國立交通大學

資訊科學與工程研究所

碩 士 論 文

內 核 的 一 對 多 串 流 轉 送 技 術

In-kernel Relay for One-to-Many Streaming

研 究 生：洪家鋒

指導教授：林盈達　教授

中 華 民 國　九 十 九 年 六 月

內核的一對多串流轉送技術

In-kernel Relay for One-to-Many Streaming

研 究 生：洪家鋒　　Student：Chia-Feng Hung

指導教授：林盈達　　Advisor：Ying-Dar Lin

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2010

Hsinchu, Taiwan

中華民國九十九年六月

內核的一對多串流轉送技術

學生：洪家鋒　　指導教授：林盈達

國立交通大學資訊科學與工程研究所

摘要

在使用應用層群播技術的系統中，代理伺服器需要同時服務大量的串流轉送連線並形成效能瓶頸的潛在因素。使用有效率的資料轉送路徑可以增加系統的最大服務量，同時減少系統計算能力的需求。本論文提出了一個叫作一對多串流疊合(OMSS)的內核一對多串流轉送服務。OMSS 藉由避免封包內容的複製來增加資料路徑效率，並且藉由使用線程池設計來減少切換執行緒的負擔。此外，藉由優先權工作排程技術來達成無關協定的服務品質保證。個人電腦平台上的實驗結果顯示，在 UDP 的流量下，OMSS 因為將轉送路徑移到內核中以及使用封包內容共用技術而分別減少了 41%及 15%的系統負載。在 TCP 的流量下則分別減少了 37%及 15%。與在用戶空間實現一對多轉送相比，系統的最大服務量可以達到兩倍。此外，即使是系統開始產生過載現象的時候，本服務可以確保高優先權轉送連線的服務品質。

關鍵字：內核，串流，群播，加速

In-kernel Relay for One-to-Many Streaming

Student: Chia-Feng Hung Advisor: Dr. Ying-Dar Lin

Institutes of Computer Science and Engineering

National Chiao Tung University

Abstract

Rendezvous nodes such as proxy relay servers may deal with a large amount of concurrent sessions and potentially cause a bottleneck. An efficient relay data path can increase the service capability and reduce the required computing power. In this paper, we propose an in-kernel one-to-many relay solution, called One-to-Many Streaming Splicing (OMSS), to reduce relay overheads. An in-kernel relay data path is realized by a payload sharing mechanism and a worker pool processing model to reduce memory copies and context switch overheads. Moreover, a priority-based task scheduling is applied to achieve differentiated service. The experimental results on PC platform demonstrate that OMSS reduces 41% and 15% CPU utilization by the in-kernel implementation and the payload sharing mechanism, respectively, for UDP traffic, while 37% and 15% CPU utilization is reduced by the two mechanisms for TCP traffic. The service capability is doubled in comparison with the original daemon solution. In addition, the QoS of high priority sessions can be guaranteed by the priority-based task scheduling when a system starts to be overloaded.

Keywords: kernel, streaming, multicast, acceleration

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Applications of streaming media over Internet become popular due to the wide-deployed broadband infrastructure and the rapid development of media compression techniques. For one-to-many streaming applications such as Internet radios and IPTVs, thousands of media consumers may access a popular channel simultaneously. Because of the insufficient outbound bandwidth and the limited computing power, a media provider cannot afford the fan-out burdens for such a large number of media consumers.

A media provider can distribute the fan-out burdens with two kinds of techniques with different implementation levels. The first one is IP multicast [1]. A media provider distributes the burdens to intermediate routers and only sends one copy of the media stream. However, IP multicast might be unavailable on the Internet level due to the problems of group management, address allocation and security [2]. As an alternative solution, overlay multicast [3-6] realizes the multicast capability with multiple unicasts. Overlay multicast organizes peers and dedicated relay servers into an overlay topology, instead of intermediate routers in a physical topology, to forward media streams. Although overlay multicast performs less efficient than IP multicast due to its unicast nature, the ease of deployment makes overlay multicast become the mainstream of data delivery and streaming applications over Internet.

Although overlay multicast helps to disperse the fan-out burdens to the involved nodes, the performance of a single node is still a key factor especially for: 1) Overloaded rendezvous nodes such as proxy relay servers or super peers, which forward a large number of streaming sessions and the fan-out burdens is large. 2) Nodes with limited computing power due to the hardware capability or the system policies. Therefore improving the performance of a critical node enhances the service

capability and the required computing power is smaller under the same relay loads.

Media stream forwarding suffers the overheads such as memory copies and system calls. Previous works [7-10] use a system-wide unified buffer structure with page-remapping and shared memory mechanisms to prevent cross domain memory copies. However, they aggressively modify the underlying buffer structure of the operating system. [11, 12] perform relay operations through the user space socket API against altering the underlying buffer structure. But the system call and the memory copy from user space to the kernel still occur. [15, 16], which intercept packets in the IP layer and use the header-altering method to prevent memory copies and system calls, support only one-to-one relay schemes and cause more maintenance overheads when transport layer protocol evolves. [13] builds an in-kernel streaming relay data path on top of I/O subsystems to prevent system calls and memory copies between the user space and the kernel space. Nevertheless, memory copies of media streams are still required for each sink connection.

This work proposes and implements One-to-Many Streaming Splicing (OMSS), an in-kernel streaming relay service in Linux-based systems. OMSS provides a one-to-many streaming relay mechanism which supports both UDP-based and TCP-based streaming. By payload sharing mechanism, OMSS minimizes the number of memory copies to improve one-to-many streaming relay performance and can achieve large-scale service capability. In addition, a differentiated QoS is provided to ensure the service quality of high priority relay sessions. The performance of OMSS is evaluated in both general PC and embedded platforms.

The remainder of this thesis is organized as follows. In Chapter 2, we take an overview of a one-to-many streaming system and then focus on the data path implementation to point out the overheads. Chapter 3 describes the design of OMSS and Chapter 4 is the implementation in Linux-based system. Chapter 5 illustrates the

evaluation environment and presents the evaluation results. Conclusion and some future work are in Chapter 6.

Chapter 2 Background & Related Works

In this chapter, we first take an overview of a one-to-many streaming system and address the situation of rendezvous nodes such as proxy relay servers to bring up the need of an efficient relay data path. A survey and comparison of relay data path solution are presented to point out the overhead of current relay data path solutions.

2.1. Overview of One-to-Many Streaming Systems

According to the deployment policy, three kinds of one-to-many streaming systems exist: 1) peer-to-peer, 2) proxy-based and 3) hybrid. A peer-to-peer system benefits from the low cost of deployment. A media provider multicasts its media stream with the assistance of the overlay network formed by peers. Nevertheless, the system is unstable because of the high variation of peers. Proxy-based systems such as traditional content delivery networks (CDNs) use dedicated proxy relay servers to provide more bandwidth and stability than a peer-to-peer system. However, it suffers high deployment costs. For a CDN service provider, a hybrid solution using proxy relay servers and super peers as the backbone could reach the balance and is a realistic solution.

Figure 1 shows the topology of a hybrid one-to-many streaming system with two streaming sessions. Once a peer wants to publish or subscribe a streaming session, it connects to the proxy relay server which services the designated domain it belongs to. If a media provider and its media consumers are not in the same designated domain, an inter-proxy-relay connection is setup to forward media streams. With stability and enough outbound bandwidth, a peer can become a super peer to alleviate the loads of proxy relay servers.

In such a system, proxy relay servers are responsible to a large amount of fan-out burdens and service a large number of streaming sessions simultaneously, while super

peers usually have limited computing power due to the hardware capability or system policies. For these nodes, an efficient streaming relay data path can enhance the single-node performance to provide large service capability while the bandwidth is sufficient or to reduce the computing power requirement under the same relay loads.

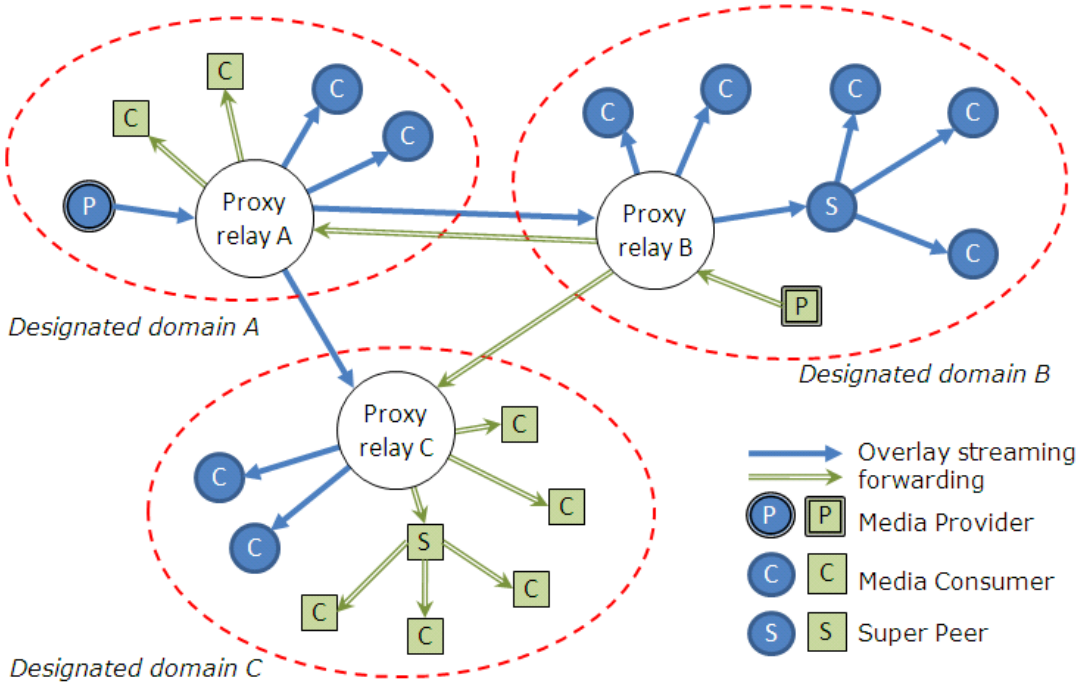


Figure 1: One-to-many streaming system topology with two streaming sessions.

2.2. Relay Data Path Solutions

A relay data path continuously forwards media streams from the source to the sink(s). The media streams passing through different data access interfaces incur different kinds of overheads. In summary, relay data paths can be classified into three kinds of solutions as: solutions with socket API, solutions on IP layer hooks and solutions on top of I/O subsystems.

Solutions with Socket API

Solutions with socket API relay media streams in the user space through the socket interface. After the incoming media streams go through the network stack, they are stored in the receiving queue of the source socket within the kernel space. The

relay mechanism copies media streams to the user space buffer and then passes them to the sending queue of the sink sockets inside the kernel space for further delivery. Because media streams are copied across the user-kernel boundary, this kind of solutions incurs a large amount of system calls and memory copies. Icecast [11] and DSS [12] are the solutions using the socket API to implement one-to-many streaming relay. The former is capable of TCP streaming whereas the latter considers UDP streaming.

Solutions on IP Layer Hooks

An in-kernel relay data path can prevent the overheads of copying data across the user-kernel boundary. To build an in-kernel relay data path, IP layer hooks such as hooks provided by netfilter [17] in linux-based systems allow kernel modules to register callback functions within the IP layer. Relaying media streams on IP layer hooks prevents media streams from copied across the user-kernel boundary. Because the media streams are not passed through the whole network stack, the tasks of transport layer, especially TCP, should be handled by the relay solution. Therefore the maintenance cost is high when the transport layer protocol evolves. TCPSP [15] and Media Proxy [16] both build relay mechanisms on IP layer hooks with the header-altering method, supporting only one-to-one relay scheme.

Solutions on top of I/O subsystems

Another kind of in-kernel relay data path solution is to build relay mechanisms on top of I/O subsystems. This kind of solutions not only prevents the overheads of copying data across the user-kernel boundary but also suffers less maintenance overheads when the transport layer protocol evolves. KStreams [13] makes a data abstraction layer on top of I/O subsystems. By the data abstraction layer, KStreams can relay media streams between different kinds of I/O subsystems and supports one-to-many relay schemes. However, the memory copy from the source to the sinks

is still needed.

Summary

According to the data access interface, a relay data path suffers different kinds of overheads. Table 1 shows a comparison among different relay data path solutions. In-kernel relay data paths such as the solutions on IP layer hooks and the solutions on top of I/O subsystems do not suffer the crossing user-kernel overheads and perform better than the solutions with socket API. Because of the lack of transport layer protocol handling, the solutions on IP layer hooks need to handle transport layer protocol by themselves. Hence the additional maintenance overheads are caused. Although the solutions on top of I/O subsystems are better than the other solutions, they still need in-kernel memory copies to sinks. The memory copy degrades the performance of relay, especially in the one-to-many relay scheme.

Table 1: Relay data path solutions.

Data access interface	Solution	Relay scheme	Protocol	Cons/Overheads
Socket API	Icecast [11]	1-to-N 1 way	TCP	1+N memory copy 1+N system call
	DSS [12]	1-to-N 1 way	UDP	
IP layer hooks	TCPSP [15]	1-to-1 2 way	TCP	L4 Protocol handling by self – hard to maintain
	Media Proxy [16]	1-to-1 1 way	UDP	
On top of I/O subsystem	KStreams [13]	Configurable 1 way	Protocol independent	1+N memory copy in 1-to-N scheme

Chapter 3 One-to-Many Streaming Splicing

The design of OMSS is to provide an in-kernel one-to-many relay data path for streaming applications. The three objectives of OMSS are: 1) reducing the overhead caused by payload copy among sink connections. 2) Achieving large-scale service capability. 3) Support differentiated QoS to guarantee the service quality of high priority relay sessions. The architecture of OMSS is presented first. And the design of each component is shown one after another.

3.1. OMSS Architecture

The architecture of OMSS is divided into two parts: the one-to-many relay data path and the relay engine as shown in Figure 2. A one-to-many relay data path consists of one source connection, multiple sink connections and a session buffer. A source connection is a connection from which OMSS receives media streams while a sink connection is a connection where the media streams are sent to the subscribers. The media streams received from the source connection are stored in the session buffer. When the session buffer is full, the *head-drop* policy is applied. OMSS proposes a *payload sharing* mechanism to reduce the memory copies of media streams for sink connections.

The reception and transmission of the media stream within a relay session are treated as a *source task* and multiple *sink tasks* respectively. All tasks are scheduled and handled by the relay engine. The relay engine adopts the worker pool processing model to achieve large-scale service capability and a priority-based task scheduling to provide the QoS guarantees for high priority relay sessions. Triggered by the socket events, application requests or task processing results, source and sink tasks are queued in the class queues and waiting to be fetched and assigned by the scheduler to the worker pool. A worker thread within the worker pool processes the selected task

and queues the further tasks generated by the selected task. For example, a source task queues the relevant sink tasks after storing the media stream in the session buffer for further transmissions to the subscribers.

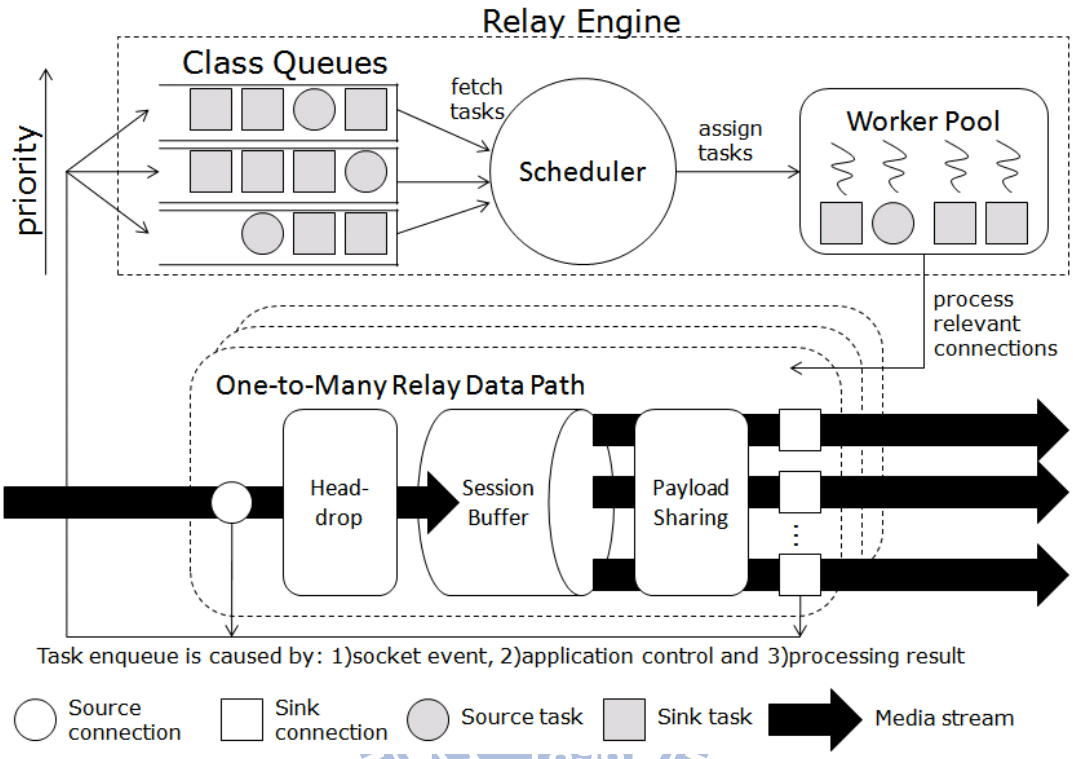


Figure 2: Architecture of OMSS

3.2. One-to-Many Relay Data Path

Head-drop Buffer Management

Due to the different connection status of each sink connection, the sending progress varies. A slower sink connection may occupy buffer entries for a long time and then cause the session buffer full. Algorithms of buffer management such as tail-drop or random early detection (RED) drop the newest media stream. Therefore all the sinks of that session miss the dropped media stream. In this work, a head drop buffer management, which skips the transmission of the occupied buffer entries only for slower sink connections, is applied to release the buffer entries when the session buffer is full.

Payload Sharing

In a relay session, the same media stream is forwarded from the source connection to multiple sink connections. In previous works [11-13], the transmission of the same media streams needs to make duplications for each sink connection as shown in Figure 3(a). Given a 1-to-N relay session, N memory copies are required. In this work, the payload sharing is achieved with the help of network interface card (NIC) which is capable of scatter-gather I/O. A NIC with scatter-gather I/O can directly transmit a frame whose contents are located in discontinuous memory regions. As shown in Figure 3(b), after extracting the transport layer payload from incoming frames, OMSS reuses the payload for all the sink connections without duplicating the payload.

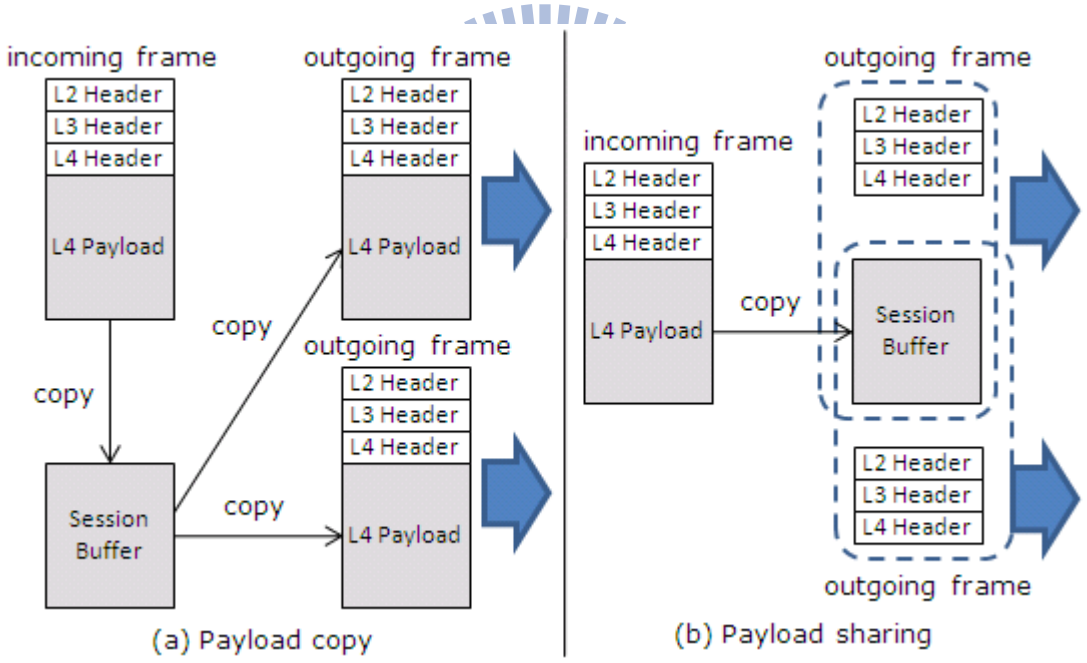


Figure 3: Payload copy vs. payload sharing

3.3. Relay Engine

Worker Pool Processing Model

A task processing model is the type of process or thread model used to handle the service operations. There are two common task processing models: 1) single-thread model adopted by [13] and 2) per-session thread model adopted by [11]. The former

cannot utilize the computing power of a shared memory multiple processors (SMP) systems, whereas the latter might incur large context switch overhead and memory consumption of thread stacks when the session number grows. To achieve large-scale service capability, OMSS divides the relay operation into two kinds of tasks: source task and sink task and adopts the worker pool processing model used by [12] to handle these tasks. A worker pool consists of a set of pre-spawned threads and a task assigning interface, which assigns a given task to one of the idle threads in the worker pool to handle. Because the threads are pre-spawned, there is no thread-creation overhead at runtime. In addition, because all the tasks are handled by the threads in the pool and the pool size is fixed, the context switch overhead is limited. Finally, the computing power of SMP system can be utilized because of the multi-threaded nature of worker pool processing model.

Figure 4 shows the flow chart of source tasks and sink tasks. When a source task is handled by a worker thread of the worker pool, it checks the session buffer first. If

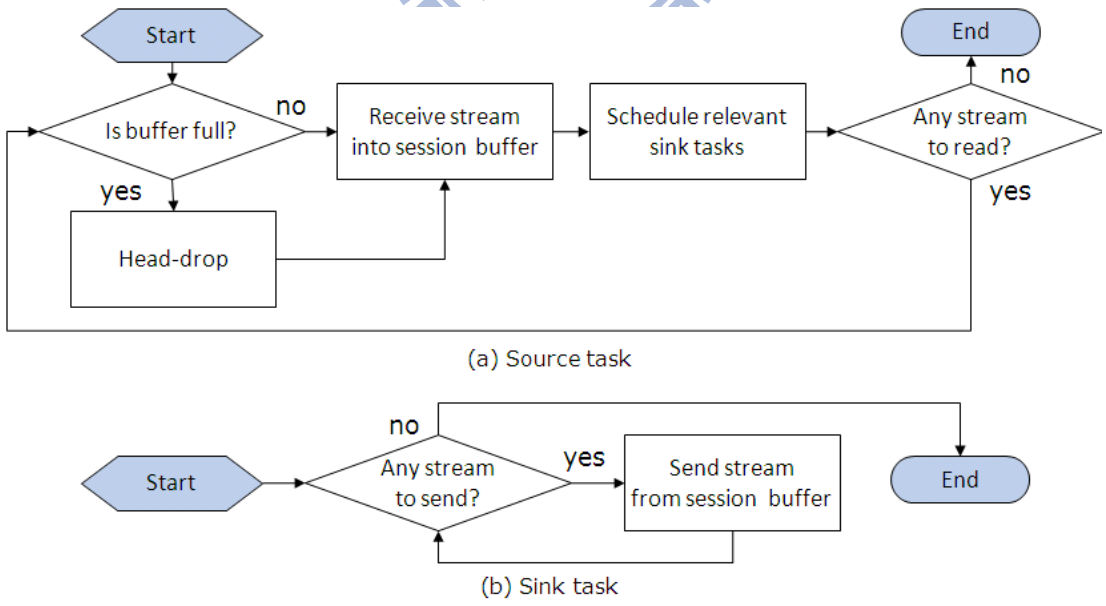


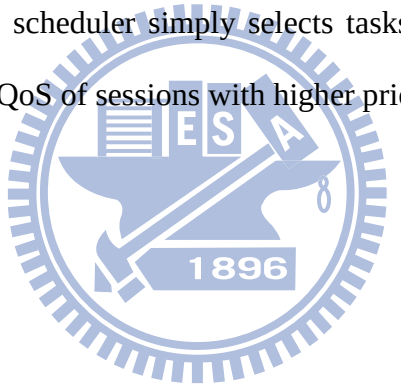
Figure 4: Flow chart of source task and sink task

the session buffer is full, the head-drop buffer management mechanism is applied. After receiving media streams into its session buffer, the source task schedule sink

tasks in the same relay session to send media streams to every sink connection. The procedure repeats to handle the remaining media streams until there is no media stream to read in the source connection. When a sink task is handled, it repeats send media streams to its sink connection until there is no media stream for its sink connection to send.

Priority-Based Task Scheduling

OMSS provides a priority-based scheduling mainly to achieve a differentiated QoS mechanism to guarantee the service quality of high priority relay sessions. A priority value is assigned to a connection when the connection is added into a relay session. The task of the connection is then queued into the class queue according to the priority value. And the scheduler simply selects tasks from queues with higher priority value to ensure the QoS of sessions with higher priority connections.



Chapter 4 System Implementation

4.1. Implementation Overview

OMSS is implemented in linux-based systems as an in-kernel service to provide an efficient one-to-many streaming forwarding data path and an application programming interface (API) to interact with user space relay daemon as illustrated in Figure 5. A user space relay daemon uses the OMSS API to organize in-kernel relay sessions and to get the status of ongoing in-kernel relay sessions. In the rest of this chapter, we detail the implementation of OMSS components.

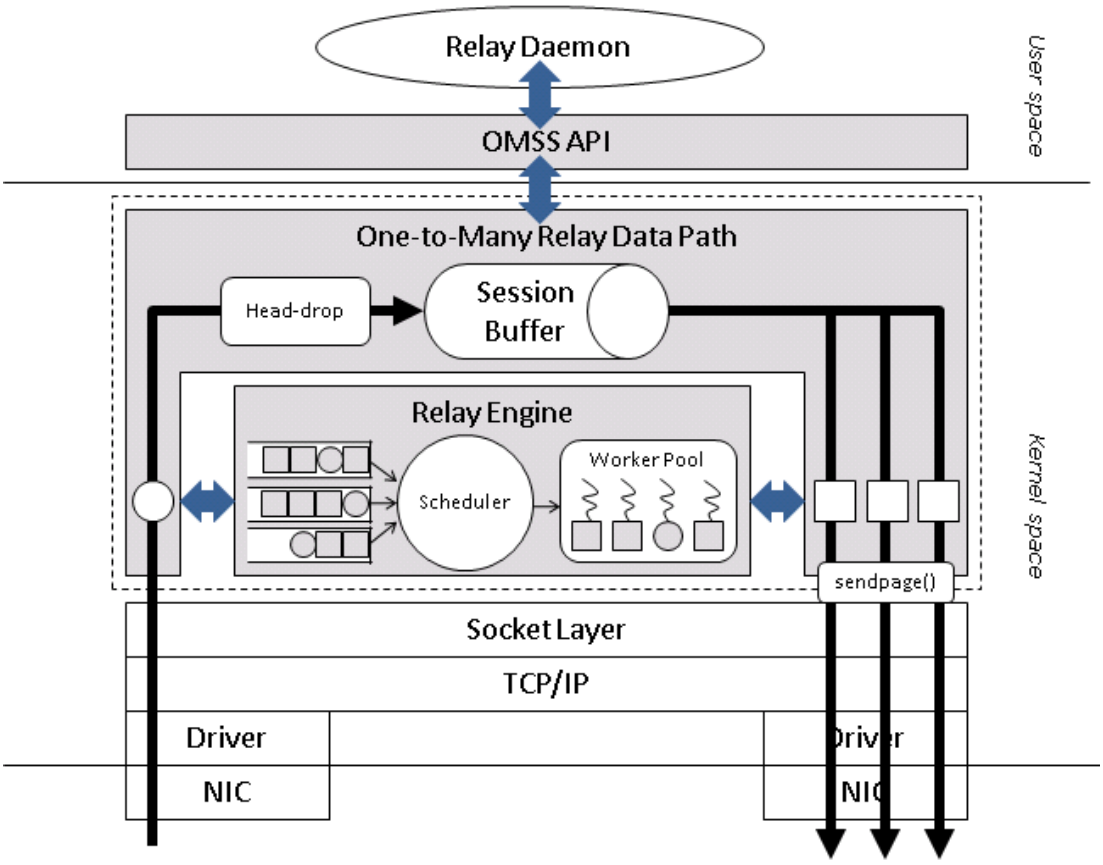


Figure 5: OMSS Implementation in Linux-based System

4.2. One-to-Many Relay Data Path

The one-to-many relay data path has the advantage of protocol independence by building on top of the socket layer. Both UDP and TCP sockets can be organized as a

session source or sink. Media streams come from a session source is stored in the session buffer and then sent with the payload sharing mechanism. The implementation of the session buffer and the payload sharing mechanism is illustrated in the following.

Session Buffer

The session buffer is implemented as a ring buffer whose buffer entries have a *user count* to indicate if the entry is needed by sink tasks and a pointer of the memory page where media streams is really stored. The source task and sink tasks also keep the buffer entry they should access. Figure 6 shows an 8-entry session buffer of a 1-to-5 relay session. In Figure 6(a), a buffer is full due to the slower sink task. In Figure 6(b), the head-drop mechanism is done by decreasing the user counts of all buffer entries with the number of slower sink tasks (one, in this case).

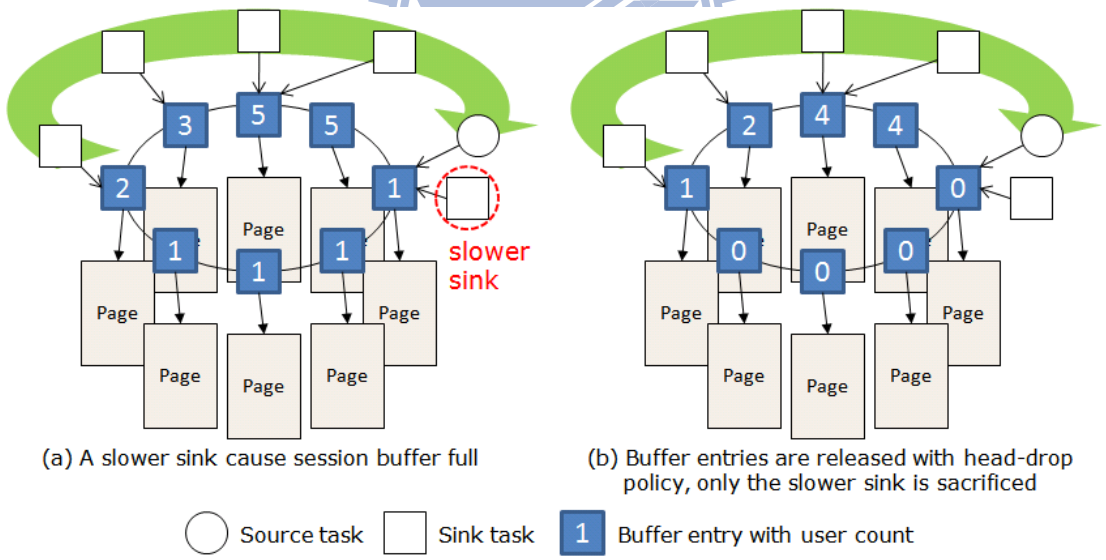


Figure 6: An 8-entry session buffer of a 1-to-5 relay session.

Payload Sharing

In linux-based systems, a packet is represented by the *sk_buff* structure and its auxiliary data structures. A scatter-gather I/O is performed while parts of frame contents are stored in the memory pages which are appended to the *frags* array of a

sk_buff structure. Figure 7 illustrates two sk_buff with a shared transport layer payload. The first frags entry of each sk_buff is used to point to the shared payload. There is a socket layer function called *sendpage*, which can send the payload stored in memory pages. By sendpage(), payload sharing in linux-based systems is as simple as the following two steps: 1) extracting the payload from the incoming packet into a memory page and 2) send out the payload to each sink connection with sendpage(). That is the reason why we use memory pages to implement the session buffer.

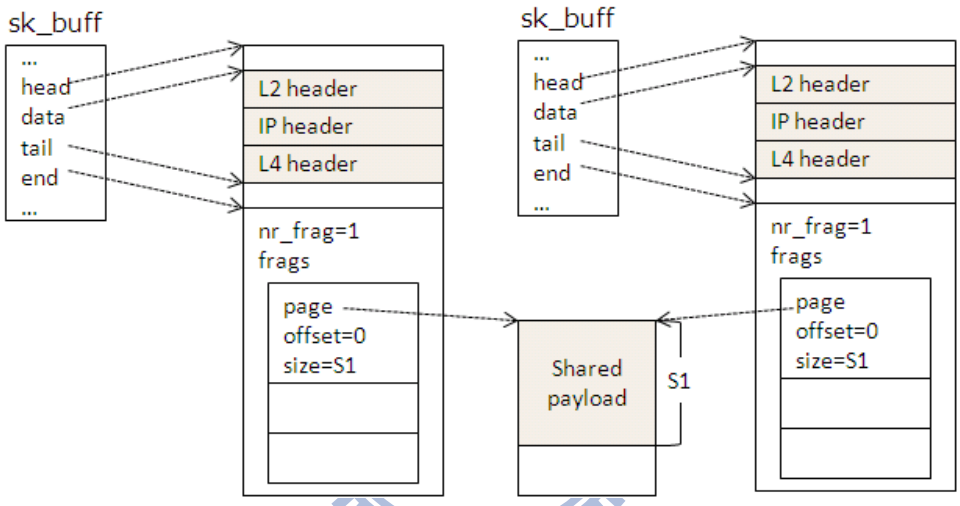


Figure 7: Two sk_buff with a shared payload.

4.3. Relay Engine

The relay operation is triggered by the socket event of a source task when incoming packets arrive. The source task is queued in the class queue according to its priority value. After the source task is pick up by the scheduler and processed by one of the worker threads in the worker pool, sink tasks of this session are queued and handled.

Worker Pool

The worker pool implementation contains a common task assigning interface and a set of worker threads which are implemented by kernel threads. Initially, each worker thread sleeps on a wait queue until a task is assigned through the task

assigning interface. Given a task, the task assigning interface chooses an idle worker thread and assigns the task to the worker thread. At that time, the worker thread is woken up and executes the assigned task.

Scheduler and Class Queues

The scheduler is implemented as a kernel thread which repeats getting task objects to be handled through the dequeue function provided by the class queues. After getting a task from the class queues, the scheduler assigns the task to the worker pool through the task assigning interface mentioned above. If there is no task object in class queues, the scheduler thread sleeps for 10 ms and then continues calling the dequeue function to get task objects to be handled.

4.4. OMSS API

OMSS provides an interface to organize in-kernel one-to-many relay sessions and to get the status of session members by adding four socket options. A user space program can call the setsockopt() system call with the four socket options listed in table 2 to interact with the OMSS module.

Table 2: The OMSS socket options and functionalities

Socket option	Functionality
OMSS_SET_SOURCE	Setup a relay session with the specified source socket.
OMSS_ADD_SINK	Add a sink socket to a specified relay session.
OMSS_REMOVE_SINK	Remove a sink socket from the specified relay session.
OMSS_GET_STATUS	Get the status of the specified source/sink socket.

For example, after the connection from media source is setup, a relay daemon can call setsockopt() system call with the OMSS_SET_SOURCE option to setup a relay session and set this connection as the session source. And then a relay daemon uses the OMSS_ADD_SINK option to add session sinks. To check the status of each session member, the OMSS_GET_STATUS option is used. If an error or timeout is

reported, a relay daemon can use the OMSS_REMOVE_SINK option to remove the session member which is out of function.

Chapter 5 Evaluation

5.1. Evaluation Environments

The evaluation environments are shown in Figure 8. The media server (MS) and the media client (MC) are in the same PC, while the relay in another PC. Streaming flows is transmitted through a direct Ethernet link from MS to relay and back to MC in the same PC. In the evaluation environment for relay-PC, a controller is connected to these two PCs through an out-of-band link for separating the control flow from streaming flows, while a controller is connected to the relay-embedded through a console line instead of an Ethernet connection. Table 3 shows the details of the evaluation platforms. The relay-PC is equipped with AMD Athlon 64 X2 CPU and Intel 82574L gigabit adaptor which is capable of scatter-gather I/O for the payload sharing mechanism. And the relay-embedded is equipped with RTL8186 SoC which has a MIPS processor and a fast Ethernet interface which is capable of scatter-gather I/O. The direct link throughputs of these two evaluation environments are 941 Mbps and 40 Mbps separately, which are measured with Iperf.

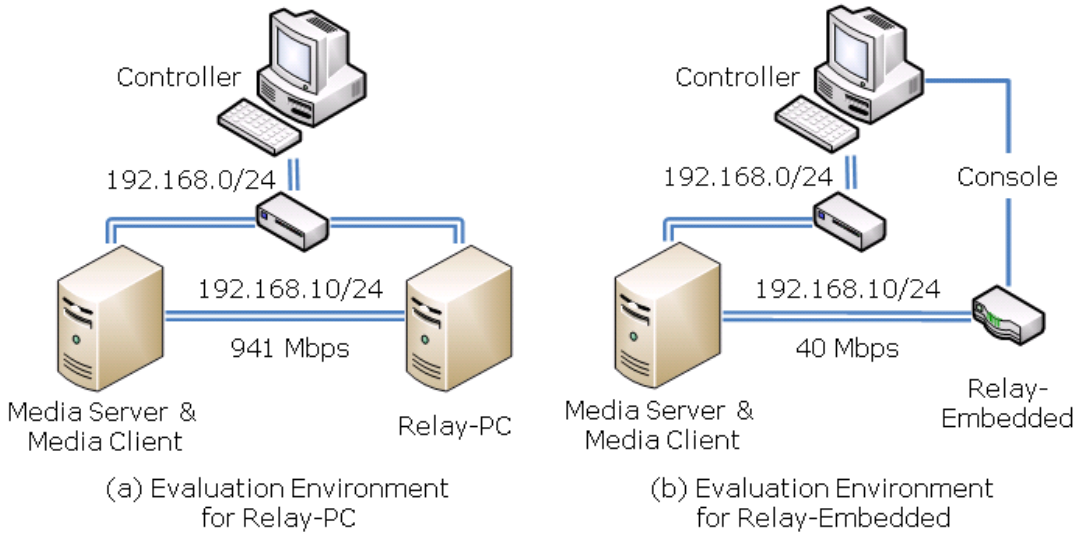


Figure 8: Evaluation environments.

Table 3: Evaluation Platforms

	Media Server & Client	Relay-PC	Relay-embedded
Kernel	2.6.32.4	2.6.32.4	2.4.18
CPU	Intel Core2 Duo E8400 @3GHz	AMD Athlon 64 X2 4000+ @2.1GHz	MIPS @180MHz
Memory	2G	2G	16M
Ethernet Interface	Marvell 88E8056 Gigabit adaptor	Intel 82574L Gigabit adaptor	RTL8168 Ethernet

The MS program is a multithreaded program which can emulate multiple media sources to transmit streaming flows with different payload sizes and transmission rates. The MC program is also a multithreaded program to emulate multiple MCs to subscribe streaming flows and report statistics such as packet loss rates and transmission latency. Three kinds of relay solutions can be requested by the MS at session setup time: daemon, OMSS and OMSS-M. The daemon is a user space one-to-many relay solution, which uses two threads, one for receive and one for send, to serve one relay session. The reception and transmission of streaming flows is through the socket API. OMSS and its memory copy version, OMSS-M, are integrated into this daemon as alternative relay solutions.

5.2. Bulk Fan-out Test

In this test, the CPU utilization of the relay-PC and the relay-embedded is measured for the three relay solutions. Only one relay session is setup by the MS and then the number of MCs is added till the maximum link capacity or CPU utilization is reached. The streaming source is 256 Kbps with the payload size of 1400 Bytes for both UDP and TCP.

Bulk Fan-out Test on PC

Figure 9 shows the CPU utilization of the UDP bulk fan-out test on PC. After reaching about 60% CPU utilization, the daemon solution cannot utilize the SMP

computing power to serve more MCs than 2200 because it has only a single sending thread. Because of reducing both the system call overhead and the memory copies to each sink connection, OMSS outperforms the daemon solution and OMSS-M.

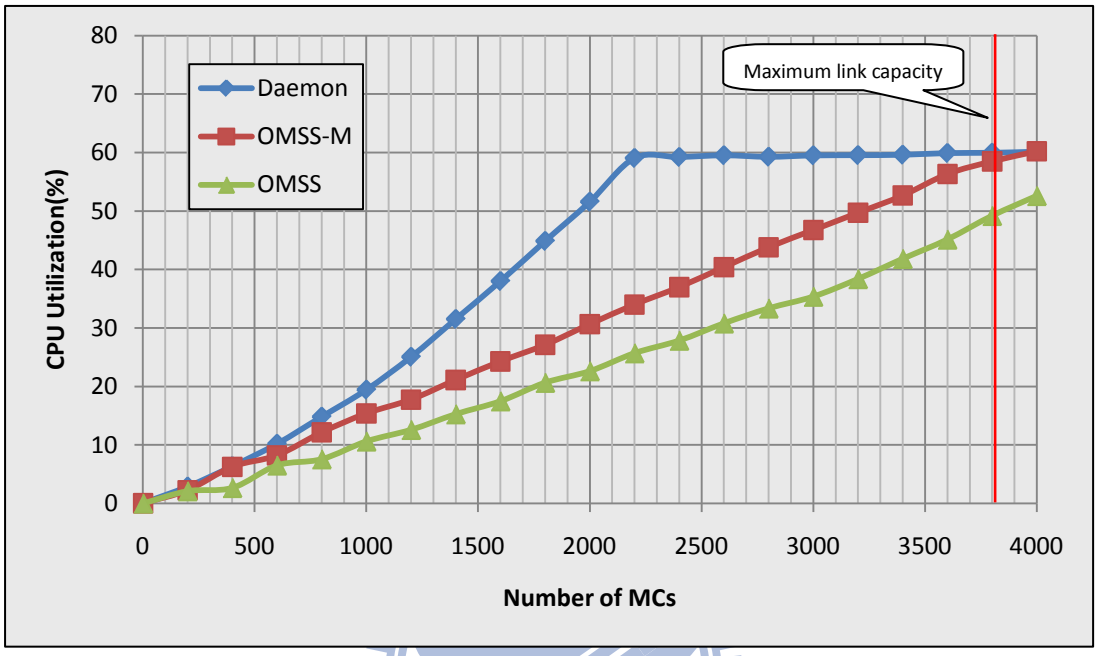


Figure 9: CPU utilization of UDP bulk fan-out test on PC.

The result of TCP bulk fan-out test is shown in Figure 10. The daemon solution suffers the same situation mentioned above after reaching about 70% CPU utilization and can serve at most 2000 MCs. OMSS also outperforms the other two in this test. After the throughput reaches maximum link capacity, packets is dropped by the NIC. Therefore, TCP starts retransmitting and cause the CPU utilization of OMSS and OMSS-M to vibrate.

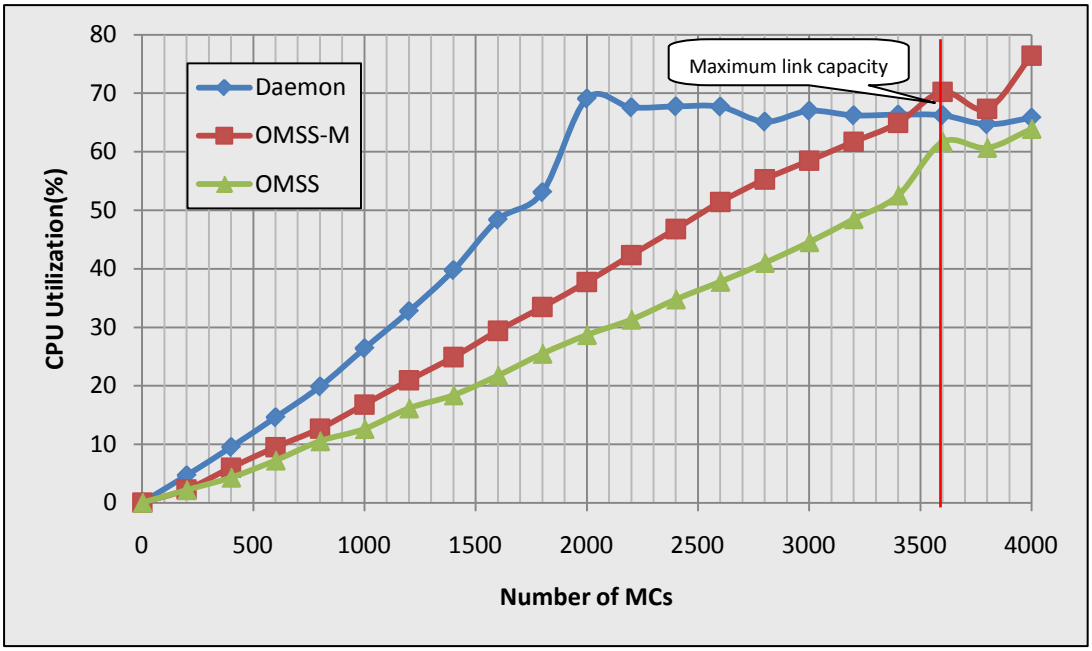


Figure 10: CPU utilization of TCP bulk fan-out test on PC.

Bulk Fan-out Test on Embedded Platform

Because there is no UDP sendpage implementation in the linux 2.4 kernel, we can do only TCP bulk fan-out test on the relay-embedded. Figure 11 shows the CPU utilization of the TCP bulk fan-out test on the embedded platform. The daemon solution, OMSS-M and OMSS service up to 45, 80 and 85 MCs separately. OMSS still outperforms the other two in this test.

According to the evaluation results on these two platforms, we find that the capability of the relay is not proportional to the computing powers of the CPUs. This phenomenon is caused by the following reasons. First, the computation of checksum is done by software on the embedded platform, while offloaded to the NIC on the PC platform. Second, the embedded platform uses linux 2.4 kernel, while the PC platform uses linux 2.6 kernel. Finally, the hardware architectures are different. We leave this question as a future work to do.

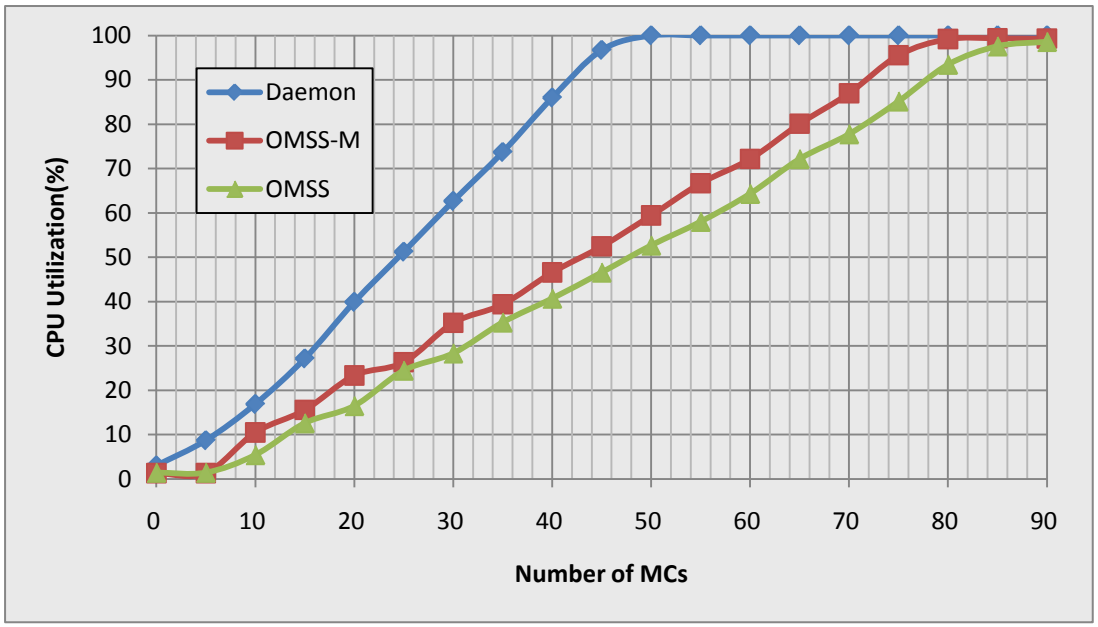


Figure 11: CPU utilization of TCP bulk fan-out test on embedded platform.

5.3. Payload Size Test

The payload sharing mechanism prevents the copy of payloads for sink connections. Therefore the payload size determines the avoided amount of memory copy. In this test, given a 1-to-800 relay session with a 256 Kbps streaming source, we evaluate how the payload size of the streaming flow affects the performance of the payload sharing mechanism on UDP and TCP. Figure 12 and Figure 13 show the testing results for UDP and TCP, respectively. Switching points are revealed at about 200 Bytes for UDP and 100 Bytes for TCP. When the payload is transmitted, the payload sharing mechanism needs to append memory pages into sk_buff structures and occupy two TX descriptors for DMA transfers than the original transmission process. Below the switching point, the memory copy overheads are smaller than the payload sharing overheads mentioned above. As the payload size grows, the memory copy overhead also grows. Therefore, the performance gain of the payload sharing is proportional to the payload size.

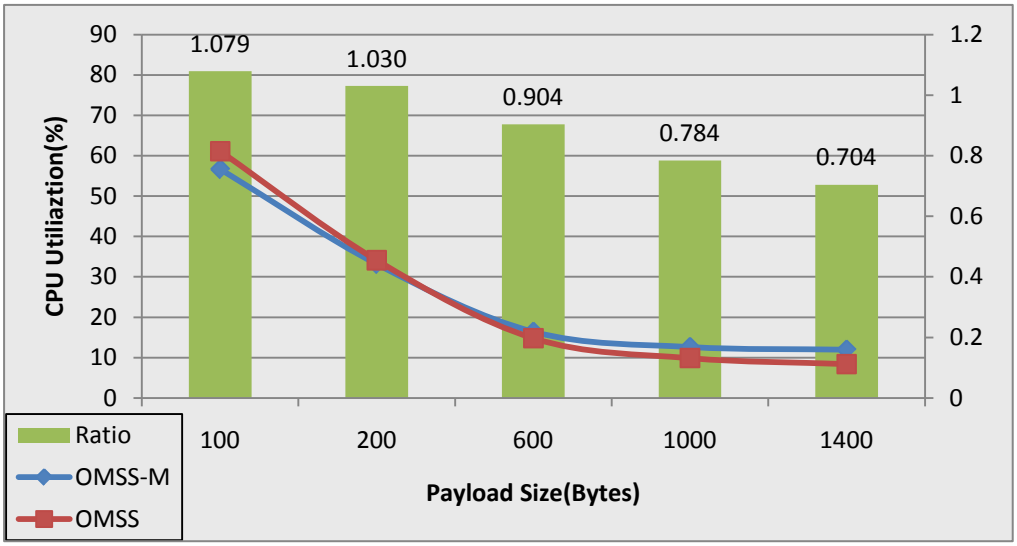


Figure 12: Payload size test for UDP.

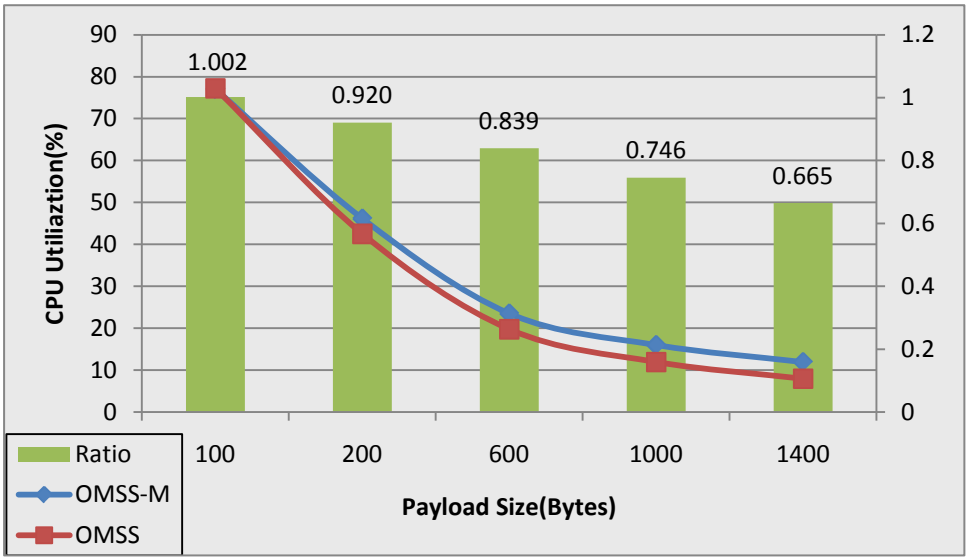


Figure 13: Payload size test for TCP

5.4. Overloading Test

In this test, three 1-to-150 relay sessions with 256 Kbps UDP streaming source is assigned to the priority classes from 1 to 3. The CPU utilization of the relay PC is 72.04% at the beginning. To see the affect of priority-based task scheduling, we control the background CPU loads at the relay PC and then observe the packet loss rates and average latencies of these three sessions. The background CPU loads is produced by the processes which are scheduled by the FIFO scheduling policy with

the highest priority. One process produces 5% CPU loads and totally 20 processes are executed to pile up the background CPU loads to 100%. Because the throughput does not reach the maximum link capacity in this test, packet losses are caused by the head-drop buffer management when the session buffer is full.

Figure 14 shows the average packet loss rates of the three classes under different background CPU loads. Before the background CPU loads run out of the remaining CPU resource, only the class 3 session suffers very slight packet losses. Once the CPU is exhausted, the packet loss rate of class 3 session increases rapidly. The high-priority sessions experiences almost no packet losses before the packet loss rate of low-priority sessions reach 1. Finally, all the three session lose the streaming packets after the whole CPU resource is occupied by the background CPU loads.

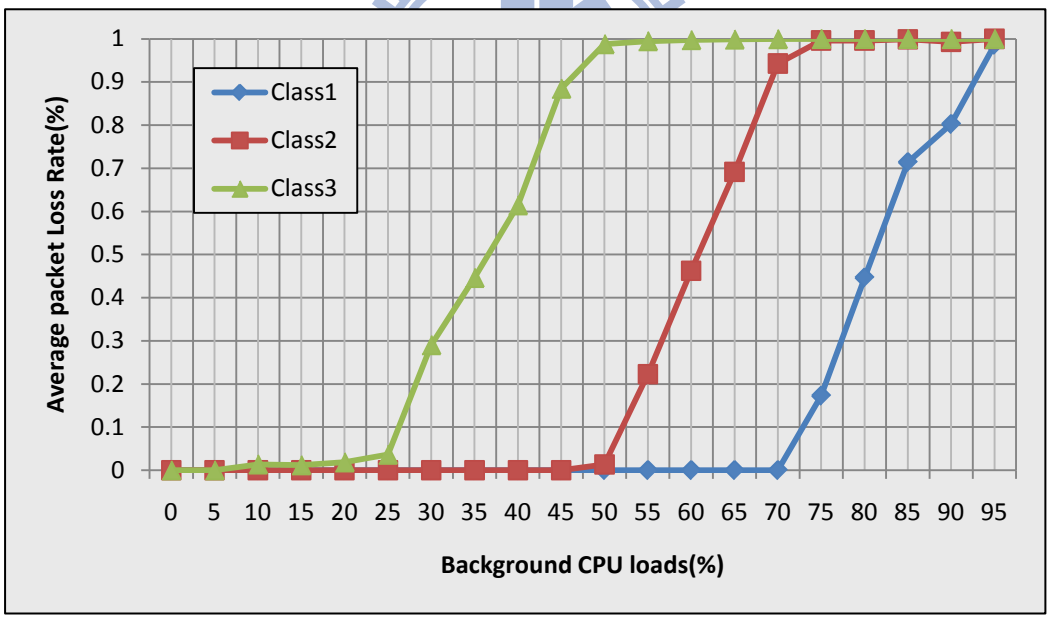


Figure 14: Average packet loss rate under overloaded test

The result of average latency is shown in Figure 15. At the beginning, the three sessions suffer different levels of average latency. And the average latency of a certain priority class increases rapidly while the lower-priority sessions have lost all their packets. These phenomena happen because tasks with lower priorities are handled only when no task with higher priority value exists in class queues. Therefore

the scheduler always sacrifices tasks form the lowest priority class.

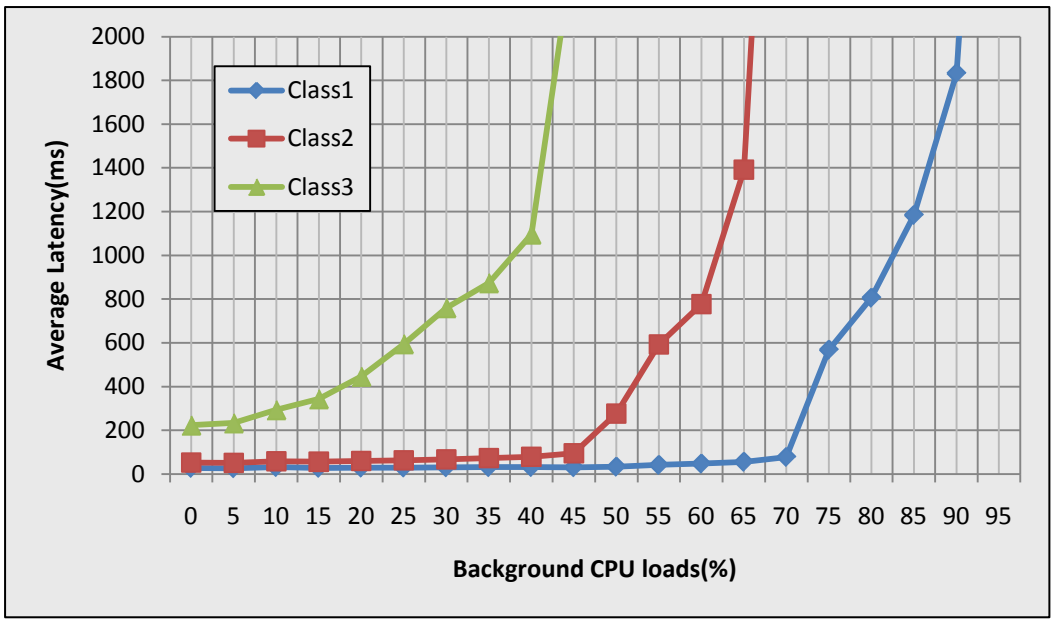


Figure 15: Average latency under overloaded test

Chapter 6 Conclusions and Future Works

In this work, we propose an in-kernel one-to-many relay service called One-to-Many Streaming Splicing (OMSS) which has three features. First, a low-overhead one-to-many relay data path is carried out with the payload sharing mechanism which prevents the payload copies to each sink connection. Second, a large-scale service capability is achieved by dividing the relay operation into source tasks and sink tasks and adopting the worker pool processing model to handle these tasks. Finally, a differentiated QoS is realized with the priority-based task scheduling.

The evaluations of 256 Kbps streaming source with 1400 Bytes payload size show that OMSS reduces the CPU utilization by 56% and 26% of that the daemon solution and OMSS-M, respectively, for UDP in 1-to-2000 relay session; 52% and 24% for TCP in 1-to-1800 relay session. In addition, the priority-based task scheduling succeeds to ensure the service quality of high priority relay sessions on the overloaded situations.

In future works, because the payload sharing mechanism is now realized by scatter-gather I/O and some low-end platforms may be not equipped with NICs performing scatter-gather I/O, we intend to design a payload sharing mechanism without scatter-gather I/O. Therefore, even on those low-end platforms, the relay data path overheads caused by memory copies can also be reduced.

References

[1] S. E. Deering and D. R. Cheriton, “Multicast routing in datagram internetworks and extended LANs,” *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85-110, 1990.

[2] C. Diot, B. N. Levine, B. Lyles H. Kassem and D. Balensiefen, “Deployment issues for the IP multicast service and architecture,” *IEEE Network*, vol. 14, no. 1, pp. 78-88, 2000.

[3] H. Deshpande, M. Bawa, and H. Garcia-Molina, “Streaming Live Media over a Peer-to-Peer Network,” Technical Report, Stanford InfoLab, 2001.

[4] M. Castro, P. Druschel, A. M. Kermarrec and A. I. T. Rowstron, “SCRIBE: A large-scale and decentralized application-level multicast infrastructure,” *IEEE Journal on Selected Areas in communications*, vol. 20, no. 8, pp. 1489-1499, 2002.

[5] Y. Chu, S. G. Rao, S. Seshan and H. Zhang, “A case for end system multicast,” *IEEE Journal on Selected Areas in communications*, vol. 20, no. 8, pp. 1456-1471, 2002.

[6] N. P. Venkata, J. W. Helen, A. C. Philip and S. Kunwadee, “Distributing streaming media content using cooperative networking,” in *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, Miami, Florida, USA, 2002.

[7] D. Peter, and L. P. Larry, “Fbufs: a high-bandwidth cross-domain transfer facility,” in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, Asheville, North Carolina, United States, 1993.

[8] J. Pasquale, E. Anderson, and P. K. Muller, “Container shipping: operating system support for I/O-intensive applications,” *IEEE Computer*, vol. 27, no. 3, pp. 84-93, 1994.

[9] A. K. Yousef, and N. T. Moti, “An Efficient Zero-Copy I/O Framework for UNIX,” Technical Report, Sun Microsystems, Inc., 1995.

[10] V. S. Pai, P. Druschel, and W. Zwaenepoel, “IO-Lite: A unified I/O buffering and caching system,” *ACM Transactions on Computer Systems*, vol. 18, no. 1, pp. 37-66, 2000.

[11] Xiph open source community, Icecast, http://www.icecast.org/, 2008.

[12] Apple Inc., Darwin Streaming Server, http://dss.macosforge.org/, 2008.

[13] J. Kong and K. Schwan, “KStreams: kernel support for efficient data streaming in proxy servers,” in *Proceedings of the international workshop on Network and operating systems support for digital audio and video*, pp. 159-164, Stevenson, Washington, USA, 2005.

[14] D. A. Maltza, and P. Bhagwata, “TCP Splice for application layer proxy performance,” *Journal of High Speed Networks*, vol. 8, no. 3, pp. 225-240, 1999.

[15] LVS project, TCPSP Software – an open source TCP splicing implementation, http://www.linuxvirtualserver.org/software/tcpsp/index.html, 2003

[16] AG Project, MeidaProxy, http://mediaproxy.ag-projects.com/, 2010.

[17] The netfilter.org project, http://www.netfilter.org/.