# Group Unicast for the Real World

Elad Lahav, Martin Karsten, Tim Brecht, Weihan Wang, and Tony Zhao
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada
{elahav,mkarsten,brecht,w23wang,y8zhao}@uwaterloo.ca

## ABSTRACT

Kernel-based group unicast has been suggested as an efficient mechanism for transmitting the same data to multiple recipients. In this paper, we present a new system call, `sendgroup()`, which also supports per-recipient private data, but only uses a single in-kernel copy of the shared data. We assess the performance of the new system call using micro-benchmarks on three different operating systems. Further, we incorporate `sendgroup()` into a popular multimedia server and demonstrate an efficiency improvement of ∼45% in a representative live-broadcasting scenario. These results show that the new system call is applicable in real-world scenarios, and that its usage can lead to significant performance improvements. Moreover, we demonstrate how Amdahl's Law, when applied to the results of the micro-benchmarks, along with precise analysis of the cost of sending packets, can be used to accurately predict the impact of `sendgroup()` on this server.

## 1. INTRODUCTION

Sending the same packet to multiple recipients is a common operation in several scenarios, including live broadcasting, multiplayer online games, VoIP conferencing and more [1, 6, 17]. We refer to all recipients of some shared information as a group, and to the operation of transmitting the data as *groupcasting*.

There are several ways in which groupcasting can be implemented. The most obvious of those is standard unicasting, in which packets are sent independently to each recipient. However, such a technique wastes both local and global resources. Locally, the server needs to perform a separate system call for each recipient, with the overhead of mode switches and memory copies that are associated with such calls. Globally, unicasting results in several independent packets that are transmitted over the network and consume bandwidth.

Both of these problems are addressed by network-level multicasting [5]. In this case, each member of the group is given the same group address, and the server only needs to send one packet to the group IP. The actual work of distributing the packets is performed by multicast-aware routers that create extra packets on demand (i.e., when the next hop separates group members). Unfortunately, while these properties of multicasting make it an appealing solution, the complexity of maintaining multicast groups across networks, and especially between ISPs, has so far inhibited its wide-spread adoption [7, 21]. We consider multicast in more detail in Section 2.

We have previously proposed operating-system support for group unicast as an alternative implementation of UDP groupcasting [14]. With kernel-based group unicast, an application can use a single system call for sending packets to all group members. Though the server is still transmitting a different packet to each group member (and thus there are no bandwidth savings), this interface greatly reduces the number of system calls required for the operation (from one per group member to one per group). Furthermore, when combined with network interfaces that support scatter-gather I/O (virtually all modern server-class NICs provide this feature), this interface needs to copy the shared data only once, from user space to kernel space. By comparison, a standard unicast implementation requires a different copy of the shared data for each recipient.

This paper contains three main contributions:

1. A new API and implementation for group unicast in the form of the `sendgroup()` system call;
2. The integration of this system call with a real-world multimedia server.
3. A performance evaluation showing that micro-benchmarks can be used to predict real-world application improvements, by applying Amdahl's Law.

The new system call provides a more flexible and powerful interface for applications. This gives applications more control over group management and over the content being sent to different recipients. Specifically, the ability to add per-recipient information to the shared group data is an important extension to groupcasting, which cannot be implemented with standard IP multicast techniques. Our implementations for FreeBSD, Linux and Solaris ensure that only a single kernel-space copy of the shared data is used for the entire group.

The purpose of integrating the system call with the multimedia server is three-fold: to show that `sendgroup()` has real-world applications; to investigate the complexity of modifying existing code for using the new API; and to demonstrate the performance benefits that can be achieved by using `sendgroup()`. We conclude that while, in this instance, integration was not as straightforward as we had assumed, the improvements to the server resulting from the use of `sendgroup()` easily justify the effort.

Finally, we combine the results of the micro- and macro-benchmarks by using Amdahl's Law. The results of the micro-benchmarks are used to determine the improvement factor of


*NOSSDAV* '08 Braunschweig, Germany

`sendgroup()` over a loop of `sendmsg()` calls. Next, we combine the power of performance counters with careful analysis of the cost of `sendmsg()`, to extract the fraction of time spent on sending packets during the multimedia server experiments. These two numbers are used to predict the overall improvement factor in the macro-benchmarks, and the result is compared to the actual factor observed in those experiments. Our prediction turns out to be highly accurate, suggesting that the cost analysis, as well as the actual measurements, are correct.

## 2. RELATED WORK

The two main benefits of using the `sendgroup()` system call are reducing the number of user-kernel mode switches and avoiding redundant memory copies of the shared data. These two aspects of user-kernel interaction have been the subject of extensive research in the past, and are still believed to be important factors in the performance of I/O-intensive servers [11]. General techniques for alleviating these problems include executing user code in kernel mode (either entire applications or selected code paths) [16, 24], and applying smart schemes for memory sharing across the operating system [8, 19]. Alternatively, new system calls are often proposed to address specific scenarios. A common case is for one system call to consolidate the work of several calls with the immediate benefit of reducing the number of mode switches. For example, reading a file and sending it to a socket [9] and several other examples designed specifically to improve Web server performance [18]. Furthermore, such an approach may also open up new possibilities for improved memory usage, as an entire operation is now performed in kernel mode, without the need to communicate intermediate data through user space. For example, `sendfile()` can be used to replace a loop of `read()` and `write()` system calls. Since the calling application no longer has to control the process of transferring data from the file to the socket, the NIC can fetch this data directly from the file cache, without further copying. The system call proposed in this paper, `sendgroup()`, also takes advantage of the in-kernel implementation of groupcasting for sharing a single memory block with multiple packets.

Network-level multicast, such as IP multicast [5], is typically regarded as the right mechanism for efficient group communication. However, IP multicast is not deployed on the public Internet for a variety of reasons. IP multicast is costly, since it requires extra control overhead for group maintenance. Furthermore, multicast group addresses cannot be aggregated like unicast IP addresses, thus each group requires extra space in the forwarding table. This adds to the cost of packet forwarding in the data path. The trade-off between this extra control and forwarding overhead in comparison with the saved bandwidth and reduced sender overhead is somewhat favourable for a relatively small number of large groups, but it becomes problematic for a larger number of smaller groups. Therefore, IP multicast is sometimes deployed in enterprise networks, but not on the public Internet and this is unlikely to change any time soon. As mentioned earlier, IP multicast is also unable to handle a mixture of private and shared data.

An alternative approach to reduce the traffic and sender overhead from group communication in large-scale networks is application-level multicast (ALM). Here, packet replication is performed by conventional end systems, rather than network routers. The forwarding infrastructure is typically implemented as a user-level application process on end systems to increase robustness. Examples of ALM include ALMI [20], Narada [3], NICE [2], Overcast [13], and Yoid [10]. Since our groupcasting mechanism provides performance improvements even for very small groups (see Section 4 for details), it complements ALM approaches nicely by providing an efficient operating system interface for implementing the ALM forwarding infrastructure.

In this paper we consider a new API and implementation for group-unicast [14]. Our previous API attaches a group to a socket. A standard `sendmsg()` system call can then be used to perform a groupcast operation on that socket. However, separate `setsockopt()` calls are required for managing the group (e.g., adding and removing group members). The new interface decouples the group from the socket by using a new system call and delegates the task of group management to the application. We believe this to be both a more powerful and a more flexible solution (see Section 3).

While the FreeBSD and Solaris implementations of groupcasting have not changed since the original paper (other than being adjusted to the new API), our Linux implementation, which is now similar to that of the other operating systems, is considerably different. The previous implementation, referred to as *lazy-copy*, reuses entire socket buffers, and does so only after these buffers are freed by the network interface. This means that there is no guarantee that multiple copies of the shared data will be avoided, and that the result is highly dependent on the different timings of the processor (the producer) and the NIC (the consumer). The new implementation ensures that only a single copy of the data is used. Each packet is allocated a new socket buffer, and the shared data is only pointed to from within this buffer. Another benefit of this approach is the ability to prepend and append private data to each message, which cannot be easily done when the entire payload resides in a single, contiguous buffer.

Surprisingly, results show that the single-copy-multiple-buffer technique does not have a clear-cut performance advantage over lazy-copy. Although the new implementation is better on the mainstream hardware used in our current tests (see Section 4), the old one performs better on an old machine, with a relatively slow processor and fast network interfaces. This combination results in most of the packets using reclaimed buffers, thus avoiding both data copying and expensive socket buffer allocation and initialisation. It would be interesting to see if this situation repeats itself when 10 Gigabit NICs are coupled with processors working at today's speeds.

An analytic framework for assessing the benefits of kernel-based group unicast was proposed in [15]. These models cover more complex scenarios than are covered by Amdahl's Law. In this paper, we present the results of experiments conducted on a commercial multimedia server that is widely used by content providers around the world. The live broadcast streaming scenario we consider (see Section 4) turns out to be quite simple to analyse and as a result we do not require the more elaborate models presented in [15]. Instead we show that Amdahl's Law works well in our case, providing quite an accurate prediction of the performance benefits.

## 3. IMPLEMENTATION

The new groupcasting interface consists of a single system call, `sendgroup()`, which takes a group structure and uses the information contained in this structure to create per-recipient packets. This interface has two major benefits over the previous one:

1. Management of the group, including adding and removing members, is now in the hands of the application;
2. Per-recipient information can be prefixed and/or suffixed to the shared group information block;

Giving applications control over group management allows developers to come up with schemes that are more suitable for those

applications. Moreover, adding and removing group members no longer requires separate system calls. This can be especially important in highly dynamic environments, where group membership changes often.

The ability to add per-recipient data to each packet was added in response to community feedback to our original interface and as a result of more closely examining how other applications send data to group members. This feature is useful in some scenarios where the bulk of the information to be transmitted is shared, but each client also requires a small amount of unique data. For example, the Doom/Quake server (quakeforge-0.5.4) sends per-client and shared data in the same packet [23]. The new API allows this data to be transmitted in the same packet as the shared data, saving on both the number of system calls required and the number of packets generated. The `sendgroup()` system call has the following signature:

```
int sendgroup(int sd, struct giobuf* buf,
              size_t recnum, int flags,
              int* gerrno)
```

where *sd* is the socket descriptor, *buf* is a pointer to the group-description buffer (described below), *recnum* is the number of recipients, *flags* has the semantics of its namesake parameter to the `sendmsg()` system call and *gerrno* is an (optional) pointer to an array, into which the system call writes per-recipient error codes. The function returns the number of recipients to which the data was successfully sent out, or an error value if the call failed altogether.

Group information is conveyed in the form of a variable-sized structure, defined as follows:

```
struct giovec {
        /* Destination address. */
        struct sockaddr_in giov_dest;
        /* The prepended data. */
        struct iovec giov_prepend;
        /* The appended data. */
        struct iovec giov_append;
};

struct giobuf {
        /* Shared data buffer. */
        struct iovec shared;
        /* Per-recipient information. */
        struct giovec recinfo[1];
};
```

The implementation is relatively straight-forward on BSD-based operating systems (including FreeBSD and Solaris), where a network packet is represented by a chain of buffers (`mbuf` structures). A groupcasting packet is thus represented by three such buffers: the first contains the IP and UDP headers, as well as any per-recipient information to be prepended to the shared data; the second points to an external block that contains the single copy of the shared information; and the (optional) third buffer holds any appended per-recipient data.

On Linux things are somewhat more complicated, as the native packet representation uses a single socket buffer (`sk_buff` structure). While this structure can only point directly to a single chunk of contiguous memory, it also contains an array of page pointer/page offset pairs that can be used to reference additional packet data. We use this array by allocating as many pages as are required to hold the shared data (normally one), copying the shared data from the user-space buffer directly to these pages, and setting pointers to these pages in the socket buffer's array. Per-recipient appended data, if it exists, is also set as a page pointer in this array. Any prepended data is copied into the `sk_buff`'s internal buffer, along with the required protocol headers.

Both implementations rely on the scatter-gather I/O feature of modern network interfaces. This feature allows the operating system to keep segments of a single packet in non-contiguous memory. The network driver places pointers to these segments on the transmission ring, which is shared by RAM and the NIC's memory. The network card is then responsible for assembling the packet by fetching the different segments via DMA. This process avoids the need to first allocate a contiguous packet buffer in RAM. Moreover, in many cases the NIC can fetch all packet segments in a single DMA transaction.

# 4. EVALUATION

## 4.1 Environment

All benchmarks are conducted on the same server machine. The hardware consists of a SuperMicro server with a single Intel Xeon processor at 3.06 GHz, 2 GB of RAM and four Intel Gigabit NICs on a single 64 bit/66 MHz PCI-X bus. While the CPU supports Intel's HyperThreading technology, this feature is turned off in the machine's BIOS, in order to simplify performance analysis.

For Linux, we use the Fedora 7 distribution with a 2.6.22.5 kernel. The kernel is only slightly modified to export a few functions required by the `sendgroup()` call implementation. The system call itself is implemented as a loadable kernel module. All kernel configuration options are specified according to the Fedora default. While this means that some relevant optimisations are not applied, we have decided to perform our experiments on a platform that closely resembles a default user installation, where the kernel comes pre-compiled by the distribution. For the Helix experiments, the limit on open file descriptors is increased to 65,535, in order to handle a large number of clients.

Solaris experiments are performed with build 72 of Solaris Nevada for x86. FreeBSD tests are conducted on version 6.2 of the operating system. In both cases there is no need to modify the kernel, with the `sendgroup()` system call implemented entirely in loadable modules.

## 4.2 Micro-benchmarks

For these tests we use a simple program that repeatedly transmits packets to a group of clients, using either a loop of `sendmsg()` calls or a single `sendgroup()` call. Each of the tests is repeated 5 times in a row, and we report the average results. In all cases, the standard deviation is too small to show up as error bars in the graphs (less than 1%).

As mentioned earlier, the benefits of using `sendgroup()` result from two factors: reducing the number of mode switches and avoiding multiple copies of the shared data. Let $G$ be the group size, $C_{ms}$ the cost of a mode switch, $C_{mem}$ the cost of copying the shared data and $C_{send}$ the cost of sending the packet. Then the expected improvement factor $r$ of `sendgroup()` over a loop of `sendmsg()` calls is

$$r = \frac{\texttt{sendmsg()}}{\texttt{sendgroup()}} = \frac{G \cdot (C_{ms} + C_{mem} + C_{send})}{C_{ms} + C_{mem} + (G \cdot C_{send})}$$

which, for sufficiently large groups, is

$$r = \frac{C_{ms} + C_{mem} + C_{send}}{C_{send}}$$

We therefore expect the improvement factor to become constant

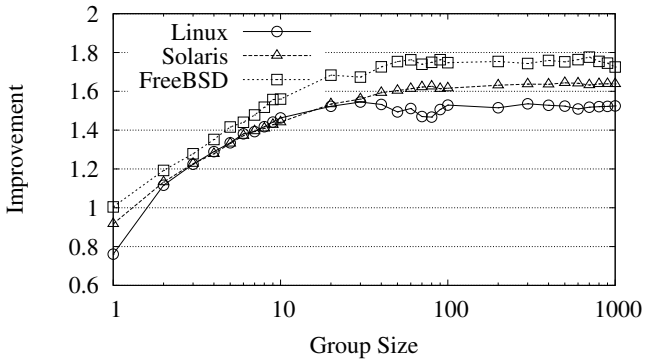


**Figure 1: The improvement factor of `sendgroup()` over a `sendmsg()` loop for different group sizes.**

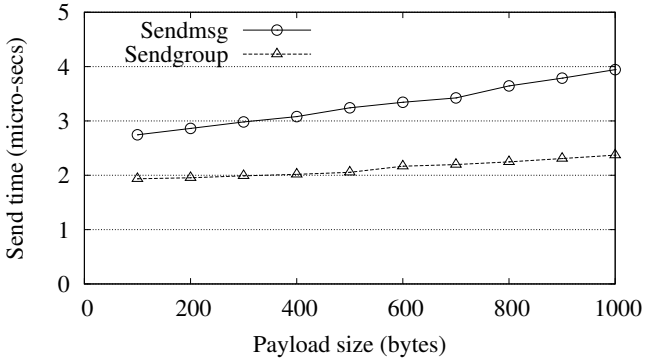


**Figure 2: The amortised per-packet send time in a group of 1000 recipients, as a function of the packet's payload size.**

with respect to the group size, once a certain threshold size is reached. On the other hand, since $C_{mem}$ clearly increases with the size of the payload data to be copied, we expect the improvement factor to be a function of that size.

The first assumption is confirmed by a set of tests, where the payload size is fixed at 300 bytes, and the group size varies from 1 to 1000. In Figure 1 we can see that a performance gain is already achieved when the group comprises only two members (note the logarithmic X scale of the graph). Moreover, the maximal improvement factor is reached for a relatively small group size, and remains almost constant for arbitrarily large groups.

As can be seen in Figure 1, results are fairly similar across the three operating systems. In the rest of this section, for lack of space, we only report Linux results, as those are relevant for the macro-benchmarks we consider in the next section (the Helix server we use for those tests is not supported on Solaris and FreeBSD).

In the next set of experiments, the group size is fixed at 1000, and payload size varies from 100 to 1000 bytes. Figure 2 depicts the amortised send time per packet for each of the implementations on Linux, obtained by dividing the total time of the experiment by the number of packets sent. The widening gap between `sendmsg()` and `sendgroup()` is the result of increased savings when reducing the number of copies of larger data chunks. This verifies our second assumption, namely that the improvement factor is a function of the payload size.

Interestingly, the packet send time is not constant (with respect to the payload size) in the `sendgroup()` case. We conjecture that it may be the result of memory and bus contention, which increase with the overall throughput. Specifically, at 1000 bytes, the

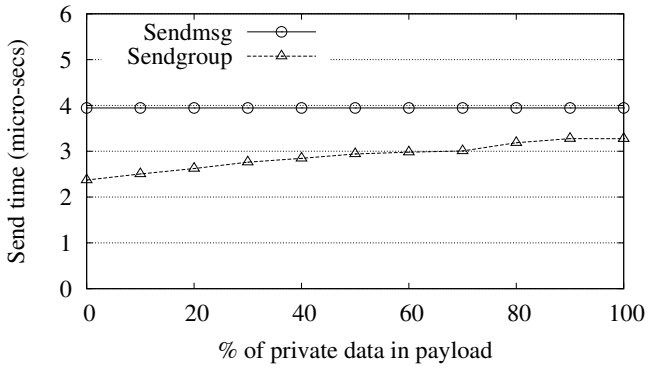


**Figure 3: The effect of private data on the overall performance of `sendgroup()`. The X-axis gives the percentage of a 1000 byte packet taken by private data. Results are expressed in amortised time for sending one packet to a member of a group of size 1000.**

PCI bus needs to sustain approximately 3.6 Gbps of packet data, which, along with the control data, comes close to the maximum bandwidth supported by a 64 bit-66 MHz bus.

Finally, we consider the new interface's ability to handle private data. In this case, each of the clients receives different payload data. An interesting question is whether `sendgroup()` is still beneficial in this case. The answer is strictly affirmative: `sendgroup()` performs better than a loop of `sendmsg()` system calls even if no data is shared, in which case the benefits of `sendgroup()` stem solely from avoiding per-message mode switches. Figure 3 shows the amortised per packet send time as a function of the percentage of private (prepended) data. Obviously, the smaller the portion of shared data the smaller the performance improvements obtained by using `sendgroup()`. Nevertheless, an improvement is observed across the range of shared/private data ratios.

We have assumed that the send time for the `sendmsg()` case is constant for a packet size of 1000 bytes, regardless of the mixture of private and shared data (and have therefore used the relevant send time from Figure 2). Depending on the application, extra work may need to be performed to create these packets, so the gap between `sendmsg()` and `sendgroup()` may be slightly larger in reality.

Although the micro-benchmarks in this section show that significant benefits can be obtained from using kernel groupcast, the benefits are not as large as reported in our previous work [14]. We believe this is due to the more advanced server hardware used in our current experiments, along with recent changes to the operating systems examined. In particular, the hardware used previously combines a slow processor with relatively fast NICs, which creates an ideal – yet somewhat unrealistic – environment for the lazy-copy variant on Linux (i.e., very few, if any, copies were required). The version of FreeBSD used in the previous experiments (5.2.1), was still a development release, following significant changes to the design of the kernel. Subsequent versions have significantly improved kernel locking, bringing its performance back in line with Linux and Solaris and decreasing the gap between the performance of `sendmsg()` and kernel groupcast.

## 4.3 Helix Experiments

In order to determine what kind of benefits a real application might obtain from using `sendgroup()`, we have added support for our new system call to version 11.1 of the open source Helix multimedia server [22]. The experimental environment consists of a producer, which generates an RTP stream; the server, which re-

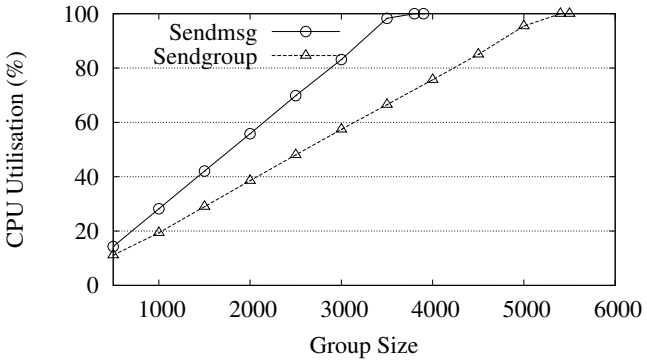


**Figure 4: A Helix server's processor utilisation while transmitting a 100 Kbps RTP/UDP stream.**

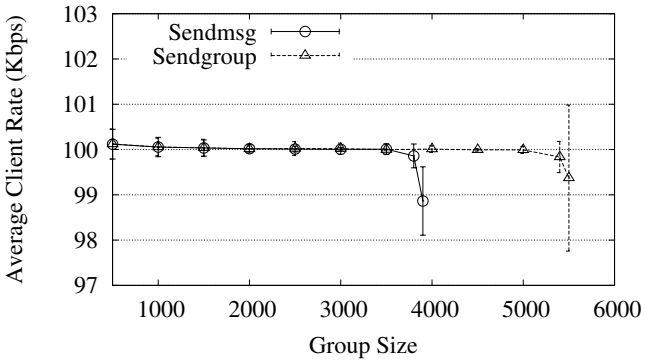


**Figure 5: Quality of service, expressed as the average rate received by clients. The transmitted rate is 100 Kbps.**

ceives the incoming stream and transmits it to registered recipients; and a set of client machines, each capable of handling multiple stream recipients. This type of configuration is common in live-broadcasting scenarios, where the producer captures video and/or audio and forwards it to a server for distribution to multiple clients.

We note that incorporating the `sendgroup()` system call into the Helix server code is not straight-forward. In the unicast scenario, the server maintains a different stream object for each client. Each of these objects sends packets independently of its peers, allowing the server to handle both on-demand and live content. We introduce a group-unicast feature to Helix, loosely based on the existing multicast code, in which the server uses a single stream object for all clients. Packet transmission can be accomplished by either a user-mode loop of `sendmsg()` calls, or a single `sendgroup()` call.

Our experiments require the transmission of a 100 Kbps media stream to increasingly larger groups of clients. Note that the stream has a fixed rate, which implies that the task of sending a given amount of data over RTP would always take the same time to complete. Thus, total execution time cannot be considered as a metric for the Helix experiments. Instead, we consider processor utilisation during the experiment, with the expectation that the benefits of `sendgroup()` will translate into lower CPU utilisation. To quantify the improvement, we fix the group size and examine the CPU utilisation for `sendgroup()` and the `sendmsg()` loop. Additionally, we examine the maximum group size supported by the two methods. These comparisons can be seen in Figure 4. For group sizes of 1000 to 3500, the graphs show a reduction in CPU utilisation of 31% when using `sendgroup()`, i.e., CPU utilisation is 1.45 times higher when using `sendmsg()` than when using `sendgroup()`. The `sendmsg()` loop saturates the CPU at 3800 clients, while saturation is not reached until 5400 clients with `sendgroup()` (a factor of 1.42 more clients).

We have found that conventional statistical methods, such as `vmstat`, yield highly inaccurate results for processor usage. As a result we use a highly accurate measurement technique, that divides the number of busy CPU cycles by the total number of cycles (busy and idle). On the x86, we use a performance counter to measure the former and the TSC (time-stamp counter) register to obtain the latter.

Since the server uses UDP to transmit packets, examining only server performance could be misleading, as packets may be silently dropped at various stages of the transmission path. Figure 5 depicts the quality of service for all clients in the group, expressed as the average bandwidth received by the clients (with the expected value being 100 Kbps for all clients). A noticeable drop in quality of service can be seen after 3800 clients in the `sendmsg()` loop case and 5400 clients for `sendgroup()`. We can deduce from the QoS results that 3800 and 5400 are indeed the maximum *supported* group sizes for each of the groupcast methods.

It is worth noting that it is important to be able to serve large numbers of clients in such scenarios. For example, the RealNetworks broadcast of the television show Big Brother 10 required support for tens of thousands of viewers [4].

## 4.4 Performance Prediction

In previous work [14] [15], we claim that micro-benchmarks can be used to predict the expected benefits to an application from using kernel groupcast. We now show that this is indeed the case and demonstrate how Amdahl's Law predicts quite nicely the benefits obtained from using `sendgroup()` in the Helix server. In these calculations we use a group size of 1000 clients and a payload size of 1000 bytes.

To use Amdahl's Law, we require the improvement factor for the sped-up part of the application, which, from the micro-benchmarks, is $s = 1.664$ for the given group and payload sizes (see Figure 2). Next, we need to determine the relative time the Helix server spends executing that part. We define this fraction of time as

$$f = \frac{T_{sc} + T_{irq} + T_{bh} - T_{sleep}}{T_{exp}}$$

where $T_{sc}$ is the time spent in the `sendmsg()` system call; $T_{irq}$ and $T_{bh}$ is the time spent servicing the asynchronous parts of transmission (hard and soft interrupts) *outside* the time frame of the system call (asynchronous time inside that time frame is already accounted for in $T_{sc}$); $T_{sleep}$ is the time the calling thread spends in a non-running state during the system call's time frame; and $T_{exp}$ is the total running time of the experiment. We use performance counters [12] to obtain the number of busy CPU cycles for each of these components, with a result of $f = 0.791$.

Applying Amdahl's Law, we get a predicted improvement factor of

$$r = \frac{1}{1 - f + \frac{f}{s}} = \frac{1}{1 - 0.791 + \frac{0.791}{1.664}} = 1.461$$

compared with the observed value of 1.45 for CPU utilisation (see Figure 4). The predicted value also agrees with the achieved capacity improvement factor of 1.42. These results demonstrate that it is possible to predict the benefits that can be obtained from using `sendgroup()` in a real and complex application, prior to ac-

tually modifying it. This permits relatively informed cost-benefit decisions about whether to perform such an integration.

# 5. CONCLUSION

In this paper we have presented a new system call, `sendgroup()`, that can be used to efficiently send copies of the same data to multiple recipients. The API for this system call is much more flexible than the previous interface for kernel-based group unicast, and allows both shared and private data to be transmitted to clients. We have shown that the system call can be introduced into the existing code base of a complex application, permitting an increase in group size of a factor of 1.42. This improvement in efficiency translates naturally to significant cost and environmental savings due to the reductions in energy consumed to cool and power fewer servers.

The benefits of using kernel-based group unicast come from reducing the number of mode switches and from avoiding redundant memory copies of the shared data. The first of these factors is constant per system call, and depends on both the hardware and the operating system. Even though some platform advances have reduced the mode switch overhead (e.g., the introduction of the `sysenter` instruction on the x86 and its adoption by operating systems), our work shows that avoiding system calls when possible is still a good strategy for improving performance. On the other hand, the cost of memory copies is proportional to the amount of memory being copied and also depends on overall memory access contention.

We have also shown that relatively simple techniques can be used to collect the performance data required to predict the benefits of using `sendgroup()` in a real application. The prediction turns out to be quite accurate in the relatively complex Helix streaming server we examined. The accuracy of the results suggests that this technique may be of value in other scenarios. In the future we hope to extend the ad-hoc framework we have built to obtain these results into a more generally-applicable framework for precise performance analysis.

# 6. ACKNOWLEDGEMENTS

This work has been supported by RealNetworks, Sun Microsystems, and the Natural Sciences and Engineering Research Council of Canada.

# 7. REFERENCES

[1] 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*, 1998.

[2] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. Technical Report UMIACS TR-2002, University of Maryland, 2002.

[3] Y. Chu, S. Rao, and H. Zhang. A Case for End System Multicast. In *Proceedings of ACM SIGMETRICS*, pages 1–12, 2000.

[4] A. Colwell. Challenges with Developing a Commercial P2P System. In *Panel Presentation, NOSSDAV 2007*, July 2007.

[5] S. E. Deering. Multicast routing in internetworks and extended LANs. In *Symposium Proceedings on Communications Architectures and Protocols*, pages 55–64, 1988.

[6] C. Diot and L. Gautier. A Distributed Architecture for Multiplayer Interactive Applications on the Internet. *IEEE Network*, 13(4):6–15, 1999.

[7] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment Issues for the IP Multicast Service and Architecture. *IEEE Network*, 14(1):78–88, 2000.

[8] P. Druschel and L. L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Symposium on Operating Systems Principles*, pages 189–202, 1993.

[9] K. Fall and J. Pasquale. Exploiting in-kernel data paths to improve i/o throughput and cpu availability.

[10] P. Francis. Yoid: Your Own Internet Distribution. `http://www.aciri.org/yoid`. accessed February 2008.

[11] P. Halvorsen, T. A. Dalseng, and C. Griwodz. Assessment of Linux' Data Path Components for Download and Streaming. *The International Journal of Software Engineering and Knowledge Engineering*, 17(4):465–481, 2007.

[12] Intel. *Intel®64 and IA-32 Architectures Software Developer Manual, Volume 3b*, 2007.

[13] 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*, pages 197–212, 2000.

[14] M. Karsten, J. Song, M. Kwok, and T. Brecht. Efficient Operating System Support for Group Unicast. In *Proceedings of the 15th International Worshop on Network and Operating Systems Support for Digital Audio and Video*, pages 153–158, 2005.

[15] M. Kwok, T. Brecht, M. Karsten, and J. Song. Modelling and Improving Group Communication in Server Operating Systems. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 207–217, 2006.

[16] T. Maeda and A. Yonezawa. Kernel Mode Linux: Toward an Operating System Protected by a Type Theory. In V. A. Saraswat, editor, *8th Asian Computing Science Conference*, pages 3–17, 2003.

[17] S. McCanne and V. Jacobson. vic: a flexible framework for packet video. In *Proceedings of the Third ACM International Conference on Multimedia*, pages 511–522, 1995.

[18] E. Nahum, T. Barzilai, and K. Kandlur. Performance Issues in WWW Servers. 10(1), February 2002.

[19] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[20] 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*, pages 49–60, 2001.

[21] S. Ratnasamy, A. Ermolinskiy, and S. Shenker. Revisiting IP multicast. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 15–26, 2006.

[22] The Helix Community. The Helix DNA Server. `https://helix-server.helixcommunity.org/`.

[23] The QuakeForge Project. `http://quakeforge.net`. accessed February 2008.

[24] E. Zadok, S. Callanan, A. Rai, G. Sivathanu, and A. Traeger. Efficient and Safe Execution of User-Level Code in the Kernel. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 8–15, 2005.