Efficient Operating System Support for Group Unicast*

Martin Karsten, Jialin Song, Michael Kwok, Tim Brecht
School of Computer Science
University of Waterloo, Canada
{mkarsten,j8song,kfkwok,brecht}@cs.uwaterloo.ca

ABSTRACT

A common requirement of many Internet services is to send exactly the same data to a number of hosts at the same time. Without IP-level multicast, this form of group communication is realized by unicasting the data to each desired host. Although this approach is portable and easy to implement, it is extremely inefficient for the sending host. In this paper, we propose a kernel-based technique to efficiently facilitate unicast send operations for group communication with only minimal additions to the sending operating system interface and implementation. We present the design and prototype implementation of our approach and experimentally demonstrate the significant performance improvements it provides. Additionally, we conduct experiments to decompose the processing costs in the network stack and show that the biggest cost reductions are not necessarily due to reduced memory copying.

Categories and Subject Descriptors

D.4.4 [**Communications Management**]: Message sending

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Group Communication, Multicast, Operating Systems

1. INTRODUCTION

Many Internet applications are faced with the task of sending the same data to a group of receivers. Because of the lack of widespread support for IP-level multicast, these applications currently have no choice but to unicast the data to each member of the group. We refer to this technique as user-level group-unicast (or simply user-groupcast) because it is implemented by the application without explicit operating system support and because at the network layer, data is actually being sent using unicast to each receiver. In this case, each send to a group member requires a separate system call and the operating system overhead incurred to send the data to every member of the group is substantial. Each system call incurs overhead due to context switching, validating the system call parameters (e.g., ensuring the data being sent is in the process' address space), and copying the data from user to kernel space. Depending on the network stack implementation additional copying may be required in the kernel before the packet is actually transferred to the network interface.

The primary application scenarios for our technique are Internet nodes that provide a centralized group communication service. Examples are servers for distributed virtual environments such as on-line games, streaming media servers, or servers for telephony services that offer conference bridge functionality. As an example, an on-line game server for a first-person shooting game is required to send updates of the virtual environment to all players in a timely manner. Recent studies report that when the server is very busy, send operations account for up to 45% of the overall processing time [1]. In addition we believe that infrastructure nodes in overlay networks can benefit from the interfaces and mechanisms we propose. Overlay networks often use UDP as the base transmission protocol. Overlay multicast is then realized by sending the same data to a group of intermediate hosts which are acting as next hops in the overlay network [17]. This further illustrates the importance of an efficient group communication.

We focus on UDP because of its widespread adoption for group communication in the Internet [2, 6, 15, 14] and because it allows us to easily illustrate how our approach can be used to improve the efficiency of group-unicast operations. UDP is an unreliable datagram protocol and is often combined with application-level mechanisms to meet application specific requirements. The targeted applications use UDP to transmit voice, video, or other real-time data. They are highly interactive, such as conferencing or online gaming, and usually transmit data in small messages to achieve a low packetization delay. They do not necessarily consume a large amount of transmission bandwidth and often, servers are the system bottleneck, rather than the network.

In this paper we describe the design, implementation, and evaluation of operating system support for efficient group-

*This work is supported in part by Hewlett Packard, the Ontario Research and Development Challenge Fund, and the Natural Sciences and Engineering Research Council of Canada.

NOSSDAV'05, June 13–14, 2005, Stevenson, Washington, USA.

unicast operations within the kernel. We refer to this approach as kernel-level group-unicast (kernel-groupcast). It greatly reduces overhead for sending operations, thereby providing the same sender with the ability to perform more computation between sends, to send more data, or to scale to significantly larger groups. We experimentally measure and analyze the system performance and come to the (somewhat surprising) conclusion that memory copy is not necessarily the dominating cost component for many applications using group communication. We conclude that the overall cost savings are really an amalgam of cost reductions in several different components.

It is important to note that our proposal is strictly confined to the implementation of group-unicast at nodes performing group send operations. It only reduces the time and processing cost of a server sending group data to clients. It requires only relatively minor additions to the server operating system interface and implementation. We do not require modifications to any protocols, network semantics or receiving hosts. On the other hand, this technique by itself does not reduce the network traffic associated with group communication because the underlying means of communication is unicasting to each member of the group.

2. RELATED WORK

Group communication requires sending information to several recipients at the same time, using either a one-to-many or a many-to-many communication model. It has become even more relevant with the increased popularity of applications such as multi-player online games (MOGs) [6], audio and video conferencing [15], and information delivery [2].

For example, in a typical MOG, players at their workstations interact with each other in a shared virtual environment. Some MOGs may involve thousands of simultaneous players. To provide the players with a consistent view of the virtual environment, any changes in a player's states (e.g., position and velocity) must be distributed to those affected in a timely manner. Efficient group communication is therefore a key requirement in MOGs. Of prime importance is the scalability of the communication mechanism.

Group communication is generally supported by multicast. IP multicast proposed by Deering [5] has long been regarded as the right mechanism. However, its deployment is still limited in today's network due to a variety of technical and non-technical reasons [7]. For instance, IP multicast requires all intermediate routers to be upgraded and to support additional tasks such as maintaining group state and routing information. This significantly increases the complexity and overhead at the routers. As a result, there have been several proposals for implementing multicast in the application-layer rather than the network-layer. This approach is known as application-layer multicast (ALM).

In ALM, multicast functionalities like data distribution and duplication are implemented at the application-layer at a number of hosts, using unicast services. Logically, these hosts form an overlay multicast network. Packets are created at a host and unicast to the users or next hosts along this network. Examples of ALM include ALMI [16], Narada [4], NICE [3], Overcast [11], and Yoid [9]. With ALM, deployment of multicast not only becomes more feasible, but it also permits higher level support for reliability, flow control, and congestion flow. Much research has focused on construction and maintenance of efficient and scalable overlay multicast networks. However, little attention has been given to improving the efficiency of operating system support for group unicast. In this paper, we investigate and propose in-kernel mechanisms for efficient group unicast to be used in ALM and other group communication applications.

In our proposed kernel-groupcast technique, data to be sent to a group is copied once from the user to the kernel space, and then re-used when possible. This avoids transferring the same data from the user to the kernel space for every send, as is required in existing user-groupcast implementations. The benefits of reducing data copying are well-known and have also been applied to network implementations. For example, zero-copy sockets [10, 13, 8] permit the sharing of user memory with the network interface, thus eliminating user to kernel data copying and reducing communication overhead.

A detailed discussion of the processing overheads of TCP, UDP, and IP can also be found in [12]. These observations indicate that checksum computation and data copying are the two dominating cost components that cause the throughput bottleneck. We are not aware of any recent studies in this field and one of our claims is that the observations from [12] are not entirely accurate anymore. First, checksum calculation can often be offloaded to network interface cards. Second, there is only limited use for UDP checksums since application semantics often mandate their own specific error correction model. For example, FreeBSD has UDP checksums disabled as default. Some simple throughput experiments have shown that the impact of UDP checksum calculation in software is less than 1-2% on modern hardware. With respect to data copying, we refer to the detailed experimental results presented in Section 4.

3. DESIGN AND IMPLEMENTATION

The goal of our operating system extensions is to enhance the functionality of sockets and the `send` system call to facilitate sending to a group of recipients with a single call.

3.1 Operating System Interface

To use kernel-groupcast, an application first creates a socket that will be used to refer to a group. A group of recipients is associated with the socket by calling `setsockopt` with the file descriptor of the socket, the `SETSENDGRP` parameter, an array containing the addresses and ports (`struct sockaddr_in`) used to reach each member of the group. Then all that is required of the application to send the same data to all members of the group is to use the `send` system call with the file descriptor associated with the group. The kernel then sends the data to each member of the group. Figure 1 shows a simple pseudo-code example of how an application could maintain and send data to a group.

3.2 Group Membership Maintenance

There are several options to establish a group address list for send operations. The first alternative would be a `sendto`-like system call, which takes the full list of group members as parameter. For large groups, this would incur significant copying costs, even if the group membership changes only rarely. Therefore, we have chosen to separate group membership maintenance from send operations. Group address lists are stored in the socket data structure, as an array of `sockaddr_in` objects. This information is completely overwritten by group change requests through the `setsockopt`

```
grp = socket(PF_INET, SOCK_DGRAM, 0);
while (!done) {
    struct sockaddr_in addrs[N];
    /* get current addresses, return group size */
    n = maintgroup(GROUP, addrs, N);
    /* set the group membership */
    setsockopt(grp, SOL_SOCKET,SO_SETSENDGRP,
        addrs, n * sizeof(struct sockaddr_in));
    /* send a message to each group member */
    bytes = send(grp, buf, bytes, 0);
}
close(grp);
```

Figure 1: Example use of kernel-groupcast

system call. In order to reduce the associated overhead for rewriting the group membership, memory re-allocation for the array of addresses takes place only when the amount of memory needs to be increased. With this mechanism, the worst-case additional cost compared to the `sendto`-like alternative is one additional `setsockopt` system call for each send operation.

Another alternative would be to add specific operations to add or delete group members, for example, using `ADDTOGROUP` and `DELFROMGROUP` operations for `setsockopt`. However, this approach would likely require a different kernel data structure to keep track of group membership, rather than a simple array. In turn, this would incur increased overhead for maintaining group membership. Clearly the costs and benefits of each approach depend on the size of the group, the frequency with which data is sent, the size of the data, the frequency of group change and the number of group members added and deleted between sends. Since these factors vary from application to application, our present interface is extremely simple and relies on the application to determine how to best support dynamic group membership. Our experiments in Section 4 (see Figure 3) show that the system-call cost for completely changing the entire group between each send operation is negligible. A more substantial exploration of the relative merits of each approach is beyond the scope of this paper and left for future research.

3.3 Operating System Kernel

Figure 2 gives an overview of the components of an operating system kernel relevant to sending UDP data. In general, carrying out a send operation consists of the following steps. After initial processing in the socket layer, the payload data is copied into a packet buffer. The UDP layer, in cooperation with the socket layer, prepends the UDP header and retrieves IP addressing information, which is then used in the IP layer to produce the complete IP packet. The packet buffer is then submitted for link layer processing and eventually DMA-transferred to the network interface card (NIC).

The basic functionality of kernel-groupcast is located in the UDP processing component. A list of receiver endpoints (address & port) is stored with the socket data structure. If this group address list is not empty, instead of just creating a single IP packet, the UDP output function loops through the list of addresses and creates multiple instances of the packet buffer. Multiple instances of the packet buffer are necessary, since lower level processing is not invoked synchronously, but rather, packet buffers may subsequently be queued in the system. Naively working with a single copy of the packet buffer would result in overwriting header data while an outgoing packet buffer is still queued for service. Ideally, packet buffer instances can be formed without copying the payload part of the packet buffer, since only header fields change between successive loop iterations. However, this depends on the detailed design of the packet buffer data structure and other considerations, such as the fragmentation strategy. It turns out that the different design philosophies of the FreeBSD and the Linux networking stacks require different approaches to attempt to avoid copying the payload while constructing packet buffer instances.

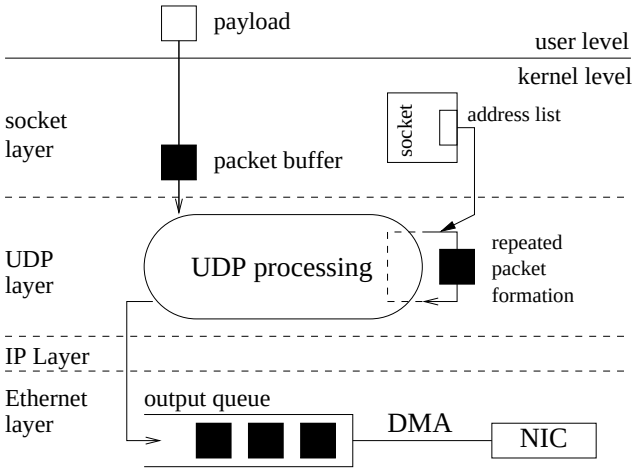

Figure 2: Layered design of UDP group unicast

3.4 Implementation

The design of the networking stack and the `mbuf` data structure in BSD-based operating systems allows for a relatively straightforward implementation of kernel-groupcast. Packets with a payload of up to 180 bytes (in our 5.2.1 version of FreeBSD) are stored in fixed-size, small `mbuf` objects and can be copied without incurring much overhead. Larger packets are split into a payload part, which is stored in an `mcluster` object, and a header part which is stored in an `mbuf` object. Constructing packets for each recipient (i.e., cloning the resulting *mbuf chain*) is possible by copying only the (relatively small) header and using a pointer with reference counting for the payload. Contemporary network interface cards can be instructed to DMA-transfer multiple memory segments which comprise a single link layer frame. This avoids overhead that might otherwise be incurred to concatenate packet segments into a contiguous buffer. IP fragmentation is done transparently by the IP layer according to the respective interface's maximum transmission unit (MTU). As a consequence of this network stack design, the FreeBSD implementation of kernel-groupcast simply loops through the address list, efficiently forms outgoing packet buffers, inserts the appropriate header information, and submits each packet buffer for further lower-level processing. The complete implementation of group unicast consists of less than 150 lines of code added to the FreeBSD kernel.

The Linux implementation of kernel-groupcast is somewhat more complicated because there is just one basic data structure for packet buffers, called `sk_buff`. There is no

separation of packet header and packet data for larger payloads, as is the case with `mbuf` and `mcluster` data structures in BSD-based systems. Therefore, it is not possible to completely avoid copying the payload when creating packets to be sent to each recipient. Essentially, the entire payload needs to be copied to avoid overwriting the header information of an `sk_buff` object waiting in a lower-level queue. Therefore, we have designed and implemented an on-demand *lazy-copy* scheme to avoid making copies until absolutely necessary.

4. EXPERIMENTS

We first present experimental results to demonstrate the efficiency gains that can be realized with kernel-groupcast. Then we conduct several alternate experiments designed to better illustrate the reasons for the dramatic performance improvements. The experimental testbed is deliberately set up to ensure that the sender machine is the system's bottleneck in our experiments. The sender machine is equipped with a 400 MHz PII CPU and two Intel dual-ported GigEthernet cards. It is connected to 4 hosts used for receiving data each of which contains a 550 MHz PIII CPU. The operating systems used for development and experiments are FreeBSD 5.2.1 and Fedora Core 2 with Linux Kernel 2.6.8.

The primary performance measure for our experiments is the average time taken by the sender to send a single request to a group of receivers (average send time). The graphs in this section show the average send time while varying the group size and/or packet lengths. A reduction in average send time indicates a corresponding increase in performance and a slope with a lower incline indicates better scaling. All experiments consist of at least 1000 send requests and are run 10 times each. We then compute 99.9% confidence intervals using the t-distribution. However, there is not much variation in many of our experiments, so not surprisingly, the confidence intervals turn out to be extremely small and in many cases are barely visible in the graphs. In these cases, we simply omit them.

4.1 Performance of Group Unicast

The first experiment illustrates the performance gains of kernel-groupcast over user-groupcast. We choose payload sizes of 100 bytes and 1000 bytes to represent small and larger UDP messages that will not be subjected to IP fragmentation. Figure 3 shows the average send time for user and kernel-groupcast send operations with these payload sizes. These results show that kernel-groupcast dramatically reduces the sending overhead when compared with user-groupcast, especially with increasing group sizes. With a UDP payload of 100 bytes, the entire packet fits into a single `mbuf` object and in the kernel-groupcast case is simply copied within the kernel when creating packets for each recipient. On the other hand, for a UDP payload size of 1000 bytes data is stored in an `mcluster` object and in-kernel copies of the payload data are avoided. In both cases kernel-groupcast works equally well and provides a significant performance enhancement by dramatically reducing the sending overhead, especially with increasing group sizes. The experiment also demonstrates that with kernel-groupcast there is no noticeable difference between sending packets of 100 or 1000 bytes. Furthermore, replacing the entire group between each successive send request does not significantly increase the performance of kernel-groupcast. This is because the

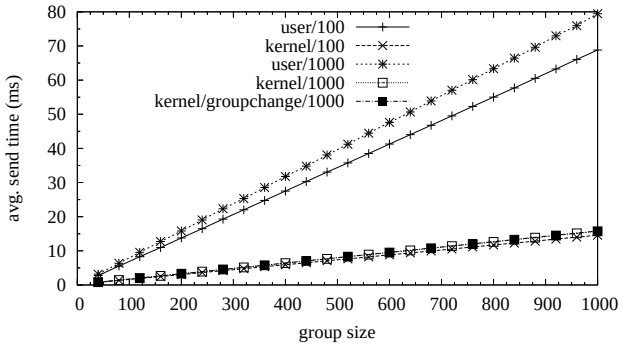

Figure 3: User- and kernel-groupcast, FreeBSD

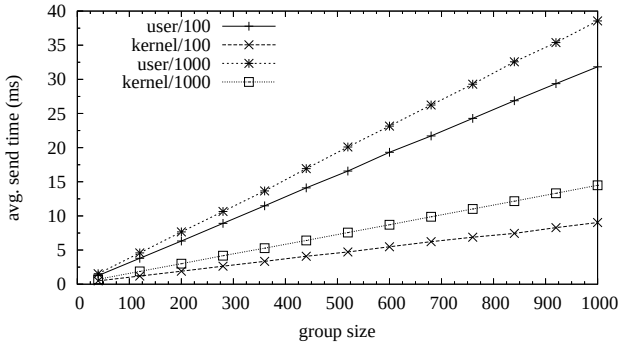

Figure 4: User- and kernel-groupcast, Linux

costs of the actual sends dominate the overhead of the extra system calls to change the group, thus demonstrating that our simple prototype interface will likely be sufficient.

Figure 4 shows a set of similar experiments conducted using a Linux sender. We observe that the basic unicast cost in Linux is much lower than in FreeBSD. Also, the kernel-groupcast performance differs for different packet sizes, likely because we cannot completely eliminate the need for in-kernel copies in Linux. It is worth noting that despite the need for lazy-copy in Linux, the kernel-groupcast implementation performs quite well when compared to user-groupcast. The number of copies observed goes up to 150 during these experiments, depending to some extent on the packet size. The larger the packets, the more copies seem necessary. Fundamentally, however, the maximum number of copies made depends on the buffering behaviour and limits of the system. We are guaranteed never to create more copies than the maximum number of entries in output queues used to move packets into the NICs. Given the difference between sending 100 bytes payload and 1000 bytes payload in combination with the fact that there is significant copying in the system, the effect of memory copies seems limited.

Although not shown in Figure 3 or Figure 4, experiments with very small group sizes down to 1 show that kernel-groupcast is always at least as efficient as regular unicast, even for a single receiver. This shows that it is safe to use this mechanism as it does not compromise performance with small packet or group sizes.

4.2 Analyzing Performance

We have applied a number of techniques to understand the

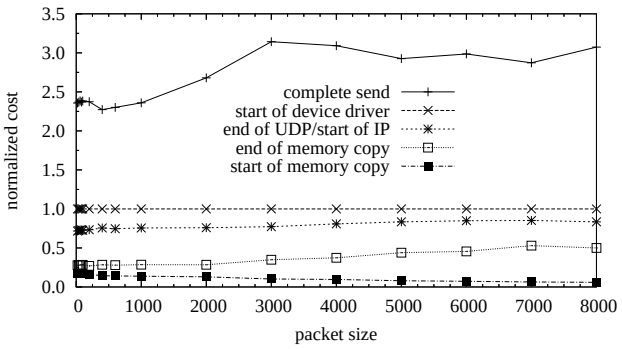

Figure 5: UDP unicast send cost, FreeBSD

composition of the significant performance improvements obtained with kernel-groupcast. We begin with Figure 5, which shows the relative breakdown of execution costs for sending a regular UDP unicast packet of different sizes in a tight user-level sending loop. We use software slicing to measure the cost of packet processing up to interesting points of execution inside the *send* system call. We divide the total per-packet cost into processing of the complete network stack (*start of device driver*) and processing in the device driver. All measurements are normalized relative to the respective cost for complete network stack processing, to distinguish between NIC/driver-independent and NIC/driver-dependent processing costs. With this normalization step, the relative contribution of the various processing steps in the network stack can be visualized and easily compared across different packet sizes.

To explain the figure we now briefly describe how to interpret the data points for a 2000 byte packet. As a percentage of the total time it takes to process the packet in the network stack, 13% of the time is spent in the socket layer getting to the point where the user-to-kernel memory copy starts. The user-to-kernel memory copy completes after 28% of the time, so 15% of the time is spent performing the copy. It takes 76% of the time to complete socket and UDP processing, so 48% of the time is spent in this processing step. After UDP processing it takes 24% of the time to execute IP and lower-level processing until reaching the device driver. Most importantly, when we switch to the scenario where the packet is actually sent (instead of being discarded after processing by the network stack) it takes 2.68 times as much time to complete the sending of the packet as it does to get through the network stack to the device driver.

From Figure 5 one can see that for packets up to 2000 bytes, the relative cost of memory copy is quite small when compared with other processing components. For example, relative to the total time spent to actually send a 2000 byte packet, the memory copy takes only 6% of the time. In fact, when considering the full cost of actually sending a UDP packet, complete network stack processing (before the device driver) accounts for less than half of the total cost.

Kernel-groupcast greatly reduces the number of context-switches and memory copies to send a packet to multiple receivers. However, given the cost decomposition shown in Figure 5, these savings are not sufficient to explain the large overall performance gains shown in Figure 3. We identify two additional cost components that are not revealed by our software slicing experiments, namely a significant reduction in the number of interrupts and improved caching behaviour.

First, with user-groupcast only a single packet is transmitted to the NIC each time. After the NIC has finished transmitting this packet, it generates an asynchronous hardware interrupt. With kernel-groupcast, multiple packets are enqueued in a tight execution loop and a single interrupt is generated after all are transmitted. We have run experiments using kernel profiling to confirm this hypothesis. The experiment results show that the number of hardware interrupts is roughly proportional to the number of actual send system calls, but independent of the group size in case of kernel-groupcast. However, we cannot yet precisely quantify the corresponding performance impact. Depending on the operating system's and device driver's interrupt coalescing strategy, the impact of hardware interrupts may vary. We also believe that such differences in the respective device drivers are responsible for the performance differences between FreeBSD's and Linux's basic unicast performance in our testbed.

Finally, kernel-groupcast seems to benefit substantially from improved cache utilization as a result of the tight low-level execution loop when sending out multiple clones of the same packet. For example, kernel profiling measurements show that invocations of the main Ethernet processing routine `ether_output()` completes roughly 10 times faster for kernel-groupcast compared to user-groupcast, despite the same number of calls and the same operations being executed in both cases. In future work we hope to work on a more precise breakdown and understanding of all sources of improvements obtained from using kernel-groupcast.

5. DISCUSSION

We now provide a simplified analysis to estimate the benefits that could be obtained from using kernel-groupcast in an example application. We consider a multi-player online game (MOG) server [1] which in a tight loop repeatedly receives data from the players, performs game related computations, and sends relevant information back to the players. Smooth video in such games requires a frame rate of 30 frames/s; therefore, each iteration of the loop has to be completed within 33.33 ms. A recent study [1] reports that for one MOG server about 45% of server execution time is devoted to user-level unicasting of messages to players.

Using the results of our user-groupcast experiments, we determine the amount of time required to send a packet to all N players in each iteration. Assuming the server is 100% utilized, the remaining time of the 33.33 ms period is then devoted to the receive and compute operations. Table 1 shows the user-groupcast send times (*send*) required in order to support groups of size N with messages of 100 bytes, obtained from the data shown in Figure 3. The remaining time is shown in the column labelled *other*. The corresponding cost of using kernel-groupcast is shown in column *send*$'$. The total cost for a round of computation and sending is then shown as *total*$'$. This can be translated into a speed-up factor of $33.33/total'$ and a theoretical improvement in the number of supported players N', both of which are shown in the table. For example, Table 1 shows that a server that is able to support 240 players using user-groupcast would be required to spend 16.5 ms of the 33.33 ms performing *send*, leaving 16.83 ms for *other* operations. However, with kernel-groupcast, the same server could support 1.63 times as many (or 390) players.

Table 1: Possible increase in supported players

	User		Kernel		Improvement	
N	*send*	*other*	$send'$	$total'$	factor	N'
40	2.78	30.55	0.75	31.30	1.06	42
120	8.26	25.07	1.90	26.97	1.24	148
240	16.50	16.83	3.64	20.47	1.63	390
360	24.76	8.57	5.36	13.93	2.39	861
480	33.06	0.27	7.10	7.37	4.52	2170

This analysis is based on the simplifying assumption that all cost components in an MOG server are constant per player and linear in the number of players in the game. Additionally, these calculations assume that the amount of data being sent to each player is independent of the number of players (although Figure 3 shows that with larger message sizes improvements from kernel-groupcast are larger). In future work we plan to examine scenarios under which these assumptions are relaxed.

Clearly, the overall performance improvement resulting from kernel-groupcast heavily depends on the relative amount of effort spent for send operations. Besides MOG servers, we envision other applications that would benefit from kernel-groupcast, as stated in Section 1.

6. SUMMARY AND CONCLUSIONS

In this paper we present the design, implementation, and evaluation of kernel-groupcast, an operating system interface and mechanism for efficient group-unicast operations. The mechanism is targeted for a specific class of applications and systems with the common characteristic of using UDP for unicast-based group communication.

We demonstrate through experiments that this relatively minor addition to the operating system kernel can improve send performance by a large factor. Any improvements obtained in applications that utilize kernel-groupcast will only apply to the group-unicast portion of the application and we show some simple analysis to demonstrate the potential impact on some example applications.

In our experimental evaluation, we make the somewhat surprising discovery that the main source of performance improvements in our kernel-groupcast implementation is not due to the reductions in memory copying overhead. This is especially true for applications that transmit data as small messages, usually to avoid a high packetization delay. In fact, we observe that the performance improvements result from savings in a number of areas including improved cache utilization as a result of the tight execution loop within the kernel, and reductions in memory copying, context switches, and the number of interrupts.

In future work, we intend to perform a more detailed breakdown of the cost components for network I/O using contemporary hardware and software. We also plan to examine the interaction between a variety of applications and the kernel-groupcast interface and implementation. Furthermore, we are investigating the benefits of applying the kernel-groupcast principles in different application scenarios using transport protocols other than UDP.

7. REFERENCES

[1] A. Abdelkhalek, A. Bilas, and A. Moshovos. Behavior and performance of interactive multi-player game servers. *Cluster Computing*, 6(4):355–366, Oct. 2003.

[2] M. Ammar, K. Almeroth, R. Clark, and Z. Fei. Multicast delivery of web pages or how to make web servers pushy. In *Proceedings of the Workshop on Internet Server Performance*, Madison, Wisconsin, June 1998.

[3] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. Technical Report UMIACS TR-2002, University of Maryland, 2002.

[4] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS*, pages 1–12, 2000.

[5] S. Deering. Multicast routing in internetworks and extended lans. In *Proceedings of ACM SIGCOMM*, pages 55–64, August 1988.

[6] C. Diot and L. Guatier. A distributed architecture for multiplayer interactive applications on the Internet. *IEEE Network*, 13(4):6–15, August 1999.

[7] C. Diot, B. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, January 2000.

[8] Z. Ditta, G. Parulkar, and J. C. Jr. The APIC approach to high performance network interface design: Protected and other techniques. In *Proceedings of IEEE INFOCOM*, volume 2, pages 7–11, April 1997.

[9] P. Francis. Yoid: Your own Internet distribution, April 2000. http://www.aciri.org/yoid.

[10] A. Gallatin, J. Chase, and K. Yocum. Trapeze/IP: TCP/IP at near-gigabit speeds. In *Proceedings of USENIX Technical Conference (Freenix Track)*, pages 109–120, June 1999.

[11] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, and J. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of Operating System Design and Implementation (OSDI)*, pages 197–212, October 2000.

[12] J. Kay and J. Pasquale. Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Transaction on Networking*, 4(6):817–828, 1996.

[13] Y. Khalidi and M. Thadani. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR95-39, Sun Microsystems Lab, May 1995.

[14] M. Macedonia, M. Zyda, D. Pratt, P. Barham, and S. Zeswitz. NPSNET: A network software architecture for large scale virtual environments. *Presence*, 3(4):265–287, 1994.

[15] S. McCanne and V. Jacobson. vic : A flexible framework for packet video. In *Proceedings of ACM Multimedia*, pages 511–522, January 1995.

[16] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 49–60, 2001.

[17] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM*, pages 73–88, August 2002.