Congestion and Flow Control in the Context of the Message-Oriented Protocol SCTP

Irene Rüngeler[1], Michael Tüxen[1], and Erwin P. Rathgeb[2]

[1] Münster University of Applied Sciences
Department of Electrical Engineering and Computer Science
Bismarckstrasse 11
D-48565 Steinfurt, Germany
{i.ruengeler,tuexen}@fh-muenster.de
[2] University of Duisburg-Essen
Ellernstrasse 29
D-45326 Essen, Germany
erwin.rathgeb@iem.uni-due.de

Abstract. Congestion and flow control are key mechanisms used to regulate the load in modern packet networks. The new IETF Stream Control Transmission Protocol (SCTP) inherited these algorithms from the Transmission Control Protocol (TCP). Although the principles used are the same, some issues arise from the fact that SCTP operates message-oriented whereas TCP operates byte-stream oriented. SCTP also supports bundling of multiple small user messages into one SCTP packet. As a consequence, the overall overhead of an SCTP packet depends on the user message size and the number of user messages that are bundled into the packet. RFC 4960 defining SCTP does not specify whether the message specific headers have to be considered when updating the parameters for congestion control. We will show that neglecting the additional headers when calculating outstanding bytes can lead to unfairness towards TCP connections. We will also show that incorrect handling of the additional memory needed to process each message in the flow control calculations will lead to an exhaustion of the receiver window resulting in a huge amount of unnecessary retransmissions. Based on experiments with the flow control of the SCTP implementations available in several operating systems, we will identify the issues and analyze them by using simulations. As a result, we will present solutions that will lead to fairness towards TCP and reduce the number of retransmissions substantially. Although we will focus on SCTP, the results are also true for other message-oriented protocols using bundling.

Keywords: Congestion Control, Flow Control, Message Orientation, SCTP.

1 Introduction

Although the Transmission Control Protocol (TCP) is still the main transport protocol for IP-based networks, the Stream Control Transmission Protocol (SCTP) [1]

L. Fratta et al. (Eds.): NETWORKING 2009, LNCS 5550, pp. 468–481, 2009.
© IFIP International Federation for Information Processing 2009

gains more and more importance being integrated in the major operating systems. In contrast to TCP, SCTP is not byte-stream oriented but message-oriented. It also supports bundling of multiple small user messages into one SCTP packet to increase transport efficiency.

In order to prevent congestion, SCTP adopted the window based algorithms for flow control and congestion control from TCP. Although the mechanisms seem to be the same for both protocols, the message orientation of SCTP and thus the data processing on a per message basis requires specific consideration.

In TCP, the header size is relatively small and almost constant (between 20 and 60 bytes depending on the TCP options used). Therefore, it is not considered for the flow and congestion control computations. In SCTP, however, every packet requires a common header and an additional header of 16 bytes for each user message. When bundling several small messages into one SCTP packet, this additional overhead is variable and can have a significant impact on the flow and congestion control calculations.

RFC 4960 defining SCTP gives no clear recommendation how to handle the message overhead, and it is up to the implementer of the protocol, if it is included in the calculations or not. As a consequence, implementations like Solaris include the additional headers whereas in FreeBSD they are excluded.

The choice on handling the message overhead in the calculation of the amount of outstanding (not yet acknowledged) data for congestion control can lead to unfairness towards competing TCP connections, if small messages are sent. Therefore, after a short introduction of the relevant SCTP features in Section 2, we will investigate this issue in detail in Section 3 by using simulations. We will show that only including the message headers allows to achieve the mandatory TCP-friendliness.

Inappropriate handling of the overhead, when calculating the amount of data a (slow) receiver is still able to accept, can lead to memory exhaustion at the receiver before the sender is prevented from sending. This leads to packet loss and consequently to unnecessary retransmissions. Based on experiments with the implementations of the Linux 2.6.25 kernel, FreeBSD release 7.0 and Solaris 10, we will highlight the flow control issues in Section 4 and propose a way to implement SCTP flow control such that no unnecessary retransmissions are triggered. This includes a correct announcement of the receiver window as well as applying the silly window syndrome (SWS) avoidance and enabling the Nagle algorithm. The proposal will be validated by simulations.

The simulations in this paper have been performed with the OMNeT++ simulation environment [2] and an extended version of the SCTP simulation model [3] we contributed to the INET framework [4].

2 SCTP, a Message-Oriented Protocol

With the emergence of IP-based networks as universal platform for communication services the need arose to send telephony signaling data across the internet. Signaling data feature relatively small single messages that have to be sent

reliably and some of them also in the correct sequence. To fulfill this task, the new protocol SCTP was designed as a reliable message-oriented protocol and finally adopted by the IETF as official standard in RFC 2960 in 2000. After some modifications, RFC 4960 [1] adopted in September 2007, is the current SCTP specification.

An SCTP packet consists of a 12 byte common header and a number of so called *chunks*. Each chunk has a chunk header that varies with the chunk type. The `DATA`-chunk header is 16 bytes long. Its payload has to be 32 bit aligned. Each chunk is identified by a transmission sequence number (TSN).

As SCTP is a reliable transport protocol, the arrival of user data has to be acknowledged. This is handled by `SACK`-chunks. Here the cumulative TSN ack parameter indicates the highest TSN received in sequence. The acceptance of additional chunks is reflected in gap ack blocks. Another important parameter of the `SACK`-chunk is the advertised receiver window (arwnd), that announces the amount of bytes the receiving endpoint can still accept.

More information about other distinctive features of SCTP like multi-homing or multi-streaming is provided in RFC 3286 [5].

3 SCTP Congestion Control

Congestion control is a mechanism to control the traffic of a network. The goal is to prevent senders from blocking links by reducing the rate of sending packets. Although in the next subsections we will focus on the congestion control mechanism integrated in the examined implementations, the considerations are also true for other congestion control algorithms and other message-orientated protocols using bundling.

3.1 The Congestion Window

The congestion window limits the number of bytes the sender is allowed to transmit before waiting for a new acknowledgement. That means, that not more than cwnd bytes may be outstanding.

The congestion control mechanism is divided into two phases. The first one is called *slow start*. It operates for cwnd values less than or equal to the slow start threshold, which is set to an arbitrary value (mostly the advertised receiver window of the peer during association setup) at the beginning of an association. *Slow start* is characterized by an exponential increase of the congestion window. Every time an incoming `SACK`-chunk announces that the cumulative TSN ack parameter has advanced and the cwnd is fully utilized, i.e. the number of outstanding bytes is greater than cwnd, the minimum of the path MTU and the acknowledged bytes is added to cwnd.

When cwnd exceeds the slow start threshold, *congestion avoidance* makes for a linear increase of cwnd. As the growth of cwnd can lead to an excessive injection of data into the network, packet loss is the consequence. While fast retransmissions result in halving the congestion window, a timer based retransmission leaves cwnd at the size of the path MTU and in *slow start* again. Thus cwnd follows usually a zigzag curve in the lifetime of a connection.

3.2 Counting Outstanding Bytes

As pointed out, cwnd has an influence on the network load and thus on the throughput. Therefore, the way the outstanding bytes, that limit cwnd, are counted, is important and should be examined.

Looking at an SCTP packet containing several data chunks, the amount of user data can vary significantly with the size of the individual chunks (i.e. messages) assuming the same packet length.

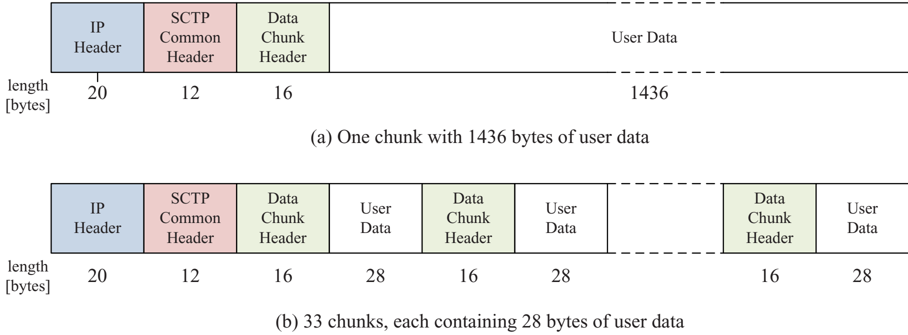

Fig. 1. SCTP packet format

In Figure 1(b) the packet contains 33 `DATA`-chunks with 28 bytes of user data each, adding up to 924 bytes of user data compared to 1436 bytes in the packet in Figure 1(a). Both packets have a size of 1484 bytes. Whereas the overhead is just 1 % in (a) the headers add up to 36 % in (b) and can be more than 60 % for even smaller user message sizes.

Therefore, we have to distinguish between the amount of data that is injected into the network and the user data that arrive at the application layer. Whereas the first has a direct impact on the network load, the second results in the goodput. Both depend on the number of packets (1), that are allowed by the cwnd.

$$NoOfPackets = \left\lceil \frac{cwnd}{Size_P} \right\rceil \tag{1}$$

Calculating the size of a packet ($Size_P$), the headers for IP (H_{IP}) and SCTP (H_{SCTP}) have to be considered as well as the size of the `DATA`-chunks ($Size_{Chunk}$).

$$Size_P = H_{IP} + H_{SCTP} + CPP \cdot Size_{Chunk} \tag{2}$$

The number of the chunks per packet (CPP) is calculated as

$$CPP = \left\lfloor \frac{MTU - H_{IP} - H_{SCTP}}{UMS + P_{UMS} + H_{Chunk}} \right\rfloor \tag{3}$$

The average user message size (UMS) per packet and the corresponding padding bytes (P_{UMS}) feature the variable parts of the packets.

$Size_P$ is a variable used for calculating the bytes allowed by the cwnd, and is not necessarily equivalent to the real size of a packet. If talking about $Size_P^+$, the bundled chunks are calculated with header $Size_{Chunk}^+ = UMS + P_{UMS} + H_{Chunk}$ and for $Size_P^-$ is $Size_{Chunk}^- = UMS$ without header. To compute the number of bytes that are induced into the network and which arrive at the receiver, four different cases are possible:

– Network load taking the header into account

$$Bytes_{SCTP}^+ = \left\lceil \frac{cwnd}{Size_P^+} \right\rceil \cdot Size_P^+ \tag{4}$$

– Bytes at the application layer if the header had been taken into account

$$Bytes_{App}^+ = \left\lceil \frac{cwnd}{Size_P^+} \right\rceil \cdot CPP \cdot Size_{Chunk}^- \tag{5}$$

– Network load without taking the header into account

$$Bytes_{SCTP}^- = \left\lceil \frac{cwnd}{Size_P^-} \right\rceil \cdot Size_P^+ \tag{6}$$

– Bytes at the application layer if the header had not been taken into account

$$Bytes_{App}^- = \left\lceil \frac{cwnd}{Size_P^-} \right\rceil \cdot CPP \cdot Size_{Chunk}^- \tag{7}$$

After the initialization of a connection, the initial window is specified in [6] to be

$$\min(4 * \text{MSS}, \max(2 * \text{MSS}, 4380 \text{ bytes})) \tag{8}$$

Assuming a path MTU of 1500 bytes, the maximum segment size (MSS) equals 1460 bytes, which is 1500 bytes minus 20 bytes for the IP header and 20 bytes for the TCP header. Thus the initial cwnd is 4380 bytes.

As one property of fairness is the evenly distribution of the link bandwidth, we will look at the behavior of associations (terminology for SCTP connections) with and without header inclusion in more detail.

3.3 TCP-Friendliness

When SCTP was designed, one of the major goals was to guarantee TCP-friendliness. In RFC 2309 [7] a *TCP-friendly* or *TCP-compatible* flow is defined as follows: "A TCP-compatible flow is responsive to congestion notification, and in steady state it uses no more bandwidth than a conforming TCP running under comparable conditions."

Since TCP is a byte-stream oriented protocol and typically the Nagle algorithm is not disabled, all packets are filled with enough user data to result in full sized link layer frames, if sufficient data are provided in the send queue. The

overhead consists of the IP-header and the TCP-header, which is independent from the user message sizes.

Although SCTP and TCP implementations, which we inspected for the differences in the handling of header bytes, are readily available, we will base our solutions on simulation results. Since we found that some implementations have bugs substantially influencing the measurement results we decided to use a simulation for the measurements instead of waiting for the bugfixes to be included in the implementations.

3.4 The Simulation Scenario

We have used the INET framework [4] as a simulation tool. Although TCP is integrated in the framework, not all optional TCP features that are common nowadays, like Appropriate Byte Counting (ABC) [8] or delayed acknowledgements [9], are implemented. However, some of these features are mandatory for SCTP and are, therefore, implemented in the INET SCTP model, contributed by us [3]. Hence, a meaningful comparison between SCTP and TCP is not possible with INET. Nevertheless, we wanted to examine TCP-friendliness for flows with and without counting the header. Therefore we used an SCTP association transporting user data messages of 1452 bytes length to mimic the behavior of a state-of-the-art TCP connection. From a congestion control perspective, such an SCTP association behaves identical to a TCP connection. When talking about including or excluding the header, we always refer to the `DATA`-chunk header of 16 bytes.

The scenario for the simulation consists of an SCTPClient, connected to an SCTPServer, and an TCP-like client, connected to a TCP-like server. The connections have to share a bottleneck link between two routers with a data rate of 1 Mbps. The SCTP client sends data with configurable user message sizes from 12 to 204 bytes to the SCTP server. The TCP-like client only sends full packets with a payload of 1452 bytes, the headers are not included. Including them does not change the result, since the difference is neglectable for large user messages.

To test the behavior with and without counting the header bytes, the SCTP simulation has been extended by two parameters, *osbWithHeader* and *padding*. They are boolean variables that can be set to true, if the header and the padding bytes should be taken into account for the congestion control calculations.Tests showed that the influence of the padding bytes is not significant. Therefore, all described simulations were run with either both variables true or false.

3.5 Fairness on the Transport Layer

The SCTP association and the "TCP-like" association have to share the bandwidth equally. This means that all bytes that have been sent over the network have to be counted, including the retransmitted bytes. To assure that the same time interval is chosen and the associations have reached a steady state, a start and stop time can be configured for counting the bytes that have arrived at the server. The timers were set for the measurement to start after 50 secs and continue for 400 secs. As the ratio of additional header bytes to the user message

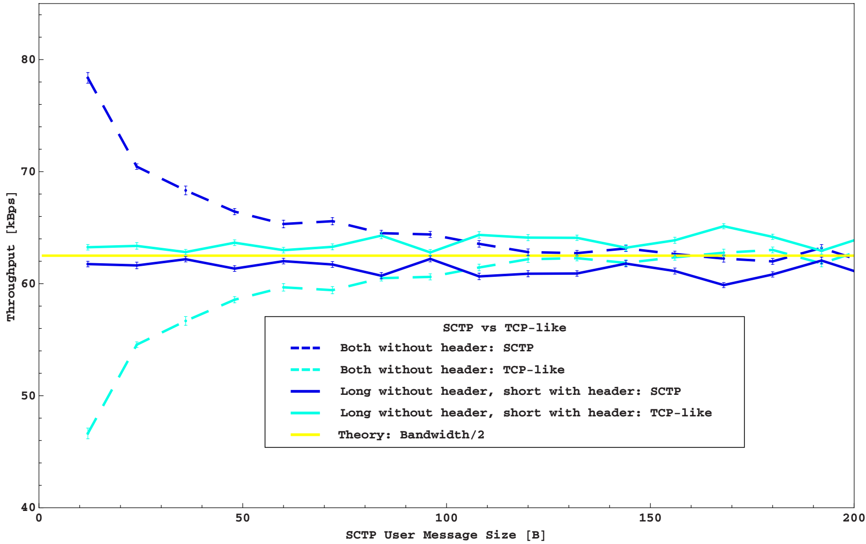

Fig. 2. Throughput on the transport layer

size is only significant for small payload sizes, we chose user messages from 10 to 200 bytes length in 10 byte intervals. Each simulation run was repeated 100 times with different seeds for the random numbers to ensure validity. Figure 2 shows the throughput on the transport layer. The vertical bars represent the 95 % confidence intervals. It is obvious that the associations that calculate the outstanding bytes with header share the link, symbolized by the straight line, equally, whereas the behavior is not fair for small message sizes, if the header is not taken into account. The SCTP client utilizes the link much more intensively than the TCP-like client, thus taking bandwidth from the other connection.

3.6 Fairness on the Application Layer

The behavior on the transport layer has an influence on the throughput on the application layer (goodput). Therefore, we chose the same setup as in the last section and counted the bytes that arrived at the user level of the servers during a predefined time period and calculated the throughput. Figure 3 shows the graphs when the header is not taken into account. Although the TCP-like client achieves a higher throughput than the SCTP client using the different message sizes, the throughput is much lower than it should be. As Figure 2 indicated, the SCTP client takes over so much bandwidth that the TCP goodput is considerably reduced. The two theoretical graphs show the ideal case, where the SCTP client (lowest graph) and the TCP-like client (top graph) share the link equally. The zigzagging of the lowest graph results from padding, i.e. the necessity to add 1 to 3 bytes to the payload to get it 32 bit aligned. The graphs

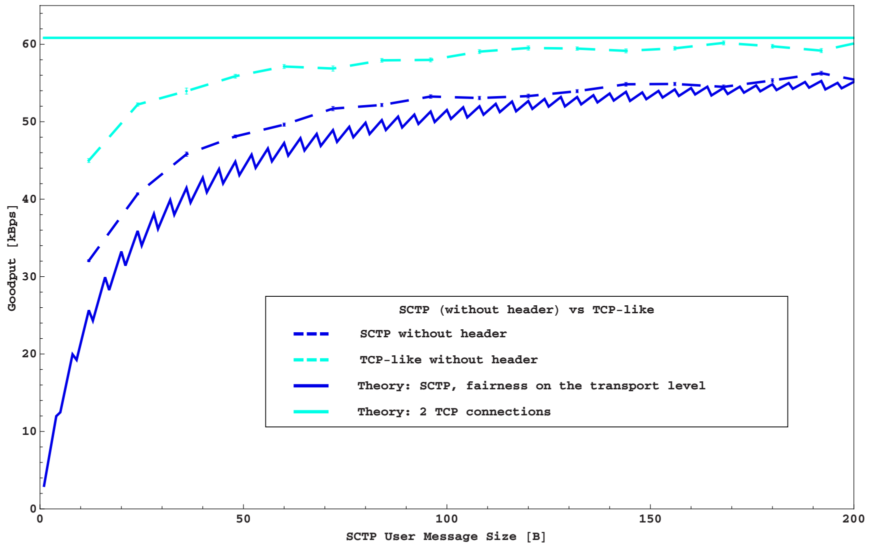

Fig. 3. Goodput, without considering the header

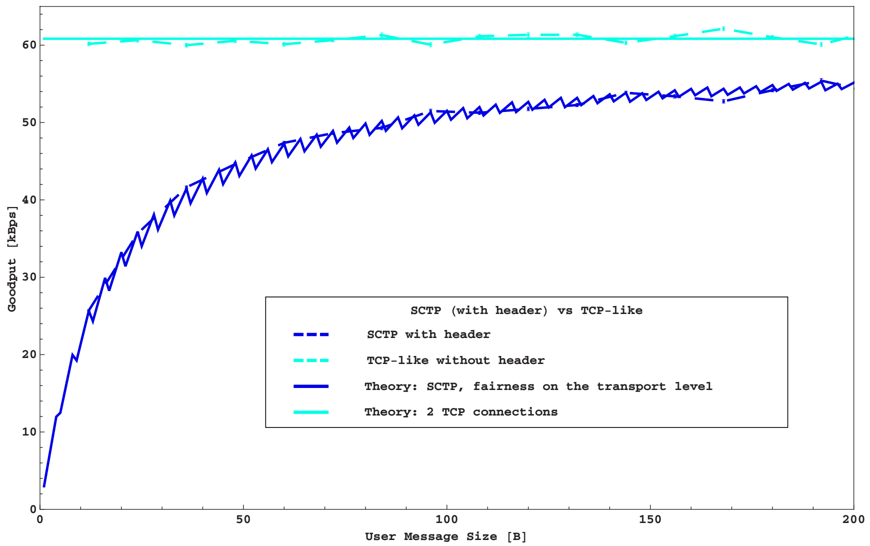

Fig. 4. Goodput, if the header is taken into account

in Figure 4 illustrate the results, if the header is taken into account. Now the curves show the desired behavior and fit the theoretical graphs.

As a result it can be postulated, that all implementations of message-oriented protocols with bundling should take the headers into account, when calculating the outstanding bytes, in order to be TCP-compliant.

4 Flow Control

4.1 The Concept of SCTP Flow Control

Flow control like congestion control is a mechanism to influence the amount of data injected into the network. Whereas the congestion control protects the network from a fast sender, the flow control should prevent the overload of the receiver.

To achieve this, the advertised receiver window parameter (arwnd) is used to announce the amount of data that the receiver is willing to accept. During the setup of the association the hosts exchange their initial arwnd in the `INIT` and `INIT_ACK`-chunk. Upon arrival of a `DATA`-chunk, arwnd is decremented by the message size. When the data is delivered to the upper layer, arwnd can be incremented again. When the receiver sends a `SACK`-chunk to acknowledge data, it includes the actual value of the arwnd. The sender tries to keep track of the size of its peer's arwnd by trying to predict the window size. It takes the value of the announced arwnd as basis, reduces it by the number of outstanding bytes, i.e. the data that are assumed to be in flight. If the peer's arwnd reaches zero, only one `DATA`-chunk may be sent to probe the window, which is similar to the Zero Window Probing mechanism in TCP (see [10]). But before zero is reached the silly window syndrome (SWS) avoidance algorithm (see [11]) has to be applied. FreeBSD achieves this by announcing a window of 1, if its size has dropped below two MTUs. During that time data is still accepted, but the sender is warned to reduce the amount of data.

4.2 SCTP Flow Control in Implementations

As flow control is a major feature of SCTP, it is supported in all available implementations. The advertised receiver window corresponds to the important resource receiver window, but the sizes are not necessarily the same. Upon arrival of a packet, the kernel has to provide memory for the storage of each chunk. Besides the actual user message, information has to be stored, like the stream sequence number, the TSN and so on. The amount of memory needed depends on the operating system.

We examined the change of the advertised receiver window for the Linux Kernel 2.6.25, OpenSolaris 10 and FreeBSD version 7.0. We distinguished two scenarios to see whether the implementations behaved in a different way, when gaps were reported or not. First, the application at the receiver was prevented from reading. Second, the first Transmission Sequence Number (TSN) was left out. Thus a gap was created preventing SCTP from pushing data to the upper layer.

The experiment was performed by using an SCTP testtool [12], to generate SCTP packets. Test scripts were programmed with the Guile scheme implementation [13] to create the desired message flow on the sending side.

FreeBSD behaved differently in the two scenarios. In the first one, the arwnd was reduced by the payload size plus an overhead of 256 bytes, which is equal to the memory that the kernel allocates for a chunk. In the second case the arwnd was only decremented by the payload size. For small message sizes a limit of the maximum number of chunks that were accepted was observed. When this limit of 3200 chunks was reached, the arwnd was not reduced any more, and newly arrived packets were dropped. This limit is a means for the kernel to protect resources. It can be configured by the network administrator, if necessary. Hence, the number of chunks accepted can be computed by

$$n = \max(\mathit{3200}, \left\lceil \frac{arwnd}{\mathit{256} + UMS} \right\rceil) \tag{9}$$

Linux showed the same behavior in both scenarios. The arwnd is always reduced by the user message size (UMS). For messages smaller than 176 bytes, only

$$n = \left\lceil \frac{arwnd * \mathit{2}}{\mathit{176} + UMS} \right\rceil \tag{10}$$

chunks are accepted. Then the arwnd is not reduced any more.

OpenSolaris decrements the arwnd by the UMS until the next message does not fit any more. Thus the window is reduced to a value smaller than UMS and the number of accepted messages equals

$$n = \left\lfloor \frac{arwnd}{UMS} \right\rfloor \tag{11}$$

Neither in the Linux nor in the OpenSolaris SCTP kernel the silly window syndrome avoidance principle is implemented.

4.3 Simulation Results

Keeping in mind that all implementations need extra memory to store the received user data, and that the arwnd is coupled with the receiver window, we wanted to examine the effects of the different implementation dependent algorithms and their impact on interoperability. Therefore, we extended the simulation by a parameter for the additional memory needed per chunk. As seen in Linux and partly in FreeBSD, the receiver reduces its receiver window by the UMS plus the additional memory, but announces an arwnd, that is only decremented by the UMS. The sender, not knowing that the arwnd does not report the true value, tries to keep track of its peer's window and adjusts the value every time a `SACK`-chunk arrives.

To simulate this behavior and examine its impact on the network load, we configured a slow receiver by distributing the reading intervals exponentially with a mean of $\frac{5000\text{bytes}}{UMS}$ secs. After each interval, one message was read, so that

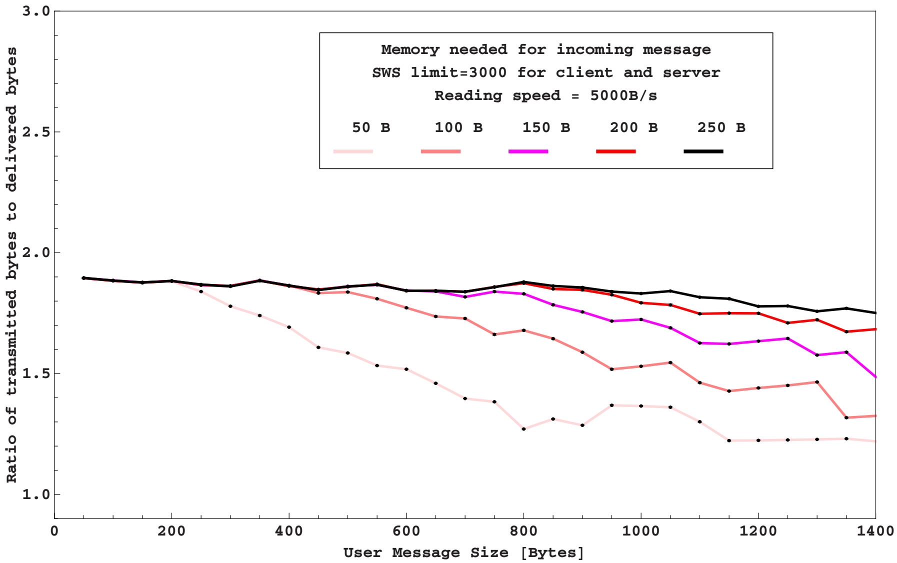

Fig. 5. Ratio of transmitted to delivered bytes for a varying amount of additional memory

approximately 5000 bytes were read per second independent from the UMS. Figure 5 shows the results for 50 to 250 bytes overhead. Here and in the next simulations, each run was repeated 10 times. The black dots represent the 95% confidence intervals. As a rate of the network load we calculated the ratio of the transmitted bytes, meaning the data sent over the network including all retransmissions, to the data that reached the upper layer. For an overhead of 250 bytes, which is even less than the memory needed by FreeBSD, almost twice the necessary data is transmitted. The other graphs show that the number of retransmissions is less for larger message sizes. Nevertheless, the ideal ratio of 1 is never reached. Noteworthy is also the slight inclination of the lowest graph for larger message sizes. This can be explained as follows. When an arwnd of 0 is announced, the sender is allowed to send zero window probes in the absence of outstanding data. Zero window probes consist of one `DATA`-chunk. The method that was chosen to simulate a slow receiver implies that the reading intervals are much smaller for small message sizes than for bigger ones. Thus the probability that data has been pushed and a new chunk can be accepted is higher for smaller messages. As a consequence the larger messages are more likely to be dropped.

Another difference between the operating systems is the implementation of the silly window syndrome avoidance algorithm. Of the three operating systems only FreeBSD has integrated this feature. In the following, we will examine the impact of its availability.

We varied the presence of SWS avoidance for sender and receiver and plotted again the ratio of transmitted to delivered bytes. We chose an additional memory

of 50 bytes, because the measurements with the more realistic memory size of 250 bytes revealed a worse ratio, that made a graphical judgment almost impossible. Figure 6 shows the results for this scenario. As the measurements of Figure 5 were taken with SWS enabled for sender and receiver, the lowest graph of Figure 6 is equal to the 50 bytes graph of Figure 5. It is well to be seen, that the absence of the SWS avoidance algorithm on the sending side has a negative impact on the network load, whereas the implementation on the receiving side is not as important. The network load can be reduced by up to 20% , if the sender follows the principals of SWS avoidance.

For smaller chunks bundling results in a better payload to header ratio. The Nagle algorithm (see [14]) is a feature, first introduced in TCP, to prevent the sending of small packets, if there are still data in flight. For SCTP this means, that chunks have to be bundled, until the next chunk does not fit in the packet any more, unless there are no bytes outstanding. Applying this algorithm leads to delaying the sending of data. To examine the impact of the Nagle algorithm on the network load, we carried out the same runs as in Figure 6 and disabled the execution of the Nagle algorithm. The results showed that the number of retransmissions stayed at 80% to 90%, if the SWS avoidance algorithm was not applied on the client.

4.4 Solutions

Our first idea was to notify the sender about the amount of additional memory needed. Thus the sender was to be able to predict the reduction of the arwnd

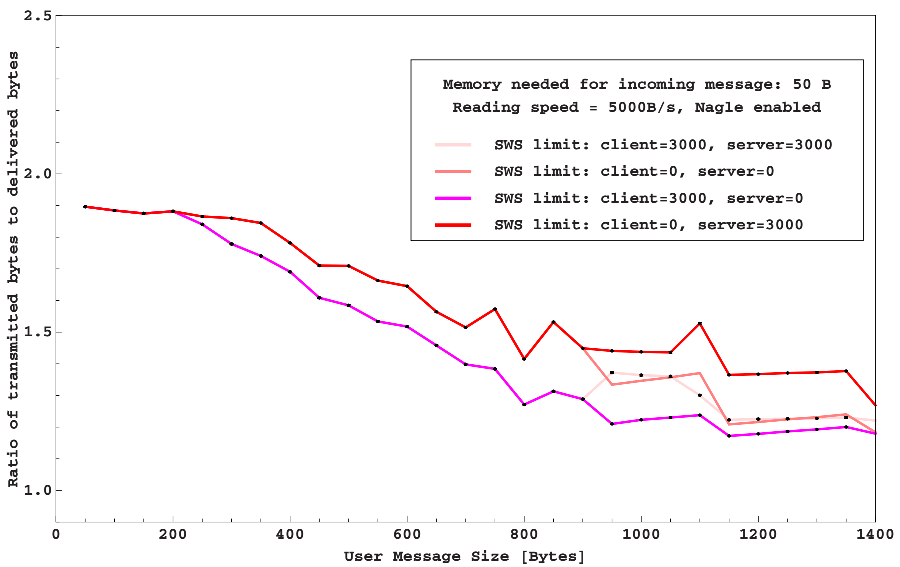

Fig. 6. Ratio of transmitted to delivered bytes in the absence or presence of the SWS avoidance algorithm

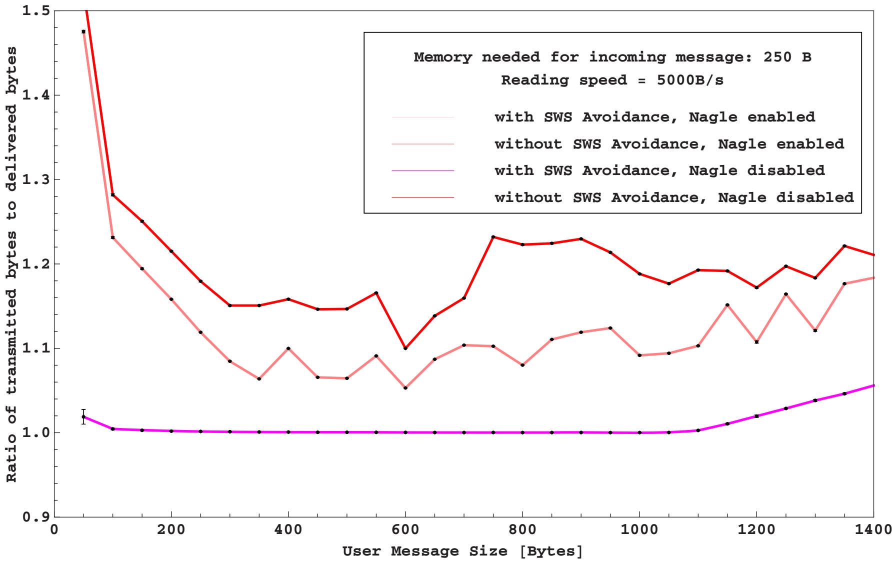

Fig. 7. Ratio of transmitted to delivered bytes, if the size of the real receiver window is announced

more exactly. However, simulation runs with this feature did not lead to significantly better results.

The best outcomes were achieved by "telling the truth". Like in FreeBSD, when the receiver did not read, the arwnd was reduced by the payload and the additional memory. Even if the sender cannot follow the peer window closely, the regular updates are enough to guide the sender.

Figure 7 shows the transmitted to delivered bytes ratio for the cases that SWS avoidance is present and Nagle enabled. It is well to be seen, that the application of the Nagle algorithm only has an impact on the number of retransmissions, if SWS avoidance is not present. In the other case, both graphs are the same. Thus, if SWS avoidance is applied, there are almost no retransmissions needed. The reason for the increase of the graph for larger user message sizes has been explained in the last subsection.

As a consequence, the strategy for implementors to avoid retransmissions in case of flow control is to set the advertised receiver window to the real value, so that it reflects the receiver window. Thus it is not possible that the receiver window runs out of memory before the arwnd reaches zero.

5 Conclusion

In this paper we discussed the influence of the message orientation on the implementation of congestion and flow control, using the example of SCTP.

We pointed out that the way the outstanding bytes are counted, has an impact on the fairness towards TCP connections. Therefore we recommended that the outstanding bytes should be calculated by taking the data message headers into account.

Looking at the different implementations and their way to apply flow control, we encountered problems, if the receiver window was exhausted faster than the advertised receiver window. The absence of the SWS avoidance algorithm still worsened the transmitted to delivered bytes ratio. Simulation results revealed that a solution to this problem is the announcement of the value of the actual receiver window in the advertised receiver window parameter. The application of the SWS avoidance algorithm even led to runs without retransmissions.

References

1. Stewart, R.: Stream control transmission protocol. RFC 4960 (September 2007)
2. Varga, A., Hornig, R.: An Overview of the OMNeT++ Simulation Environment. In: International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SimuTools) (March 2008)
3. Rüngeler, I., Tüxen, M., Rathgeb, E.: Integration of SCTP in the OMNeT++ simulation environment. In: Proc. of the 1st international conference on Simulation tools and techniques for communications, networks and systems workshops (2008)
4. Varga, A., Hornig, R.: INET Framework Documentation (2009), http://github.com/inet-framework/inet
5. Ong, L., Yoakum, J.: An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286 (May 2002)
6. Allman, M., Floyd, S., Partridge, C.: Increasing tcp's initial window. RFC 3390 (October 2002)
7. Braden, B., et al.: Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309 (April 1998)
8. Allman, M.: TCP Congestion Control with Appropriate Byte Counting (ABC). RFC 3465 (February 2003)
9. Allman, M., Paxson, V., Stevens, W.: TCP Congestion Control. RFC 2581 (April 1999)
10. Postel, J.: Transmission Control Protocol. RFC 793 (September 1981)
11. Clark, D.: Window and Acknowledgement Strategy. RFC 813 (July 1982)
12. Tüxen, M.: SCTP Testtool, http://sctp.fh-muenster.de/sctp-testtool.html
13. Free Software Foundation: Guile, http://www.gnu.org/software/guile/guile.html
14. Nagle, J.: Congestion Control in IP/TCP Internetworks. RFC 896 (January 1984)