This paper was presented as part of the 14th IEEE Global Internet Symposium (GI) 2011 at IEEE INFOCOM 2011

Augment SCTP Multi-Streaming with Pluggable Scheduling

Yaogong Wang, Injong Rhee
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
Email: {ywang15, rhee}@ncsu.edu

Sangtae Ha
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
Email: sangtaeh@princeton.edu

***Abstract*—Stream Control Transmission Protocol (SCTP) introduces the multi-streaming feature to avoid the head-of-line (HOL) blocking problem of TCP and facilitate the transport of signaling messages over the Internet. However, the current protocol specification does not define how multiple streams should be scheduled and implementations are different across platforms. They typically choose either round-robin or first-come-first-served. Such fixed choices may not satisfy the demands of different applications and limit the flexibility of the protocol. In this paper, we propose to augment the multi-streaming feature of SCTP with pluggable scheduling so that users can customize the multi-stream scheduling algorithm of SCTP to satisfy their application-specific demands. We implement our proposal in the Linux kernel and show that this extension greatly increases the flexibility of SCTP and brings visible performance enhancements. This is the first work that demonstrates the effectiveness of pluggable SCTP multi-stream scheduling through real implementations and testbed experiments. Our proposal requires modifications only on the sender side and incurs no interoperability or fairness problems. Hence, it is safe and convenient to be deployed in the current Internet.**

I. Introduction

SCTP [1] is a transport protocol originally designed to carry telephony signaling messages over the Internet. It offers a number of features not available in traditional TCP and UDP, such as multi-streaming and multi-homing. Thanks to these attractive features, SCTP gradually evolves to be a general-purpose transport protocol capable of broad applications.

Multi-streaming, where one association can bundle multiple independent streams, is one of the most important features of SCTP. It avoids the head-of-line (HOL) blocking problem of TCP and makes SCTP a proper transport for signaling messages. In SCTP, a stream is a *unidirectional* logical channel established from one to another associated SCTP endpoint. In-order delivery of user messages is retained within each stream (unless unordered delivery service is requested) but not across multiple streams. However, reliable data transfer and congestion control is applied to the entire association. To accomplish these requirements, SCTP separates data transmission and data delivery. Specifically, each DATA chunk has two independent sequence numbers: a per-association Transmission Sequence Number (TSN) and a per-stream Stream Sequence Number (SSN). TSN is used for data transmission including loss recovery, flow control and congestion control while SSN (along with Stream ID) is used for data delivery. Fig. 1 illustrates this procedure.

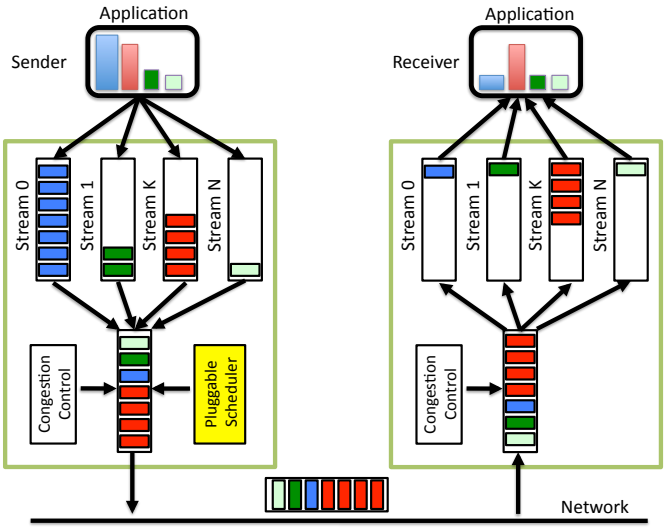


Fig. 1. SCTP multi-streaming and our pluggable scheduler: SCTP supports multiple streams within an association. It ensures independently sequenced delivery among different streams while maintaining appropriate congestion control behavior over the entire association. The pluggable scheduler may, for example, assign higher priority to Stream K.

Although SCTP supports multi-streaming, the current standard [1] does not specify how to schedule them, or more specifically, how to assign sequential TSNs to DATA chunks from different streams. The errata to the standard [2] briefly touches upon this topic and suggests a simple round-robin scheduling algorithm over all streams with pending data. This approach is adopted by FreeBSD. Linux and Solaris instead use first-come-first-served (FCFS) as the multi-stream scheduling algorithm in their protocol stack. In spite of the specific choices made by different implementations, they all share the "one size fits all" philosophy. This approach does not fully exploit the benefits provided by multi-streaming. In this paper, we propose to augment it by providing an infrastructure on which 1) scheduling algorithms can be developed in a systematic way, 2) a scheduling algorithm can be loaded/unloaded at run time, and 3) applications can easily configure the algorithms through a standard socket interface.

978-1-4244-9920-5/11/$26.00 ©2011 IEEE

Fig. 2. A sample application: Alice is chatting with Bob via an instant messenger. She also wants to share some photos with Bob. SCTP fits into this application scenario since it can open two streams under one association and use separate stream to transfer separate data. However, prioritized treatment is desirable in this case since Alice does not want her messages to be delayed by the photo transmission.

The motivation behind this proposal is that, as the application has created multiple streams, it's very likely that the data in each stream have different characteristics and need separate treatment. Imposing a round-robin scheduling algorithm to let all streams equally share the available resources may not meet the demands of an application. A better way is to let the applications choose and configure the multi-stream scheduling algorithm by themselves so as to best fit their requirements.

For example (Fig. 2), two users are chatting with each other via an instant messenger. Meantime, a user is sending a photo to the other user. This is a realistic scenario since most instant messengers today provide the functionality to transfer files between two users when both are online. In this scenario, there are two simultaneous connections between the users: one for instant messages and the other for file transfer. SCTP fits into this application scenario well since it eliminates the need to maintain two separate connections. Instead, the users need only one SCTP association with two streams. The problem is that, with a fixed scheduling algorithm such as FCFS, it is very likely that the file data will constantly occupy the output queue and thus chat messages have to be queued after them, since file transfer is typically continuous bulk data while chat messages are typically short and intermittent. As a result, the latency experienced by the instant messages is unpredictable. Assigning higher priority to the instant messenger over the file transfer is strongly desired, but this is not viable with the current SCTP implementation.

The previous example is a relatively simple use case. As applications with more complicated requirements emerge, the flexibility in multi-stream scheduling will be more important. Our proposal is to augment SCTP with pluggable multi-stream scheduling to fulfill application-specific requirements. This extension substantially improves the utility of SCTP and brings visible performance improvement with little cost. Note that our proposal does not change the protocol on the wire. It only changes the internal scheduling algorithm of the SCTP sender. Therefore, it won't cause any interoperability problem with existing implementations. The only thing it may affect is bundling, an optional feature that carries multiple small user messages in one SCTP packet, since how messages are bundled depends on their sequence. [3] discusses this issue and proposes per packet scheduling. As will be shown later, our pluggable scheduling framework is general enough for the user to implement such scheduling algorithms or their equivalents. Also note that this proposal does not change the congestion control algorithm of SCTP. It only changes the order in which new DATA chunks are sent, under the constraints of the given congestion window. Therefore, it is safe to be deployed in the Internet.

The rest of the paper is organized as follows. Section II details the design and implementation of our proposal. In Section III, we demonstrate the effectiveness of the proposed extension via real implementation in the Linux kernel and quantify its performance through testbed experiments. We then discuss the related work in Section IV and conclude in Section V.

II. Design and Implementation

A. Design Considerations

To allow SCTP applications to customize the multi-stream scheduling algorithm, we can implement several common scheduling algorithms in the SCTP protocol stack and provide an API for the applications to choose and configure the scheduling algorithm they want. The advantage of this approach is its ease of use since the applications only need to add a few lines of code to call the API. But it is less flexible because the applications can only choose from the given set of scheduling algorithms. If the application wants a specific scheduling algorithm that is not provided in the set, the demand of the application cannot be met.

Another approach is to provide a general interface within the protocol stack to let application programmers implement the scheduling algorithm by themselves. In fact, a SCTP multi-stream scheduling algorithm is just a different manipulation of the DATA chunks in the output queue. If we expose the queue manipulation functions (enqueue, dequeue, etc.) to the users, they can implement whatever special algorithm they want. However, the disadvantage of this approach is that it burdens every application programmer with the responsibility to implement a scheduling algorithm for their application.

Our design is a combination of these two approaches. We expose the manipulation functions for the DATA chunk output queues so that users can fully customize the multi-stream scheduling algorithm if necessary. In addition, we utilize this kernel interface to implement several common algorithms including FCFS, strict priority queue and weighted fair queue. These algorithms serve two purposes: 1) they demonstrate the usage of the kernel interface so that application programmers can consult them while implementing their own scheduling algorithm. 2) Application programmers can directly use them if they already meet the requirements of the application.

By default, FCFS is used so that legacy SCTP applications can operate on the new stack without any modification. If the applications need to use some common scheduling algorithm that is already provided, they can simply call the API to choose that algorithm. If an application has some special requirements and needs to adopt a customized scheduling algorithm, the user can implement it via the provided kernel interface. Thus, we maintain the balance between ease of use and maximum flexibility.

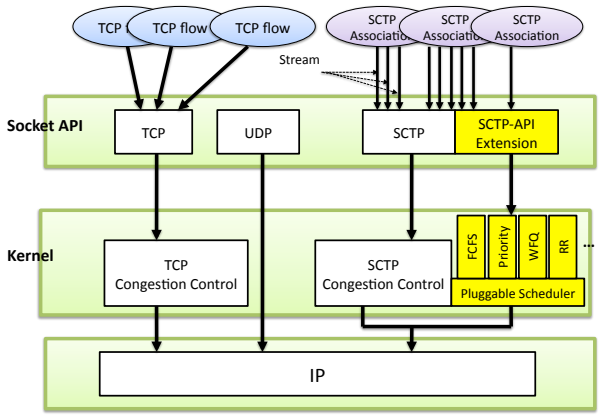


Fig. 3. Architecture of our implementation in the Linux kernel.

B. Implementation in the Linux Kernel

We have implemented our proposal in the Linux kernel. The kernel patches can be found at [4] and the architecture is shown in Fig 3. The implementation consists of three parts: 1) a pluggable scheduler, which is the infrastructure in the kernel providing a general interface for users to implement customized multi-stream scheduling algorithms as loadable kernel modules; 2) a set of kernel modules implementing several commonly used scheduling algorithms; 3) an extension of SCTP socket API providing the interface for applications to access and configure the scheduling algorithm.

More precisely, we define the *sctp_sched_ops* structure (see Fig. 4) which includes a group of hooks to functions used for queue manipulation operations on the DATA chunk output queue. To implement a custom multi-stream scheduling algorithm, a user needs to write a kernel module that defines those queue manipulation functions. Then declare a *sctp_sched_ops* structure and set its pointers to these functions. Finally, register the structure in the kernel via *sctp_register_sched* function so that applications can use this new scheduling algorithm. We implemented three common scheduling algorithms: FCFS, strict priority queue and weighted fair queue. Fig. 5 shows the implementation of the default FCFS scheduling algorithm.

We also define a new socket option for SCTP so that applications can easily use our extension. It can be set or retrieved via the standard *setsockopt*/*getsockopt* functions. The *level* and *optname* arguments should be *IPPROTO_SCTP* and *SCTP_SCHED* respectively. The *optval* argument should be a *sctp_sched* structure as defined in Fig. 6. *ssched_name* is a descriptive name of the scheduling algorithm (same as the *name* member of *sctp_sched_ops* structure). This might be enough for a simple scheduling algorithm like FCFS. But for more complicated scheduling algorithms (e.g. priority queue), the application needs to not only choose a scheduling algorithm but also configure it (e.g. set the priority of each stream). This is done via *ssched_config*. Since different scheduling algorithms may have different definitions/requirements for the configuration and its length may vary (e.g. depending on the number of outgoing streams), we provide *ssched_config* as a place holder (a zero-sized array) and use *ssched_config_len*

```
struct sctp_sched_ops {
    struct list_head list;
    char name[SCTP_SCHED_NAME_MAX];
    struct module *owner;

    int (*init)(struct sctp_outq *q, gfp_t gfp);
    void (*release)(struct sctp_outq *q);
    void (*enqueue_head_data)(struct sctp_outq *q, struct sctp_chunk *ch);
    void (*enqueue_tail_data)(struct sctp_outq *q, struct sctp_chunk *ch);
    struct sctp_chunk* (*dequeue_data)(struct sctp_outq *q);
    int (*is_empty)(struct sctp_outq *q);
};
extern int sctp_register_sched(struct sctp_sched_ops *type);
extern void sctp_unregister_sched(struct sctp_sched_ops *type);
```

Fig. 4. *sctp_sched_ops* structure: the kernel interface to implement customized scheduling algorithms

```
/* Initialize the DATA chunk output queue. One queue is enough for FCFS. */
/* More complex algorithms may need separate queues for different streams. */
static int fcfs_init(struct sctp_outq *q, gfp_t gfp) {
    q->out_chunk_list = kmalloc(sizeof(struct list_head), gfp);
    if (!q->out_chunk_list)
        return -ENOMEM;
    INIT_LIST_HEAD(q->out_chunk_list);
    return 0;
}

static void fcfs_release(struct sctp_outq *q) {
    kfree(q->out_chunk_list);
}

static void fcfs_enqueue_head_data(struct sctp_outq *q, struct sctp_chunk *ch) {
    list_add(&ch->list, q->out_chunk_list);
    q->out_qlen += ch->skb->len;
}
static void fcfs_enqueue_tail_data(struct sctp_outq *q, struct sctp_chunk *ch) {
    list_add_tail(&ch->list, q->out_chunk_list);
    q->out_qlen += ch->skb->len;
}

/* Always dequeue from the head */
static struct sctp_chunk *fcfs_dequeue_data(struct sctp_outq *q) {
    struct sctp_chunk *ch = NULL;
    if (!list_empty(q->out_chunk_list)) {
        struct list_head *entry = q->out_chunk_list->next;
        ch = list_entry(entry, struct sctp_chunk, list);
        list_del_init(entry);
        q->out_qlen -= ch->skb->len;
    }
    return ch;
}

/* Emptiness test, necessary when having multiple queues */
static inline int fcfs_is_empty(struct sctp_outq *q) {
    return list_empty(q->out_chunk_list);
}

struct sctp_sched_ops sctp_fcfs = {
    .name = "fcfs",
    .owner = THIS_MODULE,
    .init = fcfs_init,
    .release = fcfs_release,
    .enqueue_head_data = fcfs_enqueue_head_data,
    .enqueue_tail_data = fcfs_enqueue_tail_data,
    .dequeue_data = fcfs_dequeue_data,
    .is_empty = fcfs_is_empty,
};
```

Fig. 5. Sample implementation of FCFS scheduling algorithm using our kernel interface

to indicate the length of the custom configuration in bytes. Basically, the configuration of the scheduling algorithm is passed as a opaque block of memory trailing the *sctp_sched* structure and the interpretation of this configuration is dependent on the specific scheduling algorithm chosen by the user.

To customize the multi-stream scheduling algorithm, the ap-

```
struct sctp_sched {
    char ssched_name[SCTP_SCHED_NAME_MAX];
    __u16 ssched_config_len;
    __u16 ssched_config[0];
};
```

Fig. 6. *sctp_sched* structure: the socket API to configure the scheduling algorithm

```
int sk, ss_sz;
struct sctp_initmsg si;
struct sctp_sched *ss;

// create a one-to-many style SCTP socket
sk = socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);

// we need only two outgoing streams
bzero(&si, sizeof(si));
si.sinit_num_ostreams = 2;
setsockopt(sk, IPPROTO_SCTP, SCTP_INITMSG, &si, sizeof(si));

// configure priority scheduling
ss_sz = sizeof(struct sctp_sched) + 2 * sizeof(__u16);
ss = (struct sctp_sched *) malloc(ss_sz);
strncpy(ss->ssched_name, "prio", SCTP_SCHED_NAME_MAX);
ss->ssched_config_len = 2 * sizeof(__u16);
ss->ssched_config[0] = 1;
ss->ssched_config[1] = 2;
setsockopt(sk, IPPROTO_SCTP, SCTP_SCHED, ss, ss_sz);
free(ss);
```

Fig. 7. Code snippet for the scenario depicted in Fig. 2. Priority scheduling is used on two streams with Stream 0 assigned Priority 1 and Stream 1 assigned Priority 2. All error checking codes are eliminated to save space.

plication first creates an SCTP socket. Then it calls *setsockopt* on this socket to choose and configure the scheduling algorithm before initiating or accepting any association (see Fig. 7 for the sample code). Once the scheduling algorithm is configured, the program proceeds as usual: it may call *bind*, *listen*, *accept* or *sctp_recvmsg* if it acts as a server. Or it may call *connect* or *sctp_sendmsg* if it acts as a client.

A subtle implementation detail needs to be noted here. SCTP supports two socket styles: one-to-one style and one-to-many style [5]. The former controls only one association and is very similar to TCP socket. It aims to allow existing applications to be ported to SCTP easily but limits the use of some advanced features of SCTP (e.g. piggybacking data during association establishment). The latter may control multiple associations within one socket and has full support for the new features. With one-to-many style sockets, there is an important design choice on whether multi-stream scheduling should be set on a per socket basis or per association basis. The benefits of association-based scheduling are finer granularity of control and better knowledge of the association (e.g. the number of outgoing streams is known only when the association is actually established). But there are some technical challenges with this design. To configure the scheduling algorithm of an association, we must wait until the association is established. But since associations are implicitly established under one-to-many style socket and data transfer may get started on the third or fourth packet of the four-way handshake during SCTP association establishment, we inevitably need to change the scheduling algorithm when data transfer is already underway. This poses a number of problems and complications in the implementation. Therefore, we choose the simpler approach of socket-based scheduling. In this design, the multi-stream scheduling algorithm is set on a socket before the establishment of an association. Note that, at this point the user does not know the number of outgoing streams that will be actually negotiated. If it turns out to be smaller than the number proposed by the user, the scheduling algorithm will be applied in a truncated version. If the user changes the scheduling algorithm of a socket after some associations are already established, only associations established after the change are affected.

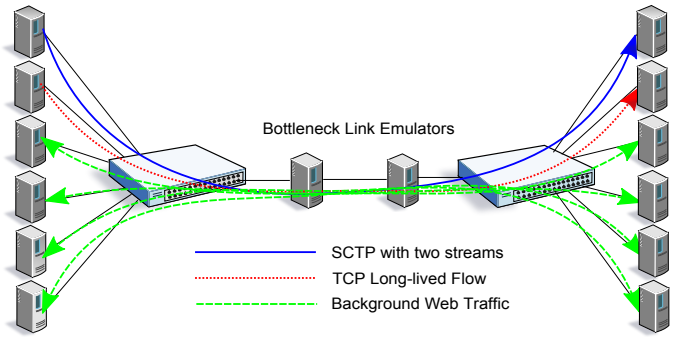
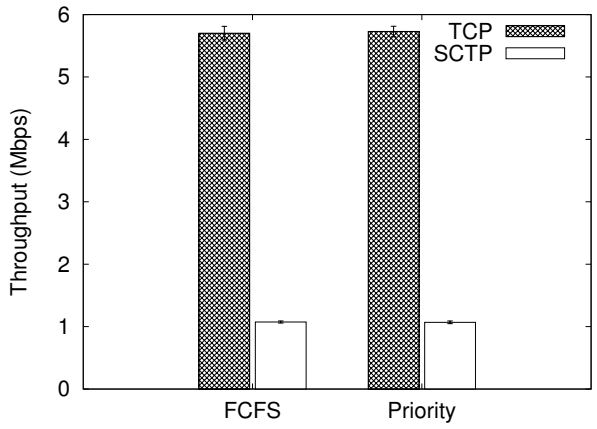


Fig. 8. Testbed Topology

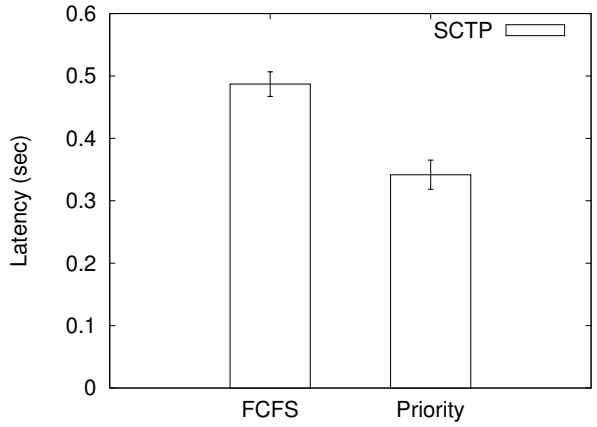
III. Experimental Evaluation

In this section, we show the proof-of-concept experimentation by running the sample application depicted in Fig. 2 on top of our implementation in the Linux kernel. Note that our extension provides the *mechanism* to customize the multi-stream scheduling algorithm of an SCTP association. It is the user or the application programmers who decide the *policy*, i.e. which scheduling algorithm best fits their demands. Our sender-side modification only provides quality of service (QoS) within the multiple outgoing streams of an association. The end-to-end QoS cannot be guaranteed unless network-assisted QoS architectures such as Diffserv [6] are deployed. But we will demonstrate through the following testbed experiments that, even a simple sender-side modification as our proposal is able to reduce the latency of the instant messages by up to 78% while preserving the same throughput for file transfer.

The topology of our testbed is shown in Fig. 8. It's a dumbbell topology with six sender/receiver pairs. One pair is used to emulate the Alice-and-Bob scenario in Fig. 2: the SCTP sender opens two streams to the SCTP receiver. On one stream, the sender sends intermittent instant messages whose sizes are uniformly distributed between 1 and 100 bytes. The inter-arrival time of the instant messages are exponentially distributed with a mean of 1 second. On the other stream, the sender sends bulk data to the receiver continuously. The other five sender/receiver pairs are used to generate background traffic in forward and reverse directions. One pair generates a long-lived TCP flow while the other four pairs generate short-lived requests/responses to emulate Web traffic. The two servers in the middle are used to emulate a symmetric

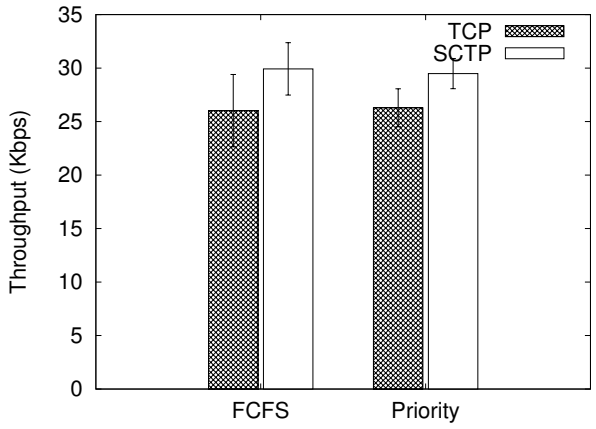


(a) TCP and SCTP Throughput Comparison

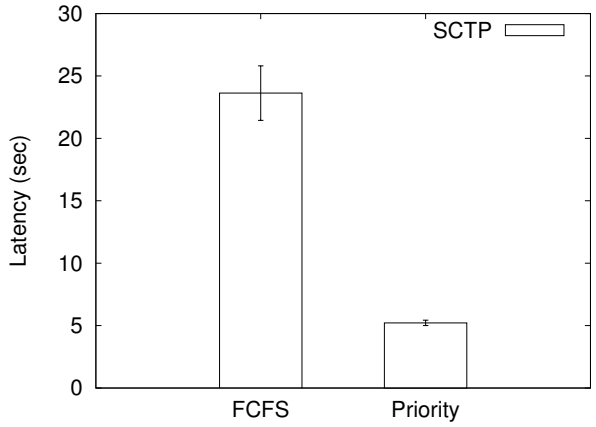


(b) Instant Message Latency Comparison

Fig. 9. Performance Comparison under 10Mbps Link



(a) TCP and SCTP Throughput Comparison



(b) Instant Message Latency Comparison

Fig. 10. Performance Comparison under 56Kbps Link

bottleneck link with configurable bandwidth, delay and buffer size. Each test lasts for 600 seconds and is repeated 10 times to calculate 95% confidence intervals of all metrics.

We first emulate the bottleneck link with a bandwidth of 10Mbps and an RTT of 300ms. The buffer size is set to the bandwidth delay product (BDP). Under this scenario, we measure 1) the average throughput of the SCTP flow and the long-lived TCP flow, 2) the average latency of the SCTP instant messages. We compare these metrics of two scheduling algorithms: FCFS and strict priority queue where instant messages are given higher priority than bulk data transfer. Fig. 9 shows the results.

As can be seen from Fig. 9(b), adopting strict priority queue as the multi-stream scheduling algorithm has improved the average latency of the instant messages by nearly 30%. Meanwhile, the average throughputs of the TCP flow and the SCTP flow remain the same despite the different scheduling algorithm being used (Fig. 9(a)). The reasons why TCP achieves higher throughput than SCTP in this scenario include: 1) SCTP uses CRC32c checksum while TCP uses simple 16-bit 1's compliment sum. 2) NIC supports checksum offload (CKO), segmentation offload (TSO) and large receive offload (LRO) for TCP but not SCTP. 3) SCTP reserves message boundaries while TCP is byte-stream-oriented. But this is beyond the scope of this paper. What we want to emphasize is that our extension does not change the congestion control behavior of SCTP and is safe to be deployed in the current Internet.

We run another similar test with different bottleneck link parameters. In this test, we try to emulate slow residential networks and set the bandwidth and RTT to 56Kbps and 300ms respectively. The buffer size is set to 40 packets, much larger than the BDP of the network, which is typical in broadband residential networks [7]. The results of this test is shown in Fig. 10. Under this scenario, the latency of the instant messages rockets up to around 23 seconds under FCFS. This is due to the low bandwidth as well as the large buffer size. With priority queue, the latency is decreased to about 5 seconds, which is a 78% improvement! Again, the average throughputs don't change much and SCTP achieves similar performance to TCP under this low bandwidth scenario.

We leave more experiments with different scheduling algorithms as our future work.

IV. Related Work

There are numerous applications that simultaneously transfer multiple types of data between the same source and destination. It's important for these applications to be able to allocate resources among the different data types according to their demands. Different solutions to this problem have been proposed in the context of TCP and SCTP.

In TCP, it's always possible to establish multiple connections between the same source and destination. But due to the congestion control algorithm of TCP, these simultaneous connections are most likely to converge to equal share of the available resources, which may be undesirable for the application. To provide service differentiation among these TCP connections, pTCP [8] strips data to multiple micro-flows (which exhibit the same behavior as normal TCP connections) and reassembles them at the receiver. By controlling the number of micro-flows, pTCP provides end-to-end service differentiation. MulTCP [9] achieves similar effects in a different manner: it manipulates the AIMD parameters of the TCP connection so as to obtain a proportional share of multiple TCP connections. The problems with these TCP-based approaches are two-fold: 1) Opening multiple connections and manipulating resource allocation at the application layer is a cumbersome burden for application programmers. 2) Modifying the congestion control behavior of TCP incurs fairness problems.

SCTP solves these two problems via its unique multi-streaming feature. Applications can have multiple streams within in a single association and their aggregate behavior conforms to TCP-friendliness. However, the current SCTP standard did not specify the multi-stream scheduling algorithm, making it difficult to provide service differentiation among the streams. To address this issue, SF-SCTP [10] groups SCTP streams into subflows and imposes independent flow and congestion control on each subflow. Service differentiation can then be implemented at subflow level. This approach may cause both interoperability and fairness issues since it changes the DATA chunk header of SCTP as well as its congestion control mechanism. [11] discusses the multi-stream scheduling issue of SCTP in the context of Concurrent Multipath Transfer (CMT). Through simulations the authors show that mapping each stream to a certain path achieves better performance than simple round-robin scheme. However, since CMT is not yet standardized in the current SCTP specification, this approach is not immediately deployable. [3] discusses the benefits of using different SCTP multi-stream scheduling algorithms under different scenarios and proposes per packet scheduling. But the idea is only validated by simulations. Our proposal is a light-weight, yet effective solution to the problem. No changes are made to the packet structure or congestion control behavior of SCTP, hence it's safe to be deployed in the current Internet. The closest idea to our proposal may be [12]. However, it only implements a single priority queue scheduling algorithm in the ns-2 SCTP module whereas our extension provides a general framework to implement any scheduling algorithm within the SCTP protocol stack in the Linux kernel and is validated via testbed experiments.

In addition to the above-mentioned end-to-end solutions, there is another class of schemes (such as [13], [6], [14]) which implements service differentiation within the network. Those schemes are able to provide performance guarantees that are not possible with end-to-end schemes. However, they fail to gain wide deployment in the Internet due to a number of technical and economic reason. We view these schemes to be complementary to our proposal. If present, they can be integrated with our scheme to provide guaranteed QoS. If not, our proposal can still bring significant performance enhancement as demonstrated in Section III.

V. Conclusion

In this paper, we augment the multi-streaming feature of SCTP with pluggable scheduling. With this extension, SCTP multi-streaming is further exploited to achieve service differentiation. We provide an interface for users to customize the manipulation functions of the DATA chunk output queue of an SCTP association so that different multi-stream scheduling algorithms can be plugged into the protocol stack. We also extend the SCTP socket API so that applications can make use of this new feature by adding only a few lines of code. This extension of SCTP is fully backward compatible and can be safely deployed in the current Internet. We demonstrate its effectiveness and flexibility through real implementations in the Linux kernel and realistic testbed experiments.

References

[1] R. Stewart, "Stream Control Transmission Protocol," IETF RFC 4960, Sep. 2007.

[2] R. Stewart, I. Arias-Rodriguez, K. Poon, A. Caro, and M. Tuexen, "Stream Control Transmission Protocol (SCTP) Specification Errata and Issues," IETF RFC 4460, Apr. 2006.

[3] R. Seggelmann, M. Tuexen, and E. P. Rathgeb, "Stream Scheduling Considerations for SCTP," in *Proceedings of the 18th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, Sep. 2010.

[4] Y. Wang, "[PATCHv2 0/5] sctp: add multistream scheduling feature," https://lkml.org/lkml/2010/9/11/172, Sep. 2010.

[5] R. Stewart, K. Poon, M. Tuexen, V. Yasevich, and P. Lei, "Sockets API Extensions for Stream Control Transmission Protocol (SCTP)," Internet Draft draft-ietf-tsvwg-sctpsocket-24, Oct. 2010, work in progress.

[6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," IETF RFC 2475, Dec. 1998.

[7] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu, "Characterizing Residential Broadband Networks," in *ACM IMC*, Oct. 2007.

[8] H.-Y. Hsieh and R. Sivakumar, "pTCP: An End-to-End Transport Layer Protocol for Striped Connections," in *IEEE ICNP*, Nov. 2002.

[9] J. Crowcroft and P. Oechslin, "Differentiated End-to-End Internet Services Using a Weighted Proportional Fair Sharing TCP," *SIGCOMM Computer Communication Review*, vol. 28, no. 3, pp. 53–69, 1998.

[10] J. Zou, "Preferential Treatment of SCTP Streams in a Differentiated Services Environment," Ph.D. dissertation, City University of New York, 2007.

[11] T. Dreibholz, R. Seggelmann, M. Tuexen, and E. P. Rathgeb, "Transmission Scheduling Optimizations for Concurrent Multipath Transfer," in *PFLDNeT*, Nov. 2010.

[12] G. Heinz, "Priorities in SCTP Multistreaming," Master's thesis, University of Delaware, 2003.

[13] R. Braden, D. Clark, and S. Shenker, "Integrated Services in the Internet Architecture: an Overview," IETF RFC 1633, Jun. 1994.

[14] M. Podlesny and S. Gorinsky, "RD Network Services: Differentiation through Performance Incentives," in *ACM SIGCOMM*, Aug. 2008.