In-Kernel Relay for Scalable One-to-Many Streaming

Ying-Dar Lin, Chia-Yu Ku
National Chiao Tung University, Taiwan

Yuan-Cheng Lai
National Taiwan University of Science and Technology

Chia-Fon Hung
National Chiao Tung University, Taiwan

The in-kernel One-to-Many Streaming Splicing (OMSS) relay method can help improve the relay data paths of critical nodes to reduce computing power for UDP and TCP streams and enhance the subscriber capacity.

As a result of advances in compression techniques and the increase of network bandwidth, streaming techniques are now widely used to distribute multimedia over the Internet. For one-to-many streaming applications, such as Internet radio and IPTV, thousands of consumers might request popular content from a media provider. The media provider can distribute the content using IP multicast or overlay multicast. IP multicast distributes the fan-out burden to intermediate routers to minimize duplicate media content. However, problems with group management, address allocation, and security issues can make IP multicast unavailable.[1] The overlay multicast technique organizes peers to forward media streams and realizes multicast transmissions using multiple unicasts in overlay networks.[2–5] Unfortunately, the duplicate unicasts make overlay multicast less efficient than IP multicast.

In overlay networks, a relay node could be a rendezvous node in a centralized architecture or a peer in a purely decentralized architecture. A rendezvous node, such as a proxy relay server, can suffer from a large amount of fan-out burdens. In addition, a peer with limited computing power due to hardware limitations, such as a low-end SOHO (small office, home office) router, might be insufficient to deal with the requested media streams. Unable to deal with the fan-out burden, such relay nodes then become a bottleneck. Therefore, improving the performance of a critical node can enhance system scalability and reduce the required computing power.

Forwarded media streams can suffer from the same overhead problems as memory copies and system calls. Previous works utilized a unified buffer structure with page remapping and shared memory to prevent cross-domain memory copies.[6–9] However, these approaches lost compatibility when the operating systems' underlying buffer structure was modified. The multicast relay projects performed the relay operations through the user-space socket API by altering the underlying buffer structure, but they still could not eliminate the overhead from system calls and memory copies. Alternative works proposed header-altering methods, which intercept packets in the IP layer.[10] However, they suffered maintenance overhead problems from transport-layer protocols that are generally applied to multimedia streams. KStream built an in-kernel streaming relay data path over the I/O subsystems to prevent system calls and memory copies,[11] but memory copies among sinks are still required. (See the ''Related Works in Relay Data Paths'' sidebar for more details.)

This article proposes an in-kernel streaming relay method for Linux-based systems called *one-to-many streaming splicing* (OMSS) that supports both User Datagram Protocol (UDP) and TCP streaming. We use a payload-sharing mechanism to reduce the number of memory copies among multiple sinks and the worker-pool processing model to raise the service scalability in a multiprocessor system. Moreover, we provide differentiated QoS scheduling to guarantee the service quality of high-priority streaming sessions.

We evaluated the performance of our OMSS approach on both PC-based and embedded platforms. Our experimental results demonstrate that OMSS reduces CPU utilization for UDP and TCP streams by 56 and 59 percent, respectively, compared with a conventional approach. In addition, the maximum number of media subscribers on the PC-based and

1070-986X/13/$31.00 © 2013 IEEE

Published by the IEEE Computer Society

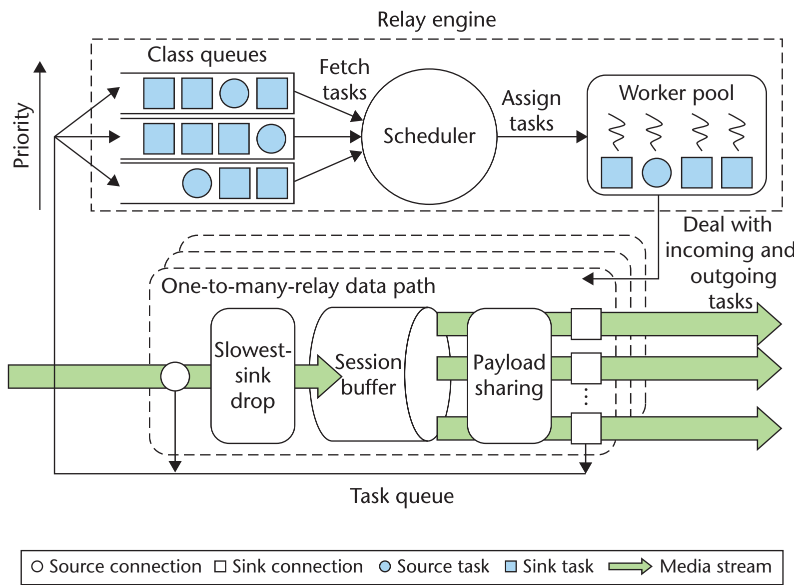


Figure 1. One-to-many streaming splicing (OMSS) architecture. A task might enter the queue as a result of a socket event, application control, or processing result.

embedded platforms increased from 2,200 to 4,000 and from 45 to 85, respectively.

One-to-Many Streaming Splicing

The OMSS architecture consists of a one-to-many relay data path and a relay engine. Media content is forwarded on the relay data path, whereas the relay engine manages the system resources for the relay sessions. Figure 1 shows that each relay session includes a corresponding relay data path. A one-to-many relay data path consists of a source connection, multiple sink connections, and a session buffer. OMSS receives media streams from the source connection while it sends the streams to the subscribers through the sink connections. The received media content is stored in the session buffer. For real-time multimedia streaming services, we apply the slowest-sink-drop policy, which drops the packets held by the slowest sink connection, to keep new media content when the session buffer is full. OMSS then utilizes a payload-sharing mechanism to send out the media content with no memory copies.

Our approach considers a relay session's receptions and transmissions as one source task and multiple sink tasks, respectively. The relay engine schedules and handles all tasks, utilizing the worker-pool processing model to achieve large-scale service scalability on multiprocessor platforms. Moreover, we also provide priority-based task scheduling to ensure the QoS of high-priority sessions. When a source task is triggered, it is added to the class queues according to its priority assigned by the streaming service program. Then the worker pool assigns an idle thread to deal with the task and puts the sink tasks generated by the source task into the class queues. The sink tasks send out the media content sharing the same payload. These tasks forward media content until no additional content is received by the source task.

Figure 2 shows the flow chart of processing tasks. When a source task is triggered, the buffer size is examined to ensure sufficient spaces. If the buffer is full, the slowest-sink-drop policy is applied. After receiving media content and

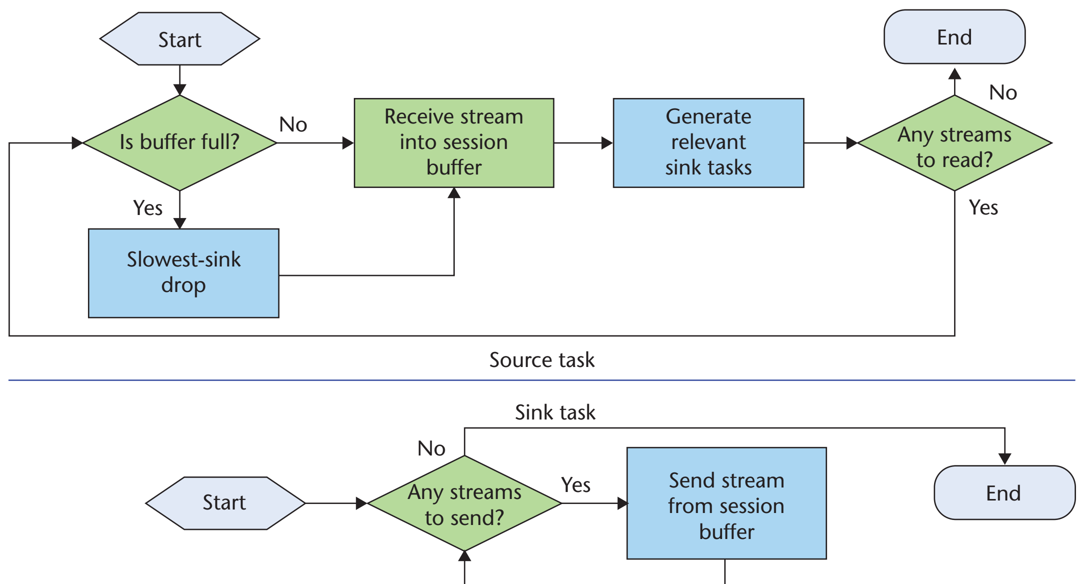


Figure 2. OMSS flow chart. OMSS provides buffer management for one-to-many streaming over the socket layer and uses the slowest-sink-drop policy to reserve buffer space for newly arriving content.

Related Works in Relay Data Paths

A relay data path fcontinually forwards media streams from the source to the sinks. Media streams relayed through different interfaces can incur different types of overhead. We categorize previous relay-data-path solutions by interface type: socket API, I/O subsystems, and IP-layer hooks. Figure A illustrates the categories of relay data paths, and Table A summarizes the features and drawbacks of each.

Relay through the Socket API

The relay data paths through the socket API forward media streams with the socket interface in the user space. Initially, the incoming media streams pass through the network protocol stack. Then, the contents of the streams are stored in the source socket's receiving queue in the kernel space. The relay application copies content to the user-space buffer and passes it to the sending queue of the sink sockets in the kernel space. Because media streams are copied across the user and kernel spaces, the relay data paths incur system call and memory copy overhead. Icecast (www.icecast.org) and Darwin Streaming Server (DSS, http://dss.macosforge.org) belong to this category. The former addresses TCP streaming, whereas the latter handles UDP streaming.

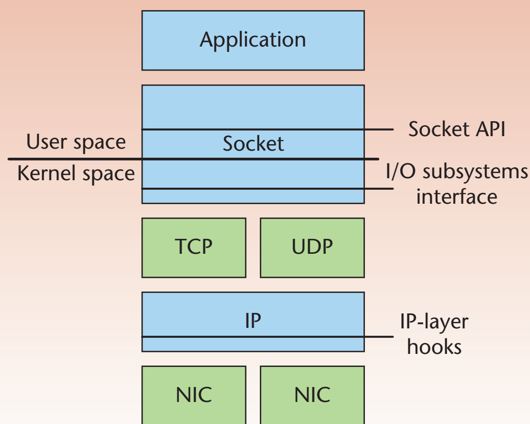


Figure A. Three categories of relay data paths.

Relay at the IP-Layer Hooks

An in-kernel relay data path can prevent the overhead caused by copying data across the user and kernel spaces. To build an in-kernel relay data path, IP-layer hooks such as Netfilter (www.netfilter.org) in Linux-based systems allow kernel modules to register callback functions in the IP layer. Packets are intercepted at each hook point and passed to the corresponding callback function.

Our previous work, called kP2PADM,[1] examined the performance improvement of a peer-to-peer (P2P) management gateway in the kernel space. kP2PADM was implemented by the L7 filter, which is an extension of the IP-layer hooks. Our experimental results showed an in-kernel design can significantly enhance throughput and CPU utilization. However, kP2PADM did not address one-to-many relay applications and did not investigate the overhead caused by memory copies. Although kP2PADM let us eliminate the memory copies between the user and kernel spaces, we did not explore all other issues, including the overhead of memory copies among

putting it into the buffer, the source task invokes the corresponding sink tasks in the same relay session to send out the content. A sink task gets the content from the buffer and sends it out until there is no additional content. The content can be removed after all the sink tasks have completed.

Slowest-Sink-Drop Buffer Management

The OMSS design applies one-to-many streaming features to buffer management. When the buffer is full, the buffer manager should apply a queuing discipline to drop packets. Unlike buffer management at a router, OMSS provides buffer management for one-to-many streaming over the socket layer. The decision to drop packets should not affect transport-layer behaviors, such as the congestion control.

Media content delivered by the sources is stored in the session buffer. Sending this content to a requesting set of sinks can require various amounts of time due to the different transmission qualities of the sink connections. Therefore, a slow sink connection might occupy the buffer. In one-to-many streaming, tail-drop or random early detection (RED) algorithms might drop newly arriving media content when the buffer is full. Dropping the newly arrived content forces all the sinks to lose the dropped content. That loss can result in penalties such as video lags, even though a part of the sink connections has sufficient bandwidth. To avoid this problem, we apply the slowest-sink-drop policy, which drops the content held by slowest sink connections, to reserve buffer space for newly arriving content.

Note that the buffer management mechanism we choose, whether it be the slowest-sink-drop, tail-drop, or RED, does not influence the overhead, such as memory copies, in a

relay data paths. Moreover, the transport-layer processing must be handled by the callback functions because the media streams are not passed through the entire network protocol stack. The maintenance costs increase when a transport-layer protocol, especially TCP, is applied to the media streams.

TCPSP[2] (www.linuxvirtualserver.org/software/tcpsp) and Media Proxy (http://mediaproxy.ag-projects.com) both built relay mechanisms with the IP-layer hooks using header-altering methods to support one-to-one relay schemes.

Relay over the I/O subsystems

An alternative solution to in-kernel relay data paths realizes streaming relay over I/O subsystems. Such approaches create a unified interface over the I/O and network subsystems in the kernel space. The solution intercepts packets over the interface and prevents streaming relay from the overhead caused by copying data across the user and kernel spaces. Moreover, the maintenance overhead of transport-layer protocols can be avoided because the packets pass through the entire network protocol stack. KStreams made a data-abstraction layer over I/O subsystems.[3] Based on the data-abstraction layer, KStreams can support one-to-many relay among different kinds of I/O subsystems but does not eliminate the overhead caused by memory copies from multiple sinks.

References

1. Y.-D. Lin et al., ''kP2PADM: An In-Kernel Architecture of P2P Management Gateway,'' *IEICE Trans. Information and Systems,* vol. E91-D, no. 10, 2008, pp. 2398–2405.
2. D.A. Maltza and P. Bhagwata, ''TCP Splice for Application Layer Proxy Performance,'' *J. High Speed Networks,* vol. 8, no. 3, 1999, pp. 225–240.
3. J. Kong and K. Schwan, ''KStreams: Kernel Support for Efficient Data Streaming in Proxy Servers,'' *Proc. Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video* (NOSSDAV), ACM, 2005, pp. 159–164.

Table A. Related work in relay data paths.

Category	Features	Drawbacks
Socket API	Relay in the user space Different processing for TCP and UDP	Memory copies between the user and kernel spaces System calls
IP-layer hooks	Relay in the kernel space Callback functions at the IP-layer hooks	Transport-layer protocol processing
I/O subsystems	Relay in the kernel space A unified interface over I/O subsystems	Memory copies for multiple sinks

one-to-many relay node. This is because buffer management affects end-to-end performance rather than the system utilization of a relay node, which is the main performance metric we focus on in this study.

To realize the slowest-sink-drop mechanism, we implement the session buffer as a ring buffer. Each buffer entry has a pointer to link the corresponding memory page and a variable (user count) to indicate the number of sinks that requires the entry. Each task can trace the buffer entry with the corresponding media content. Figure 3 shows an eight-entry session buffer for five sinks. In Figure 3a, four buffer entries in the session buffer—that is, a single user count—are occupied by the slowest sink task. Using the slowest-sink-drop policy, newly arriving media content results in dropped entries, so the buffer entry with the smallest user count would be dropped. Then the user count of the page allocated to the slowest sink task is decreased. Figure 3b shows that the entry's user count is decreased by one because this media content is no longer required by the slowest sink. The released page is then reallocated to the newly arriving content.

Payload Sharing

In a relay session, the same media stream is forwarded from a source connection to multiple sink connections. In previous works,[11] the transmissions of the same media content required duplicating it for the various sink connections. That is, a 1-to-N relay session required N memory copies, as Figure 4a shows.

To avoid memory copy overhead, the OMSS payload-sharing mechanism provides only one payload that is shared by the sink connections. OMSS can reuse the payload

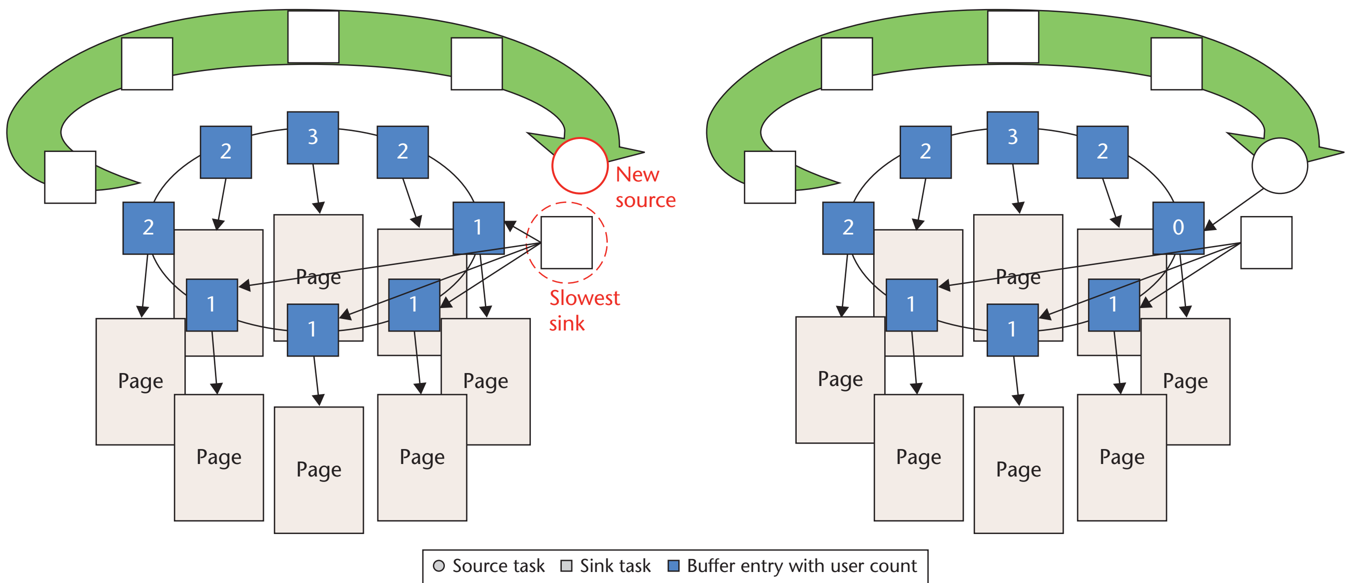
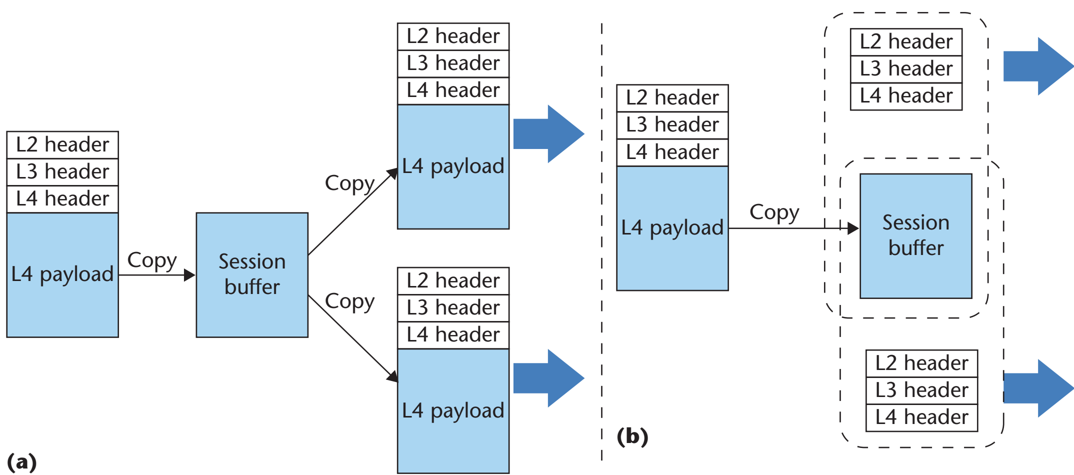


Figure 3. An eight-entry session buffer for five sinks. (a) A slow sink can cause the session buffer to fill up. (b) Buffer entries are released using the head-drop policy, which only sacrifices the slowest sink.



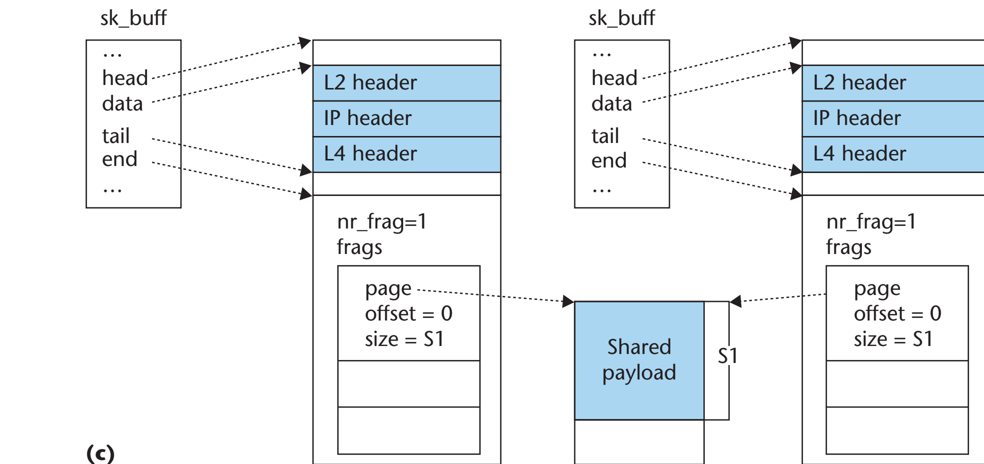


Figure 4. OMSS payload-sharing mechanism. (a) Payload copy, (b) payload sharing, and (c) sk_buff structures with a shared payload.

OMSS provides priority-based scheduling to achieve a differentiated QoS mechanism for high-priority relay sessions.

for all the sink connections without duplicating it (see Figure 4b).

A user-space solution using the socket API only requires the payload's address and size. The socket libraries handle additional operations such as fragmentation. However, to port the relay to the kernel space, we must consider the packet structure and fragmentation. In Linux-based systems, a packet is represented by the sk_buff and auxiliary data structures. The scatter-gather I/O is performed when payloads are stored in the memory pages, which are appended to the frags array of a sk_buff structure. Figure 5c illustrates the sk_buff structures with a shared transport-layer payload. The frag entries of each sk_buff point to the shared payloads. The socket-layer function, `sendpage()`, can send the payloads stored in discontinuous memory pages. When using `sendpage()`, we can simply achieve payload sharing in Linux-based systems in two steps: extracting the payload into memory pages and sending out the memory pages using `sendpage()`. By cooperating with the session buffer implemented by memory pages, different sinks can share the payload.

Relay Engine

When receiving media content from the source socket, a source task is triggered and put into the class queues based on its priority. The corresponding sink tasks are invoked by the source task and put into the class queues. When the worker pool contains idle threads, the scheduler selects a task to be handled. Otherwise, the worker pool creates new as many threads_ as the existing ones.

A task-processing model manages processes or threads to handle service tasks. There are two kinds of task-processing models. The *single-thread model*[11] cannot fully utilize a multiprocessor system's computing power because a single thread cannot utilize multiple processors in parallel. The *per-session thread model*, on the other hand, might incur context-switching overhead when the number of sessions grows. To achieve large-scale service capability, OMSS adopts the worker-pool processing model to handle source and sink tasks. A worker pool consists of a set of prespawned threads and a task assignment interface. Initially, each thread is idle and waits for service tasks. When a task is triggered, the worker pool assigns an idle thread to the task through the task-assignment interface. Because the threads are prespawned, the thread creation overhead is small. The idle threads in the pool incur no context switch, which eliminates the unnecessary context-switching overhead.

OMSS provides priority-based scheduling to achieve a differentiated QoS mechanism for high-priority relay sessions. Each session's priority value, which is assigned by the streaming service program, is transferred to the corresponding sink tasks. Tasks are then put into the class queues according to their priority values. The scheduler simply selects a task from the class queues with the highest priority to guarantee the QoS of media streams.

Evaluation

We performed two tests to evaluate the OMSS approach on both PC-based and embedded platforms. The bulk fan-out test shows the maximum number of media clients that a media server can serve, and the payload-size test measures how much we can reduce memory copies.

To examine the improvement of OMSS, we investigated one solution in the user space and another in-kernel solution without the payload-sharing mechanism. The user-space solution (Daemon) performs media-streaming relay through the socket API without the payload-sharing mechanism. To explore the load distribution of multimedia streaming applications over multiprocessor systems, Daemon uses as many threads as the number of processors. One of the threads deals with receiving media streams from the media server while the other handles the transmission to the media client. On the other hand, the in-kernel solution (OMSS-M) is the same as OMSS, except for the payload-sharing mechanism.

Evaluation Environments

Figure 5a shows the evaluation environments for our tests. We applied two kinds of platforms to evaluate the performance of OMSS and investigate the bottlenecks on each platform. Using a PC-based platform with sufficient computing power, we aimed to examine network bandwidth usage. We then used an embedded platform with limited computing power to evaluate a potential bottleneck. The media server and media client are located on the same PC. The media server sends out the media streams to Relay-PC through a direct Ethernet link. Then Relay-PC forwards the streams back to media client through the same Ethernet link. A controller is connected to the devices through an out-of-band link to separate the control flows from the data flows. Figure 5b shows that the controller is connected to Relay-Embedded through a console line instead of an Ethernet link. The link capacities measured by Iperf (http://sourceforge.net/projects/iperf) were 941 and 40 Mbps, respectively (see Figure 5).

Table 1 gives the specifications of each evaluation platform. Relay-PC is equipped with an AMD Athlon 64 X2 CPU and an Intel 82574L gigabit adapter with the scatter-gather I/O. Relay-Embedded is equipped with the Realtek RTL8186 SoC, which consists of an MIPS processor and a fast Ethernet controller with the scatter-gather I/O.

The media server is a multithreaded program that can emulate multiple media providers to send out media streams with different payload sizes and transmission rates, and the media client is a multithreaded program that can emulate multiple media subscribers to receive media streams and report packet-loss ratios and transmission latency.

Bulk Fan-Out Test

In the bulk fan-out test, the media client creates multiple subscribers to request media streams

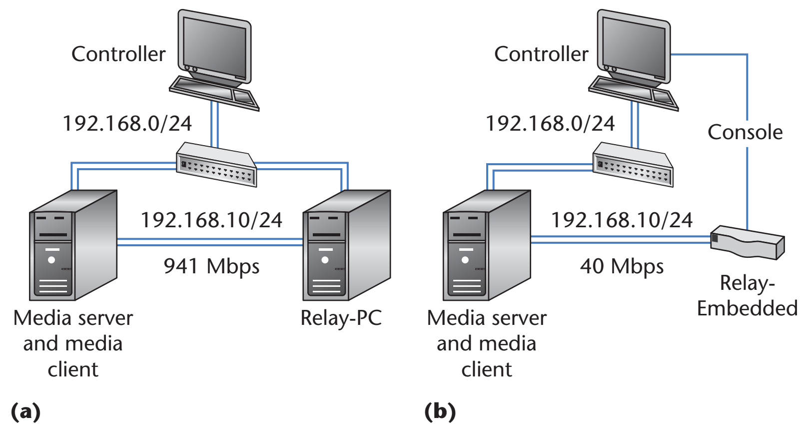


Figure 5. Evaluation environments. (a) Relay-PC. (b) Relay-Embedded.

from the media server. Each stream sends 1,400-byte TCP/UDP payloads at 256 Kbps. Only one relay session is established. The number of subscribers increases until the system capacity is reached. We measured each solution's CPU utilization on both Relay-PC and Relay-Embedded.

Relay-PC Results. Figure 6a shows Relay-PC's CPU utilization as it dealt with UDP media streams. When the CPU utilization reaches 60 percent, Daemon is limited by the computing power and capable of serving fewer than 2,200 media subscribers because it uses only two threads to deal with the transmissions and receptions. Also, as a result of the unbalanced workloads, one of the processors becomes overloaded while the other is underutilized. Therefore, the multiprocessor system's computing power is not fully utilized.

OMSS and OMSS-M, on the other hand, are not bounded by the computing power and can serve more than 4,000 media subscribers. Because OMSS reduces the number of memory copies using the payload-sharing mechanism, it requires less computing than OMSS-M.

Table 1. Evaluation platform specifications.

Specification	Media server and client	Relay-PC	Relay-Embedded
Kernel	2.6.32.4	2.6.32.4	2.4.18
CPU	Intel Core2 Duo E8400 at 3 GHz	AMD Athlon64X2 4000+ at 2.1 GHz	MIPS at 180 MHz
Memory	2 Gbytes	2 Gbytes	16 Mbytes
Ethernet interface	Marvell 88E8056 gigabit adaptor	Intel 82574L gigabit adaptor	RTL8168 Ethernet

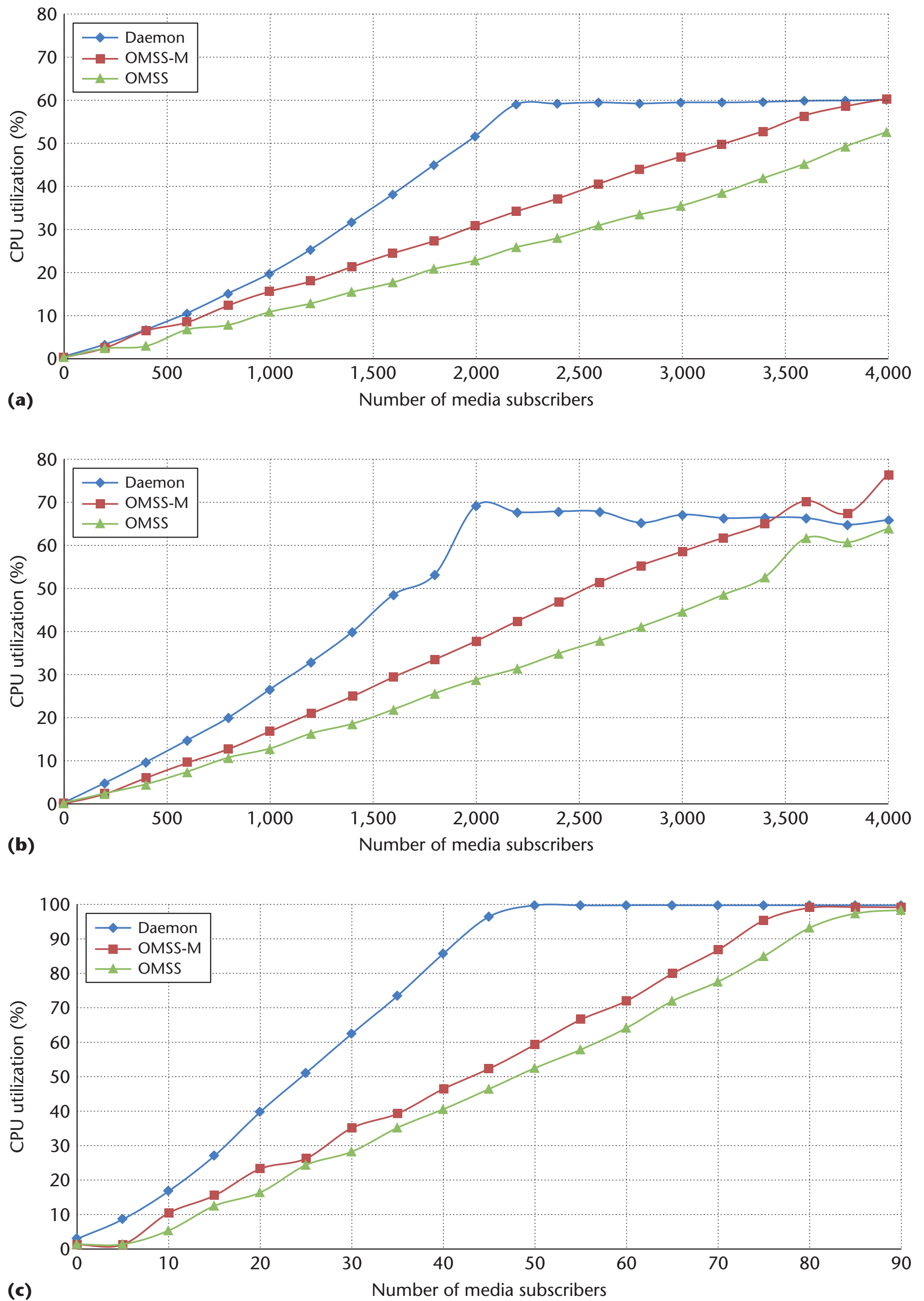


Figure 6. CPU utilization during the bulk fan-out test. (a) UDP on Relay-PC. (b) TCP on Relay-PC. (c) TCP on Relay-Embedded.

Compared with Daemon, OMSS reduces CPU utilization by 56 percent.

Figure 6b shows the experimental results of the bulk fan-out test with TCP payloads. Daemon again suffers from insufficient computing power due to the unbalanced loads. At most, Relay-PC can serve 2,000 media subscribers when the CPU utilization is 70 percent. OMSS can serve up to 3,500 media subscribers, using less computing power than OMSS-M. In this

case, OMSS reduces CPU utilization by 59 percent compared to Daemon.

We noticed that CPU utilization fluctuated when the number of media subscribers exceeded 3,500 because the network bandwidth was exhausted. TCP started the congestion control and tried to retransmit the lost media content. Thus, the CPU utilization fluctuated with the TCP's congestion control and retransmission overhead.

Relay-Embedded Results. Because the Linux 2.4 kernel does not support the `sendpage()` function for the UDP socket, we only performed the bulk fan-out test for the TCP socket on Relay-Embedded. Figure 6c shows the experimental results. Daemon, OMSS-M, and OMSS can serve up to 45, 80, and 85 media subscribers, respectively, until the computing power is exhausted because now the network bandwidth is never a bottleneck. Based on our evaluation results for these two platforms, we see that the number of media subscribers that the platforms can serve is not proportional to the computing power of the CPUs.

The checksum computation is completed by software on Relay-Embedded and by the network interface card (NIC) on Relay-PC. The required computing power by the software checksum is much more than offloading to the NIC. Moreover, the protocol stacks' data paths have different design details, such as the new API (NAPI) in the Linux kernel 2.4 on Relay-Embedded and Linux 2.6 on Relay-PC. Further maintenance overhead of the operating systems might also consume the computing power, so future work must investigate the relay data path to identify the overhead of the relevant functions.

Payload-Size Test

Our payload-size test looked at a one-to-800 relay session at 256 Kbps. We evaluated the performance of the payload-sharing mechanism for both UDP and TCP streams (see Figure 7). The cross points are revealed at the 200-byte UDP payload and the 100-byte TCP payload. Beyond those cross points, the payload-sharing mechanism did not enhance the performance because small payloads contributed little improvement. As the payload size increased, the memory copy overhead also grew, which shows the benefit of the payload-sharing mechanism. The payload-size test revealed a significant improvement for media streams with large payloads.

Conclusions and Future Work

The bulk fan-out test shows that OMSS eliminates the CPU bottleneck and reaches link capacity. For TCP and UDP streams, the bulk fan-out test showed 56 and 59 percent improvement, respectively, over Daemon. On the embedded platform, OMSS increased the number of concurrent streams from 45 to 85, compared with Daemon. Moreover, the payload-size test showed the benefit of the payload-sharing mechanism when the payload size is greater than 200 bytes.

In the future, we plan to investigate the buffer-management drop policy and compare the performance of relay mechanisms, such as IP-layer hooks and I/O subsystems. It is also necessary to look at QoS when playing multimedia under various buffer-management mechanisms, such as slowest-sink-drop, tail-drop, and RED.

The payload-sharing mechanism is currently realized by the scatter-gather I/O, but low-end platforms in future implementations might not be equipped with an NIC with the scatter-gather I/O. To make OMSS portable, we intend to design a payload-sharing mechanism without the scatter-gather I/O.

Lastly, we found that the number of media subscribers that PC-based and embedded platforms can serve is not proportional to the CPUs' computing power. We plan to perform a bottleneck analysis of the relay data path to identify the overhead of relevant functions in both the user and kernel spaces. **MM**

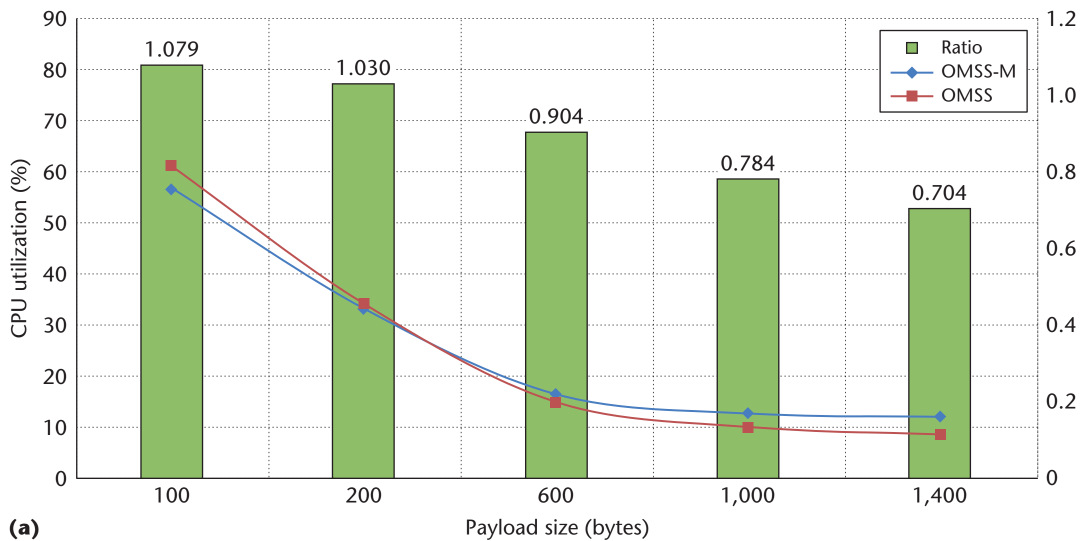
Acknowledgments

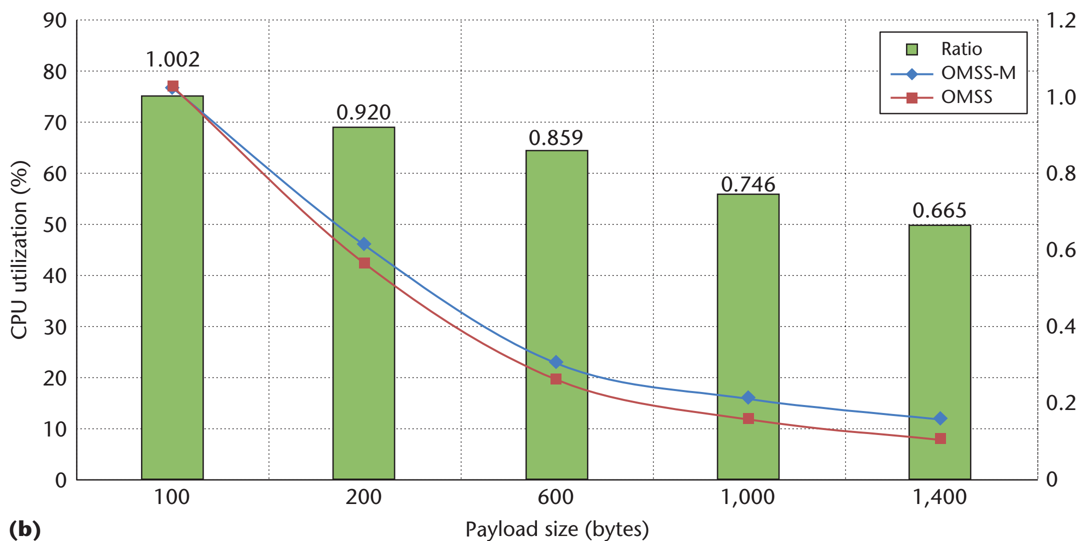
This work was supported in part by Institute for Information Industry under the project of embedded software and living service platform and technology development, which was subsidized by the Ministry of Economy Affairs, Taiwan. It was also supported in part by D-Link Corp. and Realtek Seminconductor Corp.

References

1. C. Diot et al., "Deployment Issues for the IP Multicast Service and Architecture," *IEEE Network*, vol. 14, no. 1, 2000, pp. 78–88.
2. H. Deshpande, M. Bawa, and H. Garcia-Molina, "Streaming Live Media over a Peer-to-Peer Network," tech. report, Stanford InfoLab, 2001.

Figure 7. CPU utilization of the payload-size test. (a) UDP streams. (b) TCP streams.





3. M. Castro et al., "SCRIBE: A Large-Scale and Decentralized Application-Level Multicast Infrastructure," *IEEE J. Selected Areas in Comm.,* vol. 20, no. 8, 2002, pp. 1489–1499.
4. Y. Chu et al., "A Case for End System Multicast," *IEEE J. Selected Areas in Comm.,* vol. 20, no. 8, 2002, pp. 1456–1471.
5. N.P. Venkata et al., "Distributing Streaming Media Content Using Cooperative Networking," *Proc. 12th Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video,* ACM, 2002, pp. 177–186.
6. D. Peter and L.P. Larry, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," *Proc. 14th ACM Symp. Operating Systems Principles* (SOSP), ACM, 1993, pp. 189–202.
7. J. Pasquale, E. Anderson, and P.K. Muller, "Container Shipping: Operating System Support for I/O-Intensive Applications," *Computer,* vol. 27, no. 3, 1994, pp. 84–93.
8. A.K. Yousef and N.T. Moti, "An Efficient Zero-Copy I/O Framework for UNIX," tech. report, Sun Microsystems, 1995.
9. V.S. Pai, P. Druschel, and W. Zwaenepoel, "IO-Lite: A Unified I/O Buffering and Caching System," *ACM Trans. Computer Systems,* vol. 18, no. 1, 2000, pp. 37–66.
10. D.A. Maltza and P. Bhagwata, "TCP Splice for Application Layer Proxy Performance," *J. High Speed Networks,* vol. 8, no. 3, 1999, pp. 225–240.
11. J. Kong and K. Schwan, "KStreams: Kernel Support for Efficient Data Streaming in Proxy Servers," *Proc. Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video* (NOSSDAV), ACM, 2005, pp. 159–164.

Ying-Dar Lin is professor of computer science, founder and director of the Network Benchmarking Lab (www.nbl.org.tw), and founder of the Embedded Benchmarking Lab (www.ebl.org.tw) at National Chiao Tung University. His research interests include design, analysis, implementation, and benchmarking of network protocols and algorithms; quality of service; network security; deep-packet inspection; P2P networking; and embedded hardware/software co-design. Lin has a PhD in computer science from the University of California, Los Angeles. He is an IEEE fellow and on the editorial boards of *IEEE Transactions on Computers, Computer, IEEE Network, IEEE Communications Magazine Network Testing Series, IEEE Communications Surveys* and *Tutorials, IEEE Communications Letters, Computer Communications, Computer Networks,* and *IEICE Transactions on Information and Systems*. Contact him at ydlin@cs.nctu.edu.tw.

Chia-Yu Ku is a doctoral student and a technical manager in the Broadband Mobile Lab at National Chiao Tung University, Taiwan. His research interests include wireless mesh networking, multichannel multiradio, mobility management, integration of wireless networks, and performance evaluation. Ku has an MS in mathematics and computer science from National Chiao Tung University. Contact him at cyku@cs.nctu.edu.tw.

Yuan-Cheng Lai is a professor in the Department of Information Management at the National Taiwan University of Science and Technology. His research interests include wireless networks, network performance evaluation, network security, and Internet applications. Lai has a PhD in computer science from National Chiao Tung University. Contact him at laiyc@cs.ntust.edu.tw.

Chia-Fon Hung performed this research while at National Chiao Tung University. He is now an engineer at Realtek Semiconductor. His research interests include embedded system design, multimedia applications, and performance evaluation. Hung has an MS in computer science from National Chiao Tung University. Contact him at cfhung@cs.nctu.edu.tw.

cn *Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.*

NEW TRANSACTIONS NEWSLETTER!

Stay connected with the IEEE Computer Society Transactions by signing up for our new Transactions Connection newsletter. It is free and contains valuable information like:

- News about your favorite transactions,
- Contributions from the Editorial Board,
- Information about related conferences,
- Multimedia,
- And much more.

Not a subscriber? Don't worry. You can still sign up to receive news about the transactions.

Visit http://www.computer.org/newsletters to sign up today!