers have been carried out. Two parameters have been measured: the throughput that was achieved and the CPU load that was generated. The measurements were carried out for packets of size 50, 100, 500, and 1000 bytes and for 3, 30, 60, and 90 receivers on different systems. For each parameter set, several hundred thousand packets have been sent.

Throughput has been measured at the application level by timing the `send` system calls. This also means that no protocol headers have been considered, i.e. all throughput figures report the data rate for the payload. All three implementations guarantee that no packets are lost at the sender, which was verified using a network sniffer.

[3] At first glance, this seems inefficient, as the last entry in the address list could use the original `sk_buff` rather than a copy of it. However, if the enqueue operation fails, then the `sk_buff` is freed and there is no chance to send a datagram to the last receiver.

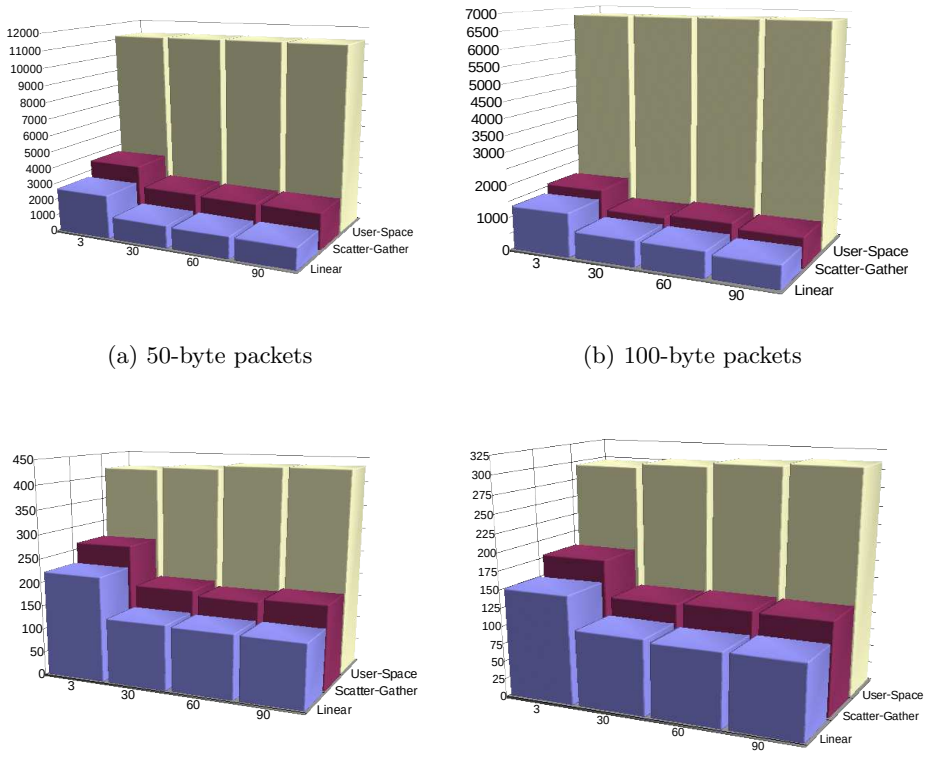
CPU load was measured using the performance-monitor counters of the processor, by means of the Linux kernel extension written by Pettersson [8]. This extension provides so-called virtual performance registers, which are performance counters on a per-process basis. The CPU load that a process generated was measured using a time-stamp counter, which measures the CPU cycles a process uses. As the result is inherently CPU dependent, it can only be used to compare different approaches executed on the same machine.

5.1 Measurement Series A

The first measurement series was carried out on a machine equipped with a 1.7 GHz Intel® Pentium® 4 processor, RAMBUS® PC800 memory subsystem and an Intel® EtherExpress PRO/100 VM fast Ethernet device. The measurements were carried out using a modified Linux kernel version 2.4.20 with the original Intel® E100 device driver version 2.2.21. The default `txqueuelen` of 100 was used.

The throughput measurements show that all three implementations achieve the same figures, independently of the number of receivers. The differences between implementations are very small and always below 1%. For 50-byte packets, all three implementations achieved 35 Mb/s net throughput. For 100-, 500- and 1000-byte packets, the results are 60, 88 and 94 Mb/s, respectively.

The situation with respect to CPU load is different. Figure 4 shows the results for different numbers of receivers and packet sizes. The ordinate gives the CPU load in millions of clock ticks. The tests were carried out on an otherwise unloaded machine.



(a) 50-byte packets

(b) 100-byte packets

(c) 500-byte packets

(d) 1000-byte packets

Fig. 4. CPU load using Intel E100

The following observations can be made:

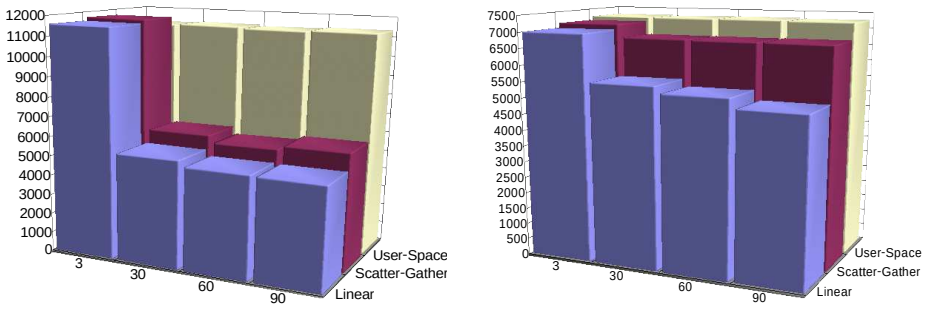
– The kernel-space implementations operate several times more efficiently than the user-space implementation. When using the E100 device, this effect is more pronounced for smaller packets.
– The efficiency of the kernel-space implementations improves with the number of receivers. This is because in the case of very few receivers, more CPU-intensive copy operations from user space to kernel space occur. Hence, referencing the payload buffer as done by "scatter-gather" or even copying within the kernel as done by "linear" is more efficient.
– A somewhat unexpected result is that the "linear" solution performs slightly better than the "scatter-gather" solution does. In the case of "linear", the payload is copied each time into a linear buffer, which is transferred in a single DMA operation from the kernel space to the card's memory. In the case of "scatter-gather", the payload is merely referenced. Two DMA transfers are required to transfer the packet to the card. The first transfer copies the Ethernet, IP and UDP header, and the second transfer copies the payload. For the EtherExpress Card, the overhead of doing two DMA operations is more expensive than copying the payload each time.

5.2 Measurement Series B

The second measurement series was carried out on the same machine as series A, but with a RealTek® 8139 fast Ethernet adapter. We used the "8139too" device driver, version 0.9.26. The rest of the software was unchanged.

The throughput figures give a slightly different picture. For the two larger packet sizes of 500 and 1000 bytes, all three implementations behave exactly the same and also achieve 88 and 94 Mb/s, respectively. For 50-byte packets, "user space" achieves 36 Mb/s, "linear" 35.5 Mb/s and "scatter-gather" 35 Mb/s. When sending 100-byte packets, "user space" manages to send 56 Mb/s, the other two achieve 60.5 Mb/s.

Figures 5 and 6 shows the CPU load of the three implementations for different packet sizes and numbers of receivers.



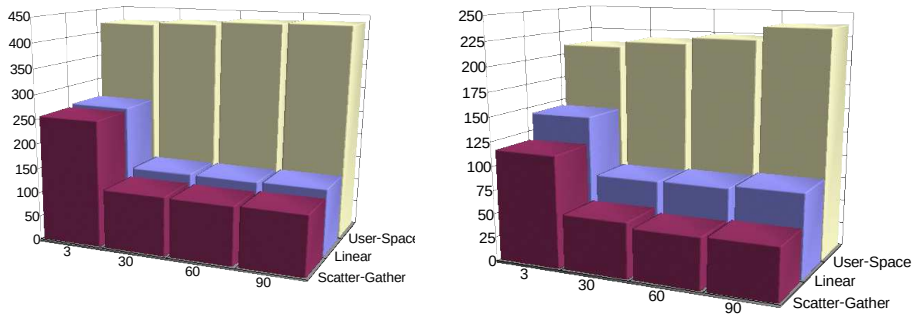
(a) 50-byte packets

(b) 100-byte packets

Fig. 5. CPU load using Realtek 8139

Again, the load measurements reveal that the kernel implementations are more efficient than the user-space implementation. An exception is the case of 50-byte packets with three receivers, where the kernel-space implementations are slightly worse than the user-space implementation. Apart from that, the following other points are noteworthy.

– For smaller packet sizes, "linear" is the most efficient of all three implementations. This changes for larger packet sizes, where "scatter-gather" is more efficient. This indicates that copy operations in the kernel are less efficient for larger packets than two DMA transfers to the Realtek 8139 network device.
– For 100-byte packets, the kernel implementations are only slightly more efficient than the user-space implementation. At the same time, the kernel implementations also achieve 8% more throughput. This



(a) 500-byte packets (b) 1000-byte packets

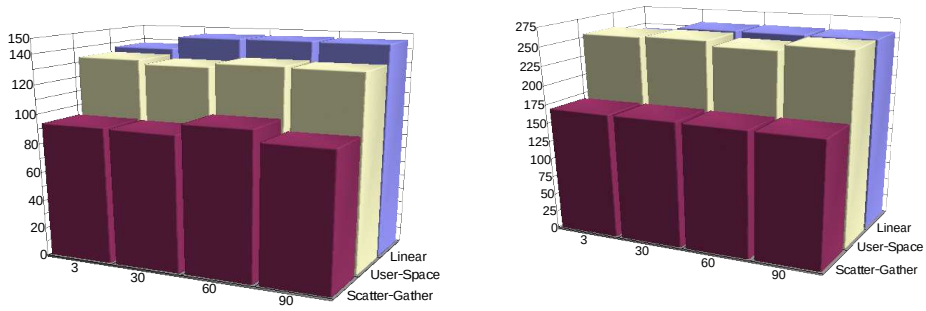
Fig. 6. CPU load using Realtek 8139

somewhat unexpected behavior seems to be a peculiarity of the RealTek device driver or the device itself, as the measurements done on the same machine with the Intel device do not show this effect.

5.3 Measurement Series C

The third measurement series was carried out on a machine equipped with a 2.66 GHz Intel® Pentium® 4 processor, PC2100 DDR memory subsystem and a Intel® PRO/1000 MT gigabit Ethernet adapter.

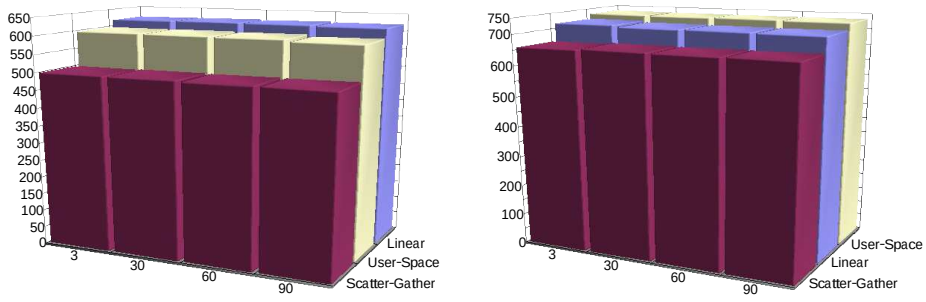
The throughput achieved by the various implementations differs rather significantly for small packets, as shown in Figure 7. "Scatter-gather" shows the weakest performance. The conclusion is that for small packets, copying the packets is more efficient than executing two DMA transfers. For larger packet sizes, "scatter-gather" performs better, as shown in Figure 8, although it does not quite achieve the performance of the other two implementations.



(a) 50-byte packets (b) 100-byte packets

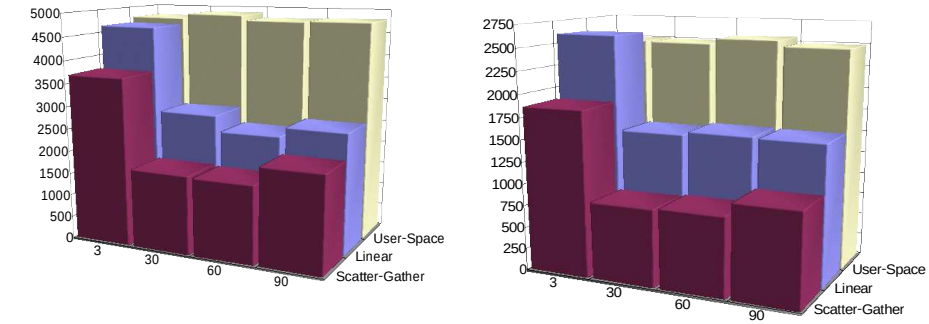
Fig. 7. Throughput in Mb/s using Intel E1000

The CPU load measurements show again that the kernel-space implementations, in general, are more efficient than the user-space implementation. Figure 9 shows the CPU load for the various parameter configurations. Figure 9(d) is rather surprising, as it shows that the user-space implementation is more efficient than "linear".

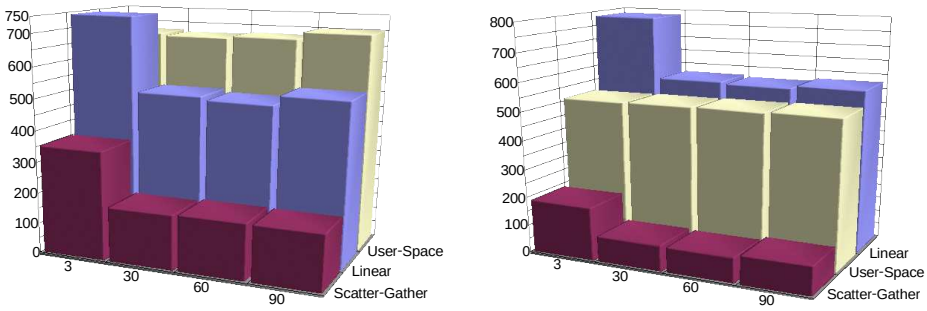


(a) 500-byte packets (b) 1000-byte packets

Fig. 8. Throughput in Mb/s using Intel E1000



(a) 50-byte packets (b) 100-byte packets



(c) 500-byte packets (d) 1000-byte packets

Fig. 9. CPU load using Intel E1000

6 Discussion of the Results

The throughput achieved by the three implementations is in the same range. Assuming a client of a typical game produces a stream of 8 Kb/s using an average packet size of 100 bytes [1], then our software multicast reflectors are able to support 7'500 clients with fast Ethernet and between 20'000 and 30'000 for Gigabit Ethernet depending on which one is used. The throughput figures are net numbers. For the gross throughput, the Ethernet, IP and UDP headers have to be taken into account. The resulting gross throughput for fast

ethernet is about 85 Mb/s and for Gigabit Ethernet, it ranges from 230 Mb/s to 360 Mb/s. This shows that a software based solution is sufficient for fast Ethernet while hardware assists could increase the throughput for Gigabit Ethernet.

The measurement results for the CPU load revealed that the kernel-space implementations are up to six times more efficient. This gain in efficiency depends on the device and device driver used as well as on the number of receivers in the multicast group. The user-space implementation consumes most of the available CPU cycles for packet forwarding. If we take into account that the multicast reflector has to execute control code for registering and unregistering participants and other administrative tasks, we expect that the throughput of the user-space implementation will be lower for real-world implementations. The kernel-space implementations, on the other hand, consume considerable less CPU resources in packet forwarding thus leaving more resources available for control tasks.

7 Conclusions

By dividing large multi-players games into a federation of many peer-to-peer systems, we scale the resources with the number of participants. The actual data forwarding is performed by multicast reflectors. In this paper, we have described and measured the performance of three different implementations of multicast reflectors on Linux. Two implementations were done in the Linux kernel space; both require only modest modifications of the network layer, essentially adding an address list and a wait queue to the kernel. Compared with a user-space implementation, the CPU load is reduced by as much as a factor of six while the throughput is maintained over a wide range of packet- and multicast group sizes. The measurements also show that the network device and device driver have a significant effect on the resulting CPU load. The low load is achieved by eliminating the busy-wait loop that is necessary in the user-space implementation. The kernel-space implementations make sure that datagrams are not lost because of overflowing queues, a task that otherwise has to be done in the application. At the same time applications are simplified as they can use a single send operation for sending datagrams to an entire multicast group.

References

[1] A. Abdelkhalek, A. Bilas, and A. Moshovos. Behavior and performance of interactive multi-player game servers. In *Proceedings of the International IEEE Symposium on the Performance Analysis of Systems and Software (ISPASS-2001)*, Nov. 2001.

[2] A. Bharambe, S. Rio, and S. Seshan. Mercury: A Scalable Publish-Subscribe System for Internet Games. In *NetGames 2002 – First Workshop on Network and System Support for Games*, Braunschweig, Germany, Apr. 2002.

[3] M. Clendenin. Dual-channel DDR chips aim for 5.3 Gbyte/s bandwith. http://www.eetimes.com/story/OEG20020809S0036, Aug. 2002. EETimes.

[4] A. Cox. Network buffers and memory management. *Linux Journal*, Sept. 1996.

[5] K. Morse. Interest management in large-scale distributed simulations. Tech report 96-27, Dept. of Information and Computer Science, University of California, Irvine, 1996.

[6] OpenSkies. OpenSkies Network Architecture. http://www.openskies.net/papers/papers.html.

[7] OpenSkies. Openskies performance test: Demonstration of scalability. http://www.openskies.net/papers/papers.html.

[8] M. Pettersson. Linux x86 Performance-Monitoring Counters Driver. http://user.it.uu.se/ mikpe/linux/perfctr, Mar. 2003.

[9] J. Postel. User datagram protocol. Request for Comments 768, Internet Engineering Task Force, Aug. 1980.