ABSTRACT

DESIGN AND IMPLEMENTATION OF SPLIT TCP IN THE LINUX KERNEL

by
Rahul Jain

The Transmission Control Protocol (TCP) was designed for reliable communication between computers over networks of unpredictable quality. It has admirably succeeded in satisfying the needs of the growing internet. Yet, there are combinations of network problems too bad even for TCP. In particular, in the situation of simultaneously very high delay (e.g. a satellite link) and high loss or even fading (a low quality earthlink or wireless link) on the same connection TCP can break down.

A known solution is "Split TCP" where one or more proxies (called Helper Boxes) are introduced to break the end-to-end connection into few (almost) independent legs. Each of the legs has its own feedback, error control, congestion control etc. mechanism. Preferably, connections are split into legs having high RTT or high loss, but not both.

The main contribution of this dissertation is the design and implementation of "Split TCP" using the Netfilter System in the Linux kernel, and the use of IP over IP for transport. The dissertation also gives a mathematical guarantee for improved TCP performance with Split TCP. By analyzing the mathematical result, this dissertation concludes that localizing network problems one per leg will guarantee the maximum improvement possible with Split TCP. Through experiments conducted over an actual network, this deduction is proven to hold true.

The kernel implementation reduces overhead. The implementation used leaves TCP packets and flags intact, thus allowing use of SSH (etc) over a Split TCP connection. The implementation lets the helper box negotiate, for "inter-HB legs", performance enhancing options like window scaling and Explicit Congestion Notification

(ECN) support irrespective of the end-host capabilities. This allows a pair of helper boxes to have improved performance, thus increasing the throughput of the overall connection. Depending on the configuration of an end host, these options will also be negotiated between the end host and the HB. The use of IP over IP allows use of several helper boxes in a connection and makes it easier to achieve transparency for the original end-hosts.

The results of the experiments have been very promising. For example, with various drop probabilities, a connection with 1 helper box was, on an average, 9.5 times faster in comparison than one without. For a similar experiment with 3 HB's a Split TCP connection is on an average 8.29 times faster than a regular TCP, with the factor of improvement increasing with increasing drop probability. These results met the theoretical expectations of large improvements in situations with higher and asymmetric drop probabilities. The implementation was also tested in a heterogeneous environment where high loss and high delay are inherent in the wireless leg of the connection. The results have also shown the solution to be scalable.

The primary area of use is for internet connections, irrespective of the user application and the medium of connection, wired or wireless. This is unlike other proxies which are either application dependent or do not support certain applications (e.g.: interactive).

DESIGN AND IMPLEMENTATION OF SPLIT TCP IN THE LINUX KERNEL

by
Rahul Jain

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

August 2007

APPROVAL PAGE

DESIGN AND IMPLEMENTATION OF SPLIT TCP IN THE LINUX KERNEL

Rahul Jain

Dr. Teunis J. Ott, Dissertation Advisor — Date
Visiting Professor, Winlab, Rutgers University.
Professor of Computer Science, New Jersey Institute of Technology (to 2006)

Dr. Cristian Borcea, Committee Member — Date
Assistant Professor of Computer Science, New Jersey Institute of Technology

---

Dr. Andrew Sohn, Committee Member — Date
Associate Professor of Computer Science, New Jersey Institute of Technology

---

Dr. Nirwan Ansari, Committee Member — Date
Professor of Electrical and Computer Engineering, New Jersey Institute of Technology

---

Dr. T. V. Lakshman, Committee Member — Date
Director, Networking Research Laboratory, Bell Laboratories

BIOGRAPHICAL SKETCH

Author: Rahul Jain

Degree: Doctor of Philosophy

Date: August 2007

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2007
- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2002
- Bachelor of Computer Engineering,
  Mumbai University, Mumbai, India, 1999

Major: Computer Science

Presentations and Publications:

R. Jain, and T. Ott, "Design and Implementation of Split TCP in the Linux Kernel," *in Proceesings of Globecom 2006*, Paper NXG-03-6.

R. Jain, "Design and Implementation of Split TCP in the Linux Kernel," *Workshop of IFIP, WG 7.3*, Saint-Malo, France, July 2006.

R. Jain, "Design and Implementation of Split TCP in the Linux Kernel," *WINLAB*, Rutgers University, March 2007.

This dissertation is dedicated to my parents and sister who have always pushed me to strive harder and had faith in me even when I was beginning to doubt myself. Your love, care, support and patience has helped me reach where I am today and shaped me into the person I am.

The woods are lovely, dark and deep,
But I have promises to keep,
And miles to go before I sleep,
And miles to go before I sleep.

—From *Stopping by Woods on a Snowy Evening* by Robert Frost

If we knew what it was we were doing, it would not be called research, would it?

—Albert Einstein

ACKNOWLEDGMENT

I would like to express my deepest gratitude to all the people who have supported, on various levels, and encouraged me through the years of my Ph.D. program. This dissertation would have not been possible without it.

First of all, I would like to thank my advisor, Dr. Teunis J. Ott, for giving me the opportunity to work with him. Over these past few years, I have learned a lot from him, both at a professional level and a personal level. Dr. Ott always provided me with an intellectually simulating environment that encouraged me to work and think independently. His extensive knowledge, both theoretical and practical, helped me learn a lot about my research area. His constant guidance, encouragement and strong feedback has greatly contributed towards the success of my work. Working with him has taught me to have an eye for detail, to leave no stone unturned and to work harder each day. I hope and look forward to collaborate with him in future projects.

Next I would like to thank Dr. Cristian Borcea, Dr. Andrew Sohn, Dr. Nirwan Ansari and Dr. T. V. Lakshman for serving as members of my dissertation committee and spending valuable time of theirs during my proposal and dissertation defense. Their insightful suggestions and constructive criticism helped in improving the work greatly.

I am thankful to Dr. Borcea for taking the time to sit through the numerous dry runs of my presentation. His implicit suggestions helped improve the quality of my presentation.

I would also like to thank Dr. Yehoshua Perl for encouraging me to do my Ph.D. and help me start it.

I also want to thank my family and friends who have been there for me at each step of my Ph.D. program. I am forever indebted to my parents for their unconditional love and understanding during the course of my studies. I would also like to thank my friends Santosh, Veena, Suresh, Vishwanath and Punit for their help, both morally and intellectually, over the past few years.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter Page

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION

Over the past few decades, we have witnessed the monumental success of the Internet. What once started as a laboratory experiment, has become an integral part of almost all aspects of ones daily life. The network environment has extended from a single Local Area Network (LAN) to multiple Wide Area Network (WAN)s, with many routers in between the end hosts. And with the introduction of the IEEE 802.11 standard the Internet took a big leap forward, since the network environment can now be a mix of wired and wireless connections or wireless connections altogether. Irrespective of the distance and medium of communication, the Transmission Control Protocol (TCP) still remains the widely accepted means of communication between computers. Under favorable conditions of small Round Trip Time (RTT), negligible probability of loss etc., like those offered in a LAN, the TCP performance is known to be good. However, if the computers are connected through network environments having unfavorable conditions like large RTT and high loss probability, the performance of a TCP connection can be poor. This dissertation proposes use of the Split TCP mechanism to improve TCP performance under such network conditions wherein, the end-to-end connection is broken down into multiple, almost independent, TCP connections.

1.1 Problem Statement

Under favorable conditions of low RTT and negligible probability of packet loss, TCP has proven good performance. However, large RTT and high probability of packet loss are known to affect TCP performance. The "Square Root Law" [1, 2] gives a theoretical argument why. Section 2.2 provides a mathematical proof for the same. To get a practical perspective of how the RTT and probability of packet loss affect the

TCP performance, one can look at the results of the PingER project [3] of the Internet End-to-end Performance Monitoring (IEPM) group. For example, in February 2006, between a host in California and India, the minimum packet loss was 2.242% with an average RTT of 316.204 ms and a maximum TCP throughput of 246.715 kbps. For the same set of computers, in February 2007, the values changed to 1.081%, 403.052 ms and 278.707 kbps respectively. Similarly, in February 2006, between a host in California and a host in New York (New York University), the minimum packet loss was 0% with an average RTT of 77.752 ms and a maximum TCP throughput of 25,391.915 kbps. For the same set of hosts, in February 2007, the values were 0.007%, 78.042 ms and 18,216.284 kbps respectively.

The example of a host in California and a host in India was picked up to reflect the current network conditions of an environment that is of primary interest for this dissertation. Such a network environment will have a high quality, high RTT leg, say one containing a satellite link between USA and a third world country, followed by a low quality, low RTT, high drop (and possibly even fading) leg in the third world country.

Measurements and theory indicate that the larger the RTT and probability of loss, the lower is the TCP throughput. There is a well known explanation for the problem. The congestion control mechanism (congestion avoidance plus fast recovery and retransmit) of TCP is used to prevent packet loss and/or recover from one. The TCP engine recognizes a packet loss with the receipt of 3 duplicate acknowledgments or by timing out while waiting for an acknowledgment. In general, the minimum of the advertised window and the congestion window governs the TCP throughput. During the congestion avoidance phase, the congestion window grows by 1 Maximum Segment Size (MSS) per RTT (or 1 MSS per 2 RTT's, if delayed acknowledgments are used). Because of the large RTT (for example, a satellite link), the congestion window grows slowly. And because of the high probability of loss (and of fading) in the "final 100 or so miles", a TCP connection will spend a significant amount of time in recovery (fast or not) and even in time-outs, with the large RTT making the time-out periods even longer. This results in poor performance that TCP experiences. Long distance networks connected on a 622 Mbps dedicated transoceanic link have also experienced the same [4].

1.2 Contributions

In this dissertation "Split TCP" is studied, enhanced and implemented as a solution for the problem stated in the previous section. This dissertation describes the design and implementation of "Split TCP" in the network stack of the Linux kernel. The results from [3] were used to choose the network parameters (RTT and drop probability) in the investigations of the actual TCP performance.

Split TCP has been extensively researched [5, 6, 7, 8, 9] and shown to improve the performance of TCP in Mobile Networks. It works by breaking the end-to-end connection into two or more "legs" as shown in Figure 1.1. In this dissertation, the term "legs" is used to describe a path that may go through several routers. It could be the path either between an end host and a helper box or the path between two helper boxes.

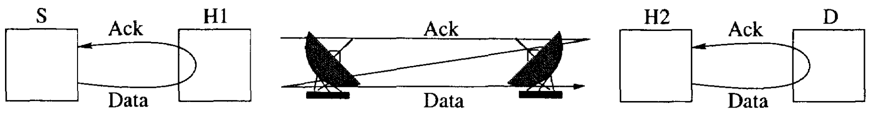

Figure 1.1 A Split TCP connection.

By breaking the connection into legs, Split TCP isolates the network problems of one leg from another. This results in legs with shorter RTT and fewer network problems when compared with the end-to-end connection. Hence, the end hosts experience improved TCP performance. This is proved in Chapter 2. The novelty of work in this dissertation is the design and implementation of Split TCP in the network stack of the Linux kernel. The implementation of Split TCP is done at the kernel level. When compared with a Link Layer implementation, the approach has an advantage of a much larger buffer at the intermediate nodes (which are called Helper Boxes). When compared with the User Level implementation, the approach has the advantage of being able to work directly with packets. This gives direct access to all the fields of the packet headers including the TCP flags, which are kept unchanged. The approach also saves time on no repacketization and no to and fro copying of data bytes to the hard disk.

A Helper Box (HB) acts as a proxy for the source while communicating with the destination host, and vice-versa. In this dissertation, the end host from which the HB received the first SYN packet is denoted as the source host for that TCP flow. In the approach described, each leg is made self sufficient and almost independent of the others. For this, the HB maintains, parameters for flow control, RTT estimation, error control and congestion control, for each leg that it is connected to. A property of the design is that Split TCP is completely transparent with respect to end hosts. Hence, no modifications are required in the network stack or in the user applications of the end hosts. Another contribution of the design is a guarantee that each data packet will pass through the same sequence of HBs. This is guaranteed by using IP over IP for sending data packets between the HBs. This is important for the correct bookkeeping at the HB which ensures the proper flow of data packets between the end hosts.

The implementation of Split TCP chosen in this dissertation allows use of any number of HBs for an end-to-end TCP flow. It thus can be used to guarantee that any leg has at most one network problem to deal with. Measures specialized for the network problem at hand can then be activated in the two HBs of that leg, without the need to modify the end hosts. So different pairs of HBs on a leg can have different special measures active.

Among such special measures are window scaling [10], SACK [11], and those already known in satellite communication [12,13]. Slightly more ambitious would be the use of ECN [14]. The use of ECN would require cooperation between operators of HBs and operators of key routers. Once such cooperation exists, it can be used for joint optimization of congestion dependent probabilistic marking in routers and window size modifications in HBs in reaction to marking in acknowledgments.

Another possibility is to give HBs the option of setting the ECE bit in an acknowledgment being sent, even if no recently arriving data packet had the CE bit set. It might be possible to this way achieve more than can be achieved by manipulating the advertised window.

All TCP/IP features of the HB design were tested successfully with the exception of the ECN option. For the ECN mechanism to work successfully, the following two operations need to work: Marking of the appropriate packet at the router (IP layer or lower) and the processing of such marked packet at the end hosts (TCP layer). Such processing involves reducing the congestion window (if required) and setting of the relevant ECN flags in the TCP header of the outgoing packet. While testing the ECN mechanism, unexpected behavior was observed when the Linux boxes were used as a router. For a given queue configuration and respective heavy traffic load, the router almost never marked a packet. It is our conclusion that the implementation of ECN marking in the Linux kernel (version 2.6.10) is defective. This is discussed further in Chapter 6.

1.3 Additional Situations of Interest

Although the solution developed in this dissertation is primarily for the environment discussed in Section 1.1 and 2.3, it can be applied and used in other situations as well.

One such environment is the heterogeneous network. With the wireless technology becoming an acceptable and preferred means of communication, heterogeneous networks are becoming a part of life. However the wireless link of these networks pose some problems. Although the available bandwidth of the wireless link has increased from 2Mbps to 56Mbps over the years, the inherent problem of a random Bit-Error Rate (BER) can prevent TCP from utilizing the whole bandwidth. This is a direct result of TCP being unable to distinguish between a loss due to congestion and a random packet loss in the wireless link. Hence the TCP congestion window algorithms reduce the Congestion Window (cwnd), decreasing the net throughput. The situation gets worse for links with high BER and frequent disconnects or fading [15].

The solution proposed in this research work can be used to separate the problems of the wireless link from the problems of the wired leg. The design was tested in a network having an actual wireless link and the corresponding results are presented in Chapter 6.

Another environment that is of potential interest is interplanetary communication. An interplanetary communication link has many interesting network problems to deal with, the most intuitive of which is the high RTT. For example, the speed-of-light delay between Earth and Mars ranges from 4 minutes to 20 minutes. Other problems include maintaining continuous connectivity among the end hosts which is questionable because of the orbital nature of the planetary bodies. Also, just like the wireless link, the communication link of these networks suffer from a low Signal to Noise Ratio (SNR), thus triggering the TCP congestion window algorithm unnecessarily. Depending on the network architecture, there might be asymmetric data rates among the links [16].

Split TCP is a good fit as a solution for such networks. By employing several HB's, as explained earlier, each of the above mentioned network problems can be isolated. Since this network environment has a very large RTT, having very large buffers in the HB will greatly improve the TCP performance.

1.4 Organization of Dissertation

The rest of the dissertation is organized as follows. Chapter 2 starts off by explaining the theory behind Split TCP, the topic of this dissertation. This is followed by a mathematical proof that shows why Split TCP will improve the TCP performance for the network situation described above. Finally, an example network scenario which is used for discussion and experiments is described in detail.

Chapter 3 presents related work along with the pros and cons of each method.

Chapter 4 describes the network stack of the Linux kernel in great detail. This study was done to understand how the kernel handles a packet. This knowledge was then used to design and implement the Split TCP mechanism in the Linux kernel. The chapter first discusses the important structures of the network stack. The processing done by the kernel at the link layer, Internet Protocol (IP) layer and TCP layer for an incoming packet is described next. This is followed by a discussion of the processing done by the kernel for an outgoing packet. Finally, the Netfilter system, which is at the core of the Split TCP design, is explained.

Chapter 5 discusses the design of Split TCP developed in this dissertation. Various design options and questions that needed to be answered are discussed in this chapter.

This is followed by the implementation level details in Chapter 6. This includes description of the important structures and functions, the revised TCP state diagram etc. It also discusses the algorithm used to process the different types of TCP packets. The various TCP/IP features implemented in the HB are also presented.

In Chapter 7 the result and analysis of the various experiments are presented. The implementation described in Chapter 5 is tested against wired LAN and heterogeneous networks. Experiments showing the overhead of the implementation with respect to CPU time and memory are also presented. The experiments also show that the design described is scalable.

Chapter 8 provides the conclusions of this dissertation research. It also lists some possible enhancements and future avenues of research with respect to Split TCP.

CHAPTER 2

SPLIT TCP

Split TCP is a well known technique that has been around since the early 1990's. It is also known as "TCP with Boosters", "TCP with proxies" etc. Some research has been done on evaluating the benefits of this technique in Mobile Networks, including cellular data networks. See [5,6,7,17,8,9]. Also see references in [18]. [19] shows that a cellular service provider has implemented a form of Split TCP within its network. However, the details of their design and implementation are unknown.

2.1 Theory Behind Split TCP

A classical TCP connection has always been an end-to-end connection. It consists of two end hosts, which even though TCP allows bidirectional traffic, for convenience of presentation are called, Source (S) and Destination (D). For the purpose of this dissertation, the end host from which the first SYN packet originates is considered to be the source host. TCP mechanisms of flow control, error control, congestion control etc and other TCP processing for a classical TCP flow are as always done by the end hosts. Also, the routers connecting the end hosts do not process the packets beyond the Internet Protocol (IP) layer.

While using Split TCP, this classical TCP connection is broken down into a sequence of TCP connections. Specialized routers, called HB in this research, are introduced within the path of the flow for this. Depending on the network architecture and the requirements, there may be one or more HB's in the path. Figure 2.1 shows a Split TCP connection with one HB within the path.

As shown in Figure 2.1, the end-to-end TCP connection is broken down into two legs; one from S to HB and the other from HB to D. HB intercepts and acknowledges any incoming data packet from S pretending to be D (Acknowledgment Spoofing).

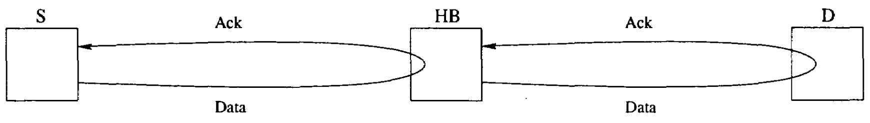

Figure 2.1 Split TCP connection with one HB.

It then becomes responsible for making sure the data packet is delivered to D. Theoretically speaking, acknowledgment spoofing can either be done for each data packet or for every 2 data packets, if delayed acknowledgments are desired. The current implementation of Split TCP does not support delayed acknowledgments. The HB maintains a buffer used to cache the data packets. It also maintains, for each of the legs, various TCP parameters that are used for flow control, error control, congestion control, and RTT estimation. These parameters along with the buffer help the HB in forwarding and in retransmission of the data packets to their respective destination hosts. HB behaves in a similar manner for data packets in the opposite direction. In other words, the HB acts as a proxy for the source host while talking with the destination host and as a proxy for the destination host while talking with the source host.

By splitting the connection, Split TCP is able to isolate the network problems of the legs. Each leg will now have a lower RTT and fewer or no network problems to deal with. It is evident that the throughput of the end-to-end TCP connection will be less when compared to the potential TCP throughput of each of the separate legs. This isolation and localization of network problems one per leg also results in the overall increase in performance of the split connection when compared to the end-to-end TCP connection. A mathematical proof is given in Section 2.2 to support this theory. For this reason, the HB is often termed as a "Performance Enhancing Proxy" [18].

Depending on the working environment and the requirements, there might be a need for more than one HB along the path between the two end hosts as shown in Figure 2.2. Most commonly HB's are introduced to handle the mismatch in TCP capabilities of the end hosts or to make use of a customized protocol within a leg [18]. For example, in Figure 2.2, a new protocol or a modified TCP or a specific TCP option (like window scaling) could be used between H1 and H2 to increase the performance of that leg. This allows for the development of protocols and techniques that will increase TCP performance of legs with either high RTT or high loss. These, when introduced in the HB, can be selected dynamically depending on the network problems of the leg. One such example is the congestion control mechanism proposed in [20]. For these options, once again, no modifications are required in the networking code of the end hosts.

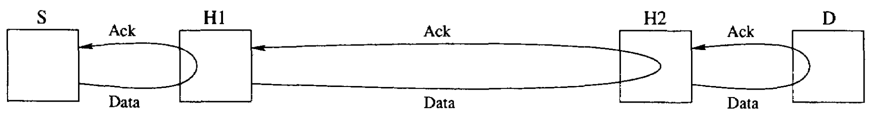

Figure 2.2 Split TCP connection with multiple HBs.

2.2 Does Split TCP Improve Performance?

If all is well for a TCP connection, the performance benefit due to a HB will be minimal or none. However if the TCP connection has the problem that is being addressed, HB's will surely help. Consider the scenario in Figure 2.3 to prove the statement.

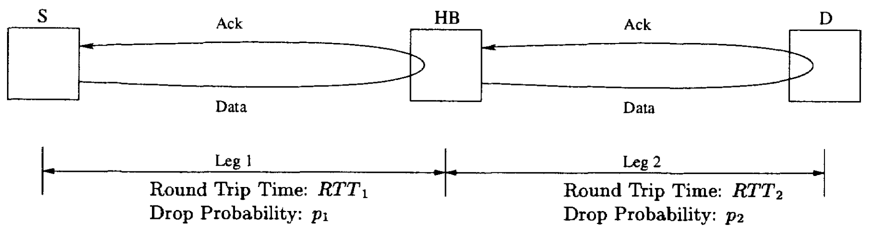

Figure 2.3 Split TCP connection with leg parameters.

In the scenario depicted, there is one HB between the end hosts. The leg between S and D, "Leg 1", has low loss with drop probability of p_1 and a large RTT, RTT_1. This leg is similar to that of a leg having a satellite or transoceanic link. The leg between the HB and D, "Leg 2", has high loss with drop probability of p_2 and a short RTT, RTT_2. This leg is similar to the final 100 or so miles of the TCP connection in a third world country.

Under normal circumstances, the throughput, *Thp*, of a TCP connection is given as

$$Thp = \frac{Flight\ Size}{RTT} \tag{2.1}$$

where "Flight Size" is the number of data packets, expressed in bytes, that the source has sent but for which no acknowledgment has been received yet. The flight size can always be determined as

$$Flight\ Size = min(Advertised\ Window, cwnd) \tag{2.2}$$

Using the "Square Root Law" [2, 1], 2.2 can be re-written as

$$Flight\ Size = min(Advertised\ Window, \frac{MSS}{\sqrt{p}}) \tag{2.3}$$

Thus substituting 2.3 for flight size in 2.1, the throughput of a TCP connection is given as

$$Thp = min\left(\frac{Advertised\ Window}{RTT}, \frac{MSS}{RTT * \sqrt{p}}\right) \tag{2.4}$$

For the network environment under consideration i.e. one having a large RTT and a high drop probability, it is quite evident from 2.4 that cwnd will be the bottleneck. Hence the rest of the discussion concentrates around cwnd and the throughput is re-written as

$$Thp \sim \frac{cwnd}{RTT} \sim \frac{MSS}{RTT * \sqrt{p}} \tag{2.5}$$

where cwnd: Congestion Window

RTT: Round Trip Time

MSS: Maximum Segment Size and

p: Probability of packet loss

This is true as long as the source host has plenty of data to send and the congestion window is the only limit on the packets in flight and the probability, p, is not too large.

Hence, the maximum possible throughput for Leg 1 would be

$$Thp_1 \sim \frac{MSS}{RTT_1 * \sqrt{p_1}} \tag{2.6}$$

(as long as the buffer in HB never fills). Similarly, the maximum possible throughput for Leg 2 would be

$$Thp_2 \sim \frac{MSS}{RTT_2 * \sqrt{p_2}} \tag{2.7}$$

(as long as the buffer in HB never empties).

If the drop probabilities p_1 and p_2 are independent of each other, the total drop probability of the end-to-end connection is

$$p_1 + p_2 - p_1 * p_2 \tag{2.8}$$

As long as at least one of p_1 and p_2 is small, we have $p_1 * p_2 \ll p_1 + p_2$. Hence we can write 2.8 as

$$p_1 + p_2 - p_1 * p_2 \sim p_1 + p_2 \tag{2.9}$$

Hence, if there were no HB in Figure 2.3, the maximum possible end-to-end throughput would be

$$Thp \sim \frac{MSS}{(RTT_1 + RTT_2) * \sqrt{p_1 + p_2}} \tag{2.10}$$

However, in the situation as shown in Figure 2.3, if either one of Thp_1 and Thp_2 is considerably larger than the other or if the buffer in HB is quite large, the effective end-to-end throughput would be

$$\begin{aligned} Eff \cdot Thp &= min\left(\frac{MSS}{RTT_1 * \sqrt{p_1}}, \frac{MSS}{RTT_2 * \sqrt{p_2}}\right) \\ &> \frac{MSS}{(RTT_1 + RTT_2) * \sqrt{p_1 + p_2}}, \end{aligned} \tag{2.11}$$

so an improvement is mathematically assured. The improvement is more pronounced in the situation described, with

$$RTT_2 \ll RTT_1 \sim RTT_1 + RTT_2 \tag{2.12}$$

and

$$p_1 \ll p_2 \sim p_1 + p_2 \tag{2.13}$$

(for example, a clear satellite link to a third world country followed by a high loss link within that country). This explanation shows that the "one helper box" solution requires the HB to be placed in the third world country.

More generally: HBs should be placed in such a way that each leg has either low RTT or low drop probability (or both).

Hence it can be concluded that using Split TCP for TCP connections having large RTT and high drop probability will result in greater performance. It should be noted that the throughput of (2.11) holds true when the HB has a large buffer. There might be situation where both the legs are fading but at non-overlapping intervals. In that situation an adequately large buffer is expected to make a large difference.

2.3 Split TCP Design Environment

In this research work, Split TCP has been designed primarily for the following network environment: There is a "campus A" in the USA and a "campus B" in an underdeveloped country, say in Africa. There is a leg from "reasonably close to campus A" to "reasonably close to campus B" that has high RTT and low packet loss. This leg could be a satellite link or a transoceanic link. From (say) the satellite earthstation on there is a leg of questionable quality in the underdeveloped country.

The theory of Section 2.2 shows that in the one HB situation, the optimal position of the HB is the place where the high RTT leg meets the high loss, low RTT leg in the underdeveloped country. One can think of that as in or close to the satellite earthstation in the underdeveloped country. However, this leaves a significant problem: How to guarantee that all the traffic between campus A and campus B flows through the HB, and that only traffic that is intended to be intercepted is indeed intercepted. These guarantees are required for maintaining the correct flow semantics, leading to improved performance, of the TCP connection at the HB.

The solution proposed in this dissertation is as follows: Place a second HB (HB_A) in campus A and a third HB (HB_B) in campus B. Call the HB at the distant earthstation HB_I (I for Intermediate). Give campus A a network address a.b.c.d/n and give campus B a network address w.x.y.z/m as shown in Figure 2.4. The IP address of all interfaces of HB_I must neither be in w.x.y.z/m nor in a.b.c.d/n.

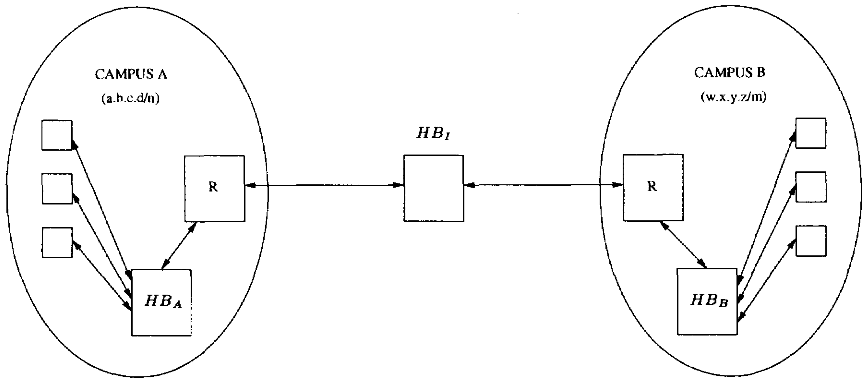

Figure 2.4 Sample real world setup using HB's.

In campus A, route all traffic destined for w.x.y.z/m through HB_A. In HB_A, embed all data packets (TCP and UDP) to w.x.y.z/m in an IP packet (IP over IP) with destination address that of HB_I. When this data packet reaches HB_I, remove the outer IP header and encapsulate the original data packet in another IP packet with destination address HB_B. At HB_B, take out the original data packet and forward it to the actual destination host within campus B. Thus, except the first and the last HB, all other HB's along a given path replace the incoming outer IP header with a new, modified IP header. Traffic in the opposite direction is handled similarly.

This mechanism guarantees that all the traffic between the two campuses will flow through the same sequence of HB's. This allows the HB to maintain the correct flow semantics and also prevents it from wrongfully registering a bypassed data packet, as a lost packet. Also, this mechanism still has the advantage that the original end hosts do not need any modifications, while at the same time between the HB's modified versions of TCP (which in this situation are implemented at the IP layer in the Linux kernel) can be used. This design will work for any number of HB's and campuses. At each HB a table mapping the destination address to either the "next HB address" or the "original destination" is maintained. This mechanism works even when packets pass through the same router or even HB twice: once unencapsulated and once encapsulated.

This mechanism will work even if the end hosts use the Authentication Header [21] of the IPsec suite of protocols [22] for authenticating the end hosts. IPsec is a commonly used mechanism for providing security services for traffic at the IP layer. It employs two security protocols: Authentication Header (AH) and Encapsulating Security Protocol (ESP) both of which could be used either in the "Transport Mode" or "Tunnel Mode". The Split TCP design that has been proposed can be used if the AH security protocol is being used.

The ESP security protocol encrypts the IP payload (TCP header + data), thus denying Split TCP direct access to the TCP header which is required for maintaining the state of the split TCP flow. For this reason, the design proposed in this dissertation will not work with the ESP security protocol.

The only changes necessary are in the forwarding tables of the routers within campus A and campus B. In case a campus uses static routing, new routes will need to be introduced in the routers (intermediate and/or gateway) and the end hosts (if need be). However, if a campus uses a routing scheme like Open Shortest Path First (OSPF) or Routing Information Protocol (RIP) (likely case), no modifications are required in any of the routers and the end hosts. The HB, say HB_A, can list itself as a router and advertise, within campus A, a very cheap route for campus B. Same can be done with HB_B, in campus B. A similar scheme is even possible for Border Gateway Protocol (BGP). With BGP, the HB, say HB_A, can list itself as the speaker node for the autonomous system to which the destination hosts belong.

CHAPTER 3

RELATED WORK

Over the years, researchers have presented work that highlighted the advantages of using Split TCP in Mobile Networks. In a mobile network, the network connection between the Fixed Host (FH) and the Mobile Host (MH) can be broken down into 2 connections. A wired connection between the FH and the base station (also know as Mobile Support Routers (MSR)) and a wireless connection between the MSR and the MH. TCP performs rather poorly in mobile networks because one of the assumptions of the TCP design is violated. TCP considers a packet drop to be an indication of congestion within the network. However this is not always true in case of a mobile network where, for example, environmental factors may cause packet drops. Irrespective of the cause, TCP will trigger the congestion avoidance phase thus reducing its current transmission rate.

Various techniques have been proposed from preventing the sender side TCP from invoking its congestion control mechanism for every dropped packet. Most of these techniques work at the physical layer. The IEEE 802.11b standard allows for the use of Media Access Control (MAC) layer acknowledgments and retransmissions. Thus a dropped packet is retransmitted by the MAC layer a specific number of times (called the retry limit) after which TCP sees the loss. Another technique used for the IEEE 802.11a standard was the use of Forward Error Correction (FEC). FEC was added to the standard to enable the receiver to identify and correct the errors made during transmission. For this, the sender would send additional data along with the primary data packet, thus eliminating the need to retransmit data packet by a substantial amount.

Most of the related work concentrate on the same principal of preventing the sender side TCP from invoking its congestion control for every packet drop. The problems of the wireless connection are separated from the wired connection by splitting the connection at the MSR. Most of the methods introduce a new protocol at the MSR and few changes in the MH. In this chapter the different methods are summarized along with their advantages and disadvantages.

An aspect of most of the protocols discussed is that they have concentrated on the network problems due to the wireless link (link between the MSR and MH). Though these problems do affect the overall performance, the network problems over the wired leg (leg between the FH and MSR) should not be discarded. This work presents a Split TCP design irrespective of the networking environment, i.e. wired or wireless. The design does not require any code modifications in the end hosts, thus making it completely transparent.

3.1 MTCP

In MTCP [17] the connection between the FH and the MH is split at the MSR by introducing a new session layer protocol. Two approaches were proposed to implement this protocol. The first approach makes use of TCP over the wireless link while the second approach uses Selective Repeat Protocol (SRP) over the wireless link. The SRP approach is similar to the SACK mechanism. A disadvantage of this method is that the network code in the MH needs to be modified.

3.2 I-TCP

In the I-TCP [5] approach, the MH sends a request to the MSR to establish a connection on its behalf with the FH. The I-TCP library is used by the MH to communicate with the MSR. One of the problems with this approach is that the applications at the MH need to be relinked with the I-TCP library. In addition to this, the networking code (kernel level) needs to be changed in the MSR to make use of the special system calls defined in the library. The I-TCP implementation also needs an I-TCP deamon running at the MSR for it to function properly. Lastly, this approach is not completely transparent with respect to the end hosts.

3.3 Snoop Protocol

The Snoop protocol [7] was the first attempt to maintain the TCP semantics by having end-to-end acknowledgments. TCP performance was increased by caching data packets at MSR for retransmission over the wireless link. However, not all data packets are cached at the MSR, creating occasional situations when the sender needs to retransmit. The Snoop protocol works well when FH acts as the sender. In order to experience comparable performance when MH acts as the sender, Selective Acknowledgement (SACK) is implemented in the MH. Because, the ACKs are end-to-end the senders window would grow slowly resulting in a low transmission rate. In order for the Snoop protocol to work, the routing code at the MH was modified by adding a new module. It also requires modifications in the TCP code of the MH in case the MH wants to initiate data transfer.

3.4 M-TCP

M-TCP [8] is meant for cellular environment and is focused on solving the problem of frequent cell exchange in addition to the bit-error rate of the wireless link, by proposing a 3-layer architecture for the mobile network. A new layer of supervisor hosts (SH) was introduced for this. A SH manages several base stations which in turn manage several mobile hosts. The connection is split at the supervisor host.

In M-TCP, a cell switch is defined as the migration of the mobile host from one supervisor domain to another. This helps curb the frequent cell exchange problem that may arise in mobile networks. M-TCP maintains the end-to-end semantics by passing the ACKs between the MH and the FH. However, to ensure that the sender does not go into the error control mechanism of TCP and reduce the congestion window during a disconnection period (or while switching cells), the SH does not forward the ACK for the last byte received. This forces TCP into persist mode in contrast to error control, thus not reducing the congestion window.

The main disadvantage of this protocol is the restructuring of the mobile network to accommodate the layer of Supervisor Host (SH)s. It also requires code modifications in the mobile hosts.

3.5 Mobile-TCP

Mobile-TCP [9] proposes to introduce a new protocol over the wireless link of the connection. Mobile-TCP advocates the use of a new compressed TCP header for data packets transmitted over the wireless link. The MSR relieves the MH from its buffer and timer management. This is done so as to reduce the processing load at the MH. The MSR also employs a different scheme for transport layer error recovery mechanism over the wireless link. On detection of a lost packet over the wireless link, the MSR retransmits all data packets sent from the lost packet on. However the error recovery mechanism of the MH remains unchanged.

This protocol also has the disadvantage of making modifications in the networking code of the MH. The MSR also incurs more software overhead by creating a timer for each outstanding data packet.

3.6 TCP Splice

TCP Splice [23], though similar to the approach being proposed in this dissertation, is not quite the same. TCP Splice concentrates on increasing the performance of web proxies by relaying the data packets at the kernel level as opposed to through the user space. The connection setup, including user authentication, between the client and the server is performed by the web proxy. The feedback, congestion control and error control mechanisms are performed by the end hosts. Hence a setup with TCP Splice may still suffer poor performance if the network conditions are as discussed in this dissertation.

3.7 Postcards from the Edge

Postcards from the Edge [24] is a "Cache and Forward" architecture designed to be used in a mobile environment. The central idea of the project is to exploit the memory at each node in the network. The project introduces the concept of a Post Office (PO) node. Each MH is associated with a PO which caches all the files destined for that MH. The PO might also cache and forward files destined for other MH's. The authors justify the need for extra storage and higher processing power because of the declining prices of the two. In order to use PO nodes, the project proposes to introduce a scheme similar to Domain Name System (DNS) that would provide a map between the MH and their respective POs.

The authors envision the project to be implemented at the transport layer. The implementation will also require addition of new protocols or modifications of existing protocols for link layer communication, link management, routing etc., that will enable the use of POs.

The approach being proposed in this dissertation uses Split TCP to overcome the network performance by providing each leg with its own feedback, congestion and error control mechanisms. Also, the connection setup including user authentication is done end-to-end. This also ensures the availability of the end hosts to each other.

Routing the data packets becomes an important issue when considering a Split connection with multiple HB's. For Split TCP to function correctly, it is important that all data packets are routed through the same set of HB's in each direction. One solution is to introduce host specific entries in the forwarding table of the routers.

Another solution, one which was explored and used, is to make use of IP over IP for HB-HB communication. This will ensure that all data packets travel through the same set of HB's. It will also give an opportunity to use a customized protocol between the HB's.

CHAPTER 4

LINUX KERNEL NETWORK INTERNALS

For the correct and complete design and implementation of Split TCP, it is essential to know and understand how a TCP/IP packet (both incoming and outgoing) is processed by the kernel. Because Split TCP provides TCP functionalities at the IP layer, this knowledge is very important and allows for the ease in duplicating, with required modifications, the relevant TCP mechanisms. It also helps find out which functions within the kernel can be called directly from the Split TCP module. It is also the goal of this dissertation to provide a complete guide for implementing and enhancing the Split TCP mechanism, for which the contents of this chapter are very useful. The chapter also acts as a good resource for someone wanting to learn about the network stack of the Linux kernel.

Since, in this dissertation, Split TCP has been implemented in the Linux kernel, a study of the network stack of the Linux kernel is provided in this chapter. The network stack of the Linux kernel constitutes nearly 20% of the total kernel code [25]. However, it still remains the least documented part of the kernel.

In this chapter the journey of a packet through the Linux kernel network stack is explained. It first explains the important data structures with respect to the network stack. This is followed by a description of the processing done by the various layers for an incoming TCP packet. The journey of a TCP packet through the various layers on its way out is then explained. The Netfilter hooks, used in the design of Split TCP, are also described in some detail. This information has largely been gained by reading and documenting the Linux kernel. For the purpose of this chapter, the findings were verified against [26]. Additional references are given in specific sub-sections.

Unless otherwise specified, the kernel source tree is assumed to be located at */usr/src/linux-n* where n is 2.6.10. For the sake of discussion, it is also assumed that the kernel is processing a TCP packet.

Over the years, the Linux kernel has gone through many revisions. It started with version 1.0 and has advanced to the current stable version of 2.6.21.3. Through each revision of the kernel, known bugs are removed, new capabilities are added and the kernel code in general is fine tuned. The project was started on version 2.4.18 and then migrated to the 2.6 family. Quite a few problems were faced during this migration primarily because of changes in structure definition, variable name, function declaration, kernel API, modification of old code, addition of new code, etc. to name a few.

In this chapter, the terms data link layer, layer 2 and L2 are used interchangeably. Similarly the terms network layer, layer 3 and L3, and the terms transport layer, layer 4 and L4 are used interchangeably.

4.1 Key Data Structures

This section describes the most critical and the most referenced data structure in the network stack, `struct sk_buff`. `sk_buff` is short for "socket buffer", also referred to as `skb` within the kernel. This structure is used to store various packet details. The kernel creates an instance of `struct sk_buff` per packet received. The packet itself is stored in a separate buffer called the packet buffer. The packet buffer is used by all the layers of the network stack to store and retrieve protocol headers and payload (data) information. `struct sk_buff` is declared in *include/linux/skbuff.h* and contains variables that represent a tremendous amount of information regarding the packet buffer and the network protocols that will process the data. Since the kernel is customizable, the structure also contains variables that are used only when a particular feature is compiled in. Figure 4.1 shows the relation between a sk_buff and its respective packet buffer.

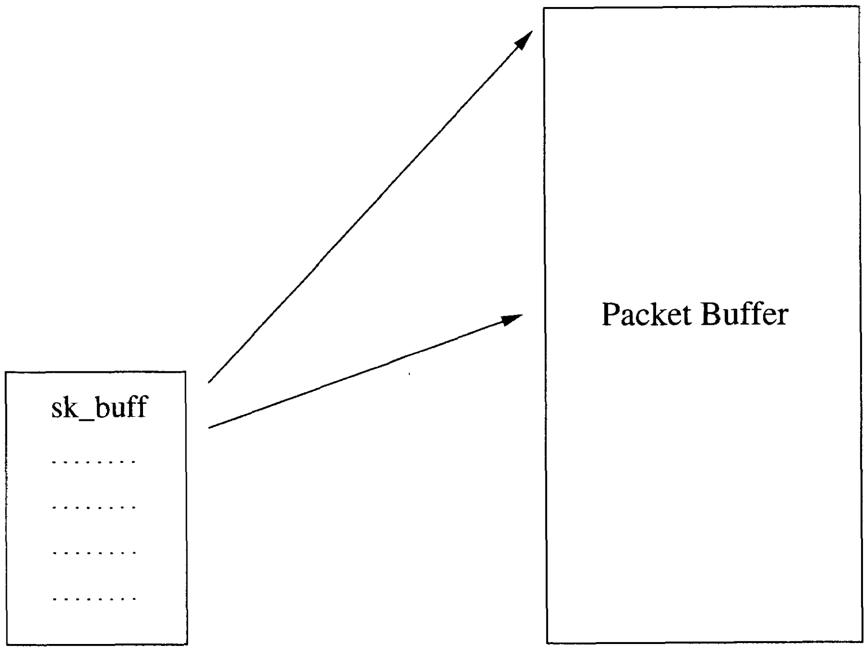

Figure 4.1 Relation between sk_buff and packet buffer.

Starting at the TCP layer, the byte stream from the application layer is broken into packets. The TCP layer creates a new sk_buff and reserves memory for it by calling the alloc_skb() function. As this buffer passes down the layers, space needs to be reserved for adding the various protocol headers. This is achieved by calling the skb_reserve() function at the start of each layer. The actual header is then added to the buffer space by making a call to the skb_push() function. For an incoming packet, starting at Layer 2, each layer needs to strip off its header before the packet is sent to the next higher layer. This is achieved by calling the skb_pull() function at the start of each layer. It is worth mentioning here, that in order to save CPU cycles, the skb_reserve, skb_push and skb_pull functions manipulate pointers declared within sk_buff as opposed to manipulating memory slabs. These functions, along with a few others, will be discussed in more detail later in this section.

The various variables of `struct sk_buff` can be classified into the following three categories:

- Layout Fields
- General Fields
- Protocol Specific Fields

The term `sk_buff` and packet are used interchangeably from now on.

4.1.1 Layout Fields

Some of the fields of the structure are used for easy access and arrangement of the packets within the kernel. The kernel arranges the `sk_buff`'s as a doubly linked list. At a given time there might be more than one (for example, for receiving, for transmitting, per CPU, etc) such linked list within the kernel. Each of these lists is identified by the head node of the list which is of type `struct sk_buff_head`. This structure is declared in *include/linux/skbuff.h* and looks as follows:

```
struct sk_buff_head {
/* These two members must be first. */
struct sk_buff  *next;
struct sk_buff  *prev;

__u32           qlen;
spinlock_t      lock;
};
```

The first 2 fields `next` and `prev`, have the conventional meaning with respect to a linked list and are used to point to the next and previous node within the list of packets. `q_len` represents the current number of packets in the list. The variable `lock` is used to prevent simultaneous access to the list.

The first 2 fields of `struct sk_buff` are the same as that of `struct sk_buff_head`, i.e. `next` and `prev`. This allows for the easy casting of `sk_buff_head` into `sk_buff` without loss of information. In order to identify the list to which a packet belongs to, `struct sk_buff` contains a variable `list` of type `sk_buff_head`. Each `sk_buff` within a given list initializes this variable to point to the head node of the list. See Figure 4.2 to understand the organization of `sk_buff`'s as a doubly linked list within the kernel.

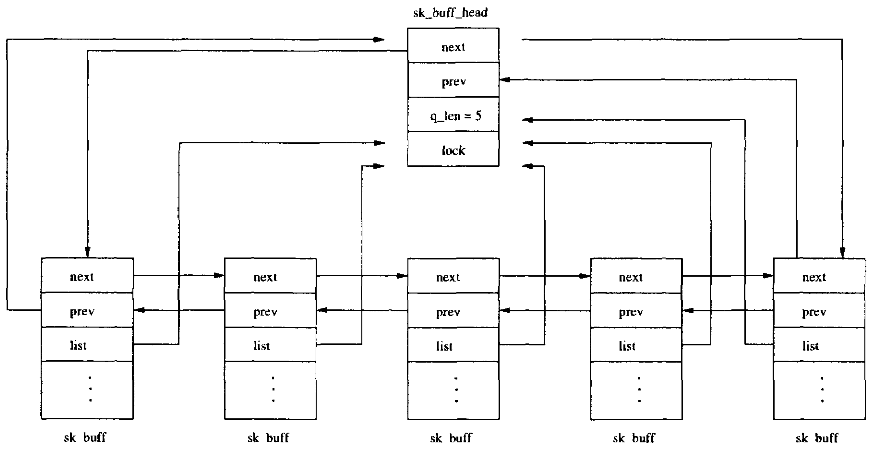

Figure 4.2 sk_buff's as a doubly linked list.

The following 4 fields represent the different lengths that are associated with a `sk_buff`.

`unsigned int len`

It represents the length of the packet as perceived by a layer. This includes the main payload, data in the fragments and all the protocol headers that have been included up to that layer. For example, at the IP layer,

$$len = IP\ header + TCP\ header + TCP\ payload + Additional\ data\ fragments$$

where either header might include options. As the packet moves across the layers, the variable is updated to represent the current relevant length.

The additional data fragments refers to the scattered memory locations that are used to store the data (payload) in case the original packet buffer had insufficient space. The kernel sometimes uses this option to add additional data in an existing packet as opposed to the method of getting a fresh chunk of continuous memory and then moving all the data to the new location. The additional data fragments do not correspond to IP fragments.

`unsigned int data_len`

It represents the number of data bytes that are stored in the additional data fragments.

`unsigned int mac_len`

It represents the length of the MAC header.

`unsigned int truesize`

It represents the total size of the buffer that was allocated when `alloc_skb` was called.

The following 4 pointers are used to mark the various boundaries of the packet.

```
unsigned char *head,
              *data,
              *tail,
              *end;
```

The `head` and `end` pointers point to the start and end location of the buffer allocated for the packet. The `tail` pointer marks the location of the last byte of the packet, whereas the `data` pointer marks the location of the first byte of the packet with respect to a given layer.

Once the buffer has been allocated, the `head` and `end` pointers are not updated as the packet moves across the layers. However, the `data` pointer is updated as the headers are either stripped off or added to the packet. See Figure 4.3 to understand the locations marked by these pointers.

As shown in Figure 4.3, the space between the `head` and `data` problem is known as the "headroom" and the space between the `tail` and `end` pointers is called the "tailroom". The various protocol headers are added in the headroom. Each layer can extend the headroom, by calling the function `skb_realloc_headroom` in case it is smaller than its header. After validating the `sk_buff` against a couple of error checks, the `skb_realloc_headroom` function calls the `pskb_expand_head` function to extend the headroom if there is insufficient space in the packet buffer. The `pskb_expand_head` function, creates a new `sk_buff` with the required headroom and then copies the bits of the old `sk_buff` to the new one.

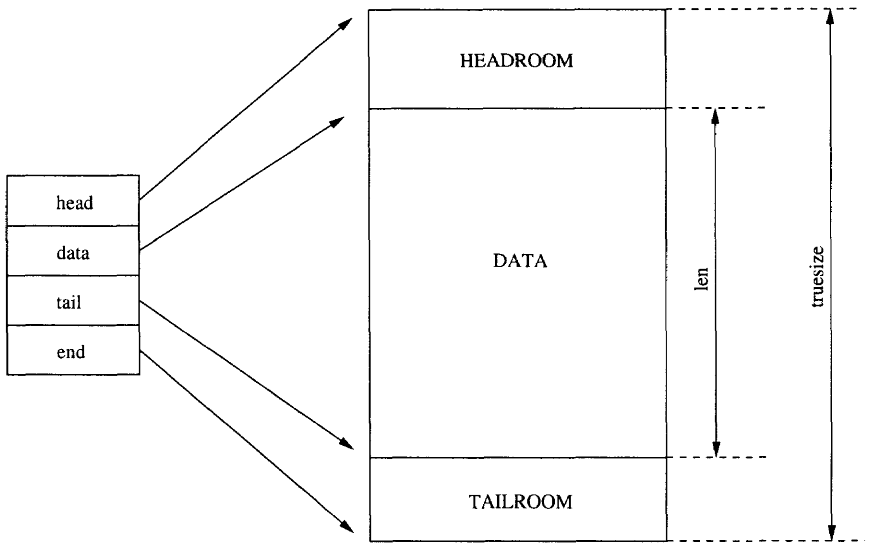

Figure 4.3 Pointers marking the boundaries of a packet.

4.1.2 General Fields

Some of the other important fields declared in `sk_buff` are as follows:

`atomic_t users`

This fields stores the number of references that have been made to an `sk_buff`. The purpose of this field is to prevent freeing of the `sk_buff` structure while someone is still using it. The memory allocated for the packet is freed when the value of this variable is 0.

This variable only covers reference count for the `sk_buff` data structure. There is a similar variable, `dataref`, that accounts for the number of processes referring a packet buffer.

`struct sock *sk`

This field represents the socket the packet belongs to. This field is initialized at the start of the TCP layer processing, as will be explained later in this chapter. The field is NULL when a packet is merely being forwarded.

`struct timeval *stamp`

This field stores the time when the packet was received by the kernel.

`struct net_device *dev`

This field has a two fold use. For packets that are received by the kernel, this field stores the information of the interface on which the packet was received. For packets that are generated locally or the ones that need to be forwarded, the field points to the interface through which the packet will be sent out.

`struct net_device *input_dev`

This field points to the interface on which the packet was received. Hence, it is NULL for packets that are generated locally.

`struct dst_entry *dst`

The details of this field are filled in by the routing algorithm of the kernel. It contains protocol independent information that is needed by the routing algorithm. The fields within `dst_entry` are used by the network layer to decide the course of action for the packet i.e. for an incoming packet, should it be routed or should it be sent to the transport layer. Similarly, it is used to store the routing details like MAC address of an interface, which function needs to be called next etc., for an outgoing packet.

Apart from these fields, there are many more fields that impart useful information regarding a `sk_buff`. For example, variables that are used to specify the priority of the packet, the type of packet, the function to be used as the destructor function for the buffer etc.

4.1.3 Protocol Specific Fields

The following fields are used to store protocol specific information for a packet.

`unsigned short protocol`

This field is used to decide which layer 3 protocol needs to process the packet. The value for this field comes from the list of protocols declared in *include/linux/if_ether.h*. The most common values are that for IP and Address Resolution Protocol (ARP).

```
union{...} h
    union{...} nh
    union{...} mac
```

These unions are used to store the header information for layer 4, layer 3 and layer 2 respectively. Within them are declared variables for the protocol headers for the different layers of the network stack. For example, within union `h` is declared a variable of type `struct tcphdr` for the TCP header. As the packet moves across the layers, the appropriate header within the respective union is populated.

`char cb[40]`

This field is like a scratch pad that can be used by each layer for storing layer specific information. It might be used to transfer information across the layers, although its main purpose is to be used within a layer. It is heavily used by the TCP code of the kernel. For example, for a received packet, the TCP code stores a copy of various TCP related sequence numbers in this buffer.

4.1.4 Supporting Functions

In the kernel are defined many support functions that are used for manipulating, retrieving information and management of `sk_buff`'s. In this section, some of the support functions that are used in Split TCP are described.

`alloc_skb`

This is the main function used to allocate memory for the packet. As seen in Figure 4.3, a packet consists of the main data buffer (packet buffer) and the `sk_buff` data structure. A call to this function will allocate memory for both of them. The packet buffer returned by this function has no headroom and a tailroom equivalent to the size of the buffer as shown in Figure 4.4. It is defined in *net/core/skbuff.c*.

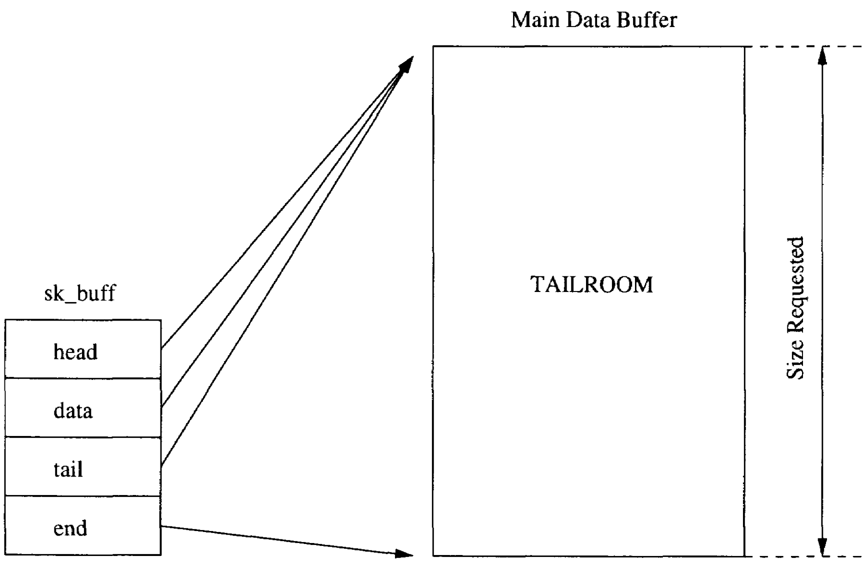

Figure 4.4 Allocating memory for a new sk_buff through alloc_skb().

`skb_copy`

This function is used to copy an entire packet i.e. the `sk_buff` data structure, the packet buffer and fragmented data if any. The copy created by the function is private to the process that called it and can be modified at will. The function is rather expensive in term of CPU cycles and should be used only when a private copy of the packet is absolutely necessary. The function is defined in *net/core/skbuff.c*.

`pskb_copy`

This function is similar to the `skb_copy` function. The only difference is that the fragmented data remains shared between the original packet and its copy. This function is preferred over the `skb_copy` function since it uses less CPU cycles. The function is defined in *net/core/skbuff.c*

`skb_clone`

A `sk_buff` data structure can be cloned using this function. The function creates a copy of only the `sk_buff` structure. The packet buffer is shared between the two `sk_buff`'s. The relevant reference count in the original `sk_buff` is incremented to reflect this situation. This function is useful when a process does not need to modify the packet buffer. The function is defined in *net/core/skbuff.c*.

`skb_reserve`

This function is used to reserve some amount of space in the headroom of the packet buffer. This is done by manipulating the `data` and `tail` pointers. For example, while creating a TCP packet this function is called as `skb_reserve(skb, MAX_TCP_HEADER)`. The outcome of this call is shown in Figure 4.5. The function is defined in *include/linux/skbuff.h*.

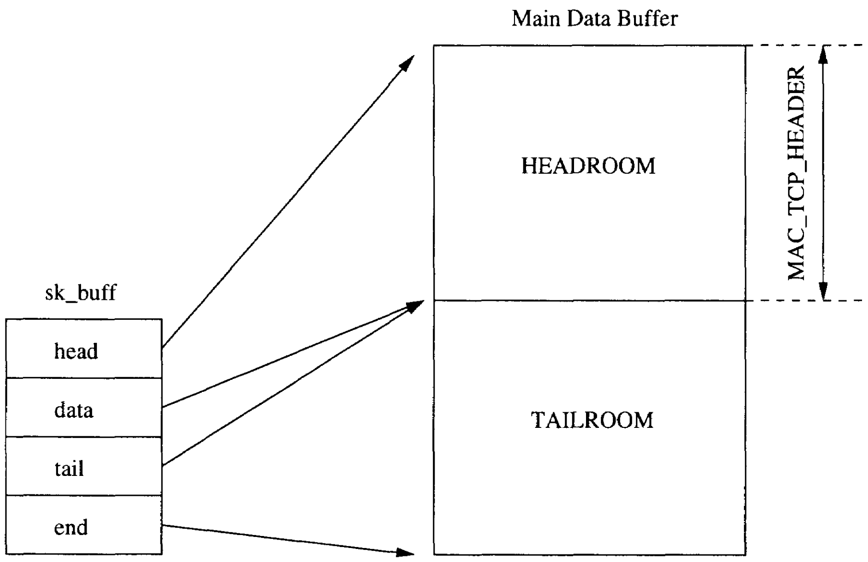

Figure 4.5 Creating headroom in the sk_buff through skb_reserve().

`skb_push`

This function is used to add data at the start of the data area of the packet buffer. The function manipulates the `data` pointer thus reducing the available headroom. For example, Figure 4.6 shows the outcome of adding the TCP header, by calling this function, to the packet buffer of Figure 4.5. The function is defined in *include/linux/skbuff.h*.

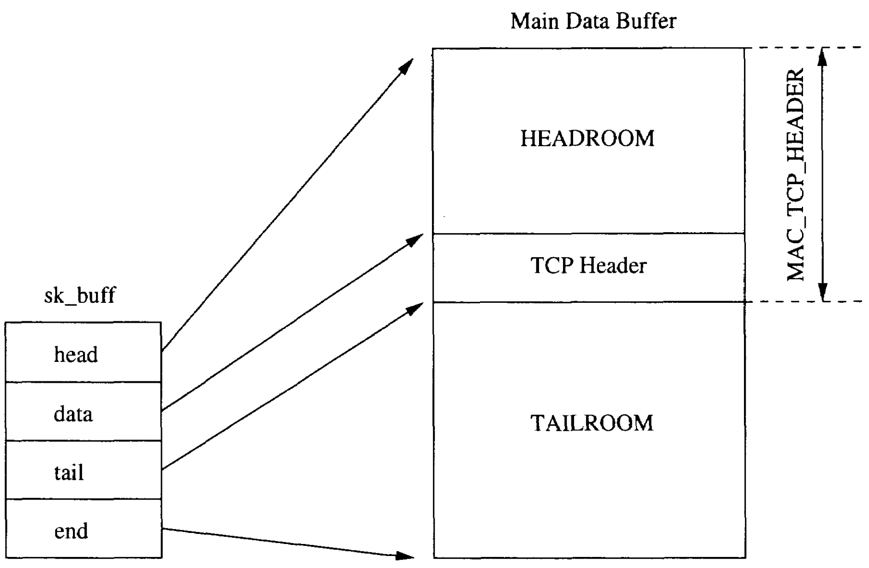

Figure 4.6 Pushing data in the sk_buff through skb_push().

`skb_pull`

This function does the exact opposite of the `skb_push` function. It removes the requested amount of data from the start of the data area of the packet buffer. The function manipulates the `data` pointer thus increasing the headroom. For example, the buffer in Figure 4.6 will look like the buffer in Figure 4.5 after `skb_pull(skb, sizeof(TCP Header))` is executed. The function is defined in *include/linux/skbuff.h.*

`kfree_skb`

This function is used to release the memory being used by a `sk_buff`. However, the function frees the memory only when the user count is 1, else it simply decrements the number of users by 1. The function is declared in *include/linux/skbuff.h.*

4.2 Incoming Packet Flow through the Kernel

The Linux kernel makes use of the layer structure, defined by the Open Standards Interconnect (OSI) model, for processing a packet. Before processing, each layer first checks the packet for errors like bad checksum. Once the packet passes all the tests, it is processed by the layer depending on the type of packet, the options that it carries, the state of the flow in case of TCP etc. This section describes the journey of an incoming packet through the network stack of the Linux kernel, starting at the Data Link layer and ending at the Transport layer. This knowledge was used for designing the flow of a packet through the Split TCP code. It also gave useful insights on the implementation of various layer specific features within the HB. One such example is the RTT estimation code.

4.2.1 First Step - Data Link Layer

The journey of an incoming packet starts at the Data Link layer or the device driver code within the kernel. The driver code used for this study is *drivers/net/sundance.c*. For the kernel to provide support for the various NIC's, many manufacturer specific network drivers are present in the kernel source tree. Even though it has not been verified, the basic algorithm for processing and handing over the packet to the next layer is the same for all of them.

The driver continuously polls the interface for incoming packets. This is done by the `rx_poll` function. Once the packet passes the error checks, a new `sk_buff` is created and stored in the "ring buffer". The driver calls the `eth_copy_and_sum` function, defined in *include/linux/etherdevice.h*, to copy the entire packet into the packet buffer. The ring buffer is a doubly linked list where `sk_buff`'s are stored as they wait to be picked up by the higher layer. The maximum number of packets that can be stored in the ring buffer is 32.

The `skb_reserve` and `skb_pull` functions are used to reserve and add data into the newly created packet buffer. The driver also records the incoming interface in the field `sk_buff->dev`. It then calls the function `eth_type_trans` (defined in *net/ethernet/eth.c*) to find out the L3 protocol which will process the packet and stores this information as shown:

```
skb->protocol = eth_type_trans(skb, dev);
```

The `sk_buff` is now ready to be handed over to the kernel code for further processing. For the purpose of this discussion, kernel code is any code that does not directly interact with the NIC. Driver code is the code that directly interacts with the interface.

4.2.2 Second Step - Intermediate Layer

The `netif_rx` function (defined in *net/core/dev.c*) is the entry point for the packet inside the kernel code. At this point, the L3 protocol that should process the packet is known. What is not known is the function which serves as the entry point for that protocol. Also, if there are multiple processors in the host, the kernel needs to allocate a processor for processing the packet. These and other operating system related tasks are handled by the functions in *net/core/dev.c* which form a transparent layer between the data link layer and the network layer.

Once the `netif_rx` function is called, the packet (`sk_buff` and the packet buffer) leaves the driver code and enters the kernel. It then becomes the kernel's responsibility to store the packet so that it can be found by the network layer. For this reason, with each processor is associated a buffer (input queue). The head node of the queue is of type `sk_buff_head` and is declared in the `sofnet_data` data structure. The `sofnet_data` structure is declared in *include/linux/netdevice.h* and looks as follows:

```
struct softnet_data
{
int throttle;
int  cng_level;
int avg_blog;
struct sk_buff_head  input_pkt_queue;
struct list_ head poll_list;
struct net_device *output_queue;
struct sk_buff  *completion_queue;
struct net_device   backlog_dev;    /* Sorry. 8) */
};
```

The first thing that `netif_rx` does is to get a reference to this input queue. If there is space within the queue (the queue can hold a maximum of 300 packets), the packet is enqueued at the tail of the queue. `netif_rx` calls `netif_rx_schedule` function, defined in the same file, before exiting.

The main task of `netif_rx_schedule` is to schedule an interrupt, Software Interrupt Request (SoftIRQ), to reflect the reception of the new packet in the `softnet_data` input queue. At regular intervals, the CPU polls the interrupts to see whether a SoftIRQ has been scheduled. If one has been scheduled, the packet at the head of the queue is picked up by the CPU for further processing. Before going further with the discussion, its worth finding out how interrupts are handled by the kernel.

There are 2 types of interrupts in the Linux kernel, Hardware Interrupt Request (HardIRQ) and SoftIRQ. Out of these, the SoftIRQ interrupts are used within the network stack. Hence the following discussion describes the SoftIRQ interrupt mechanism in some detail. The kernel provides the provision of defining 32 SoftIRQ's, some of which are defined in *include/linux/interrupt.h*. The kernel operates the SoftIRQ interrupts in what is called the "bottom halve". Bottom halves is the oldest mechanism within the kernel for scheduling work that does not need immediate attention. For packet reception and transmission, the kernel makes use of two SoftIRQ's, `NET_RX_SOFTIRQ` and `NET_TX_SOFTIRQ` respectively.

The kernel polls for pending SoftIRQ's by making use of the following code in *kernel/softirq.c:__do_softirq()*.

```
do {
if (pending & 1) {
h->action(h);
rcu_bh_qsctr_inc(cpu);
}
h}};
pending >>= 1;
} while (pending);
```

where `h` is a pointer to an array of type `struct softirq_action` and `pending` is an integer. The 32 bits of the integer variable `pending` are used to represent the 32 SoftIRQ's, one per bit. A value of 1 at the nth bit of `pending` indicates that the nth SoftIRQ is pending, where $1 \leq n \leq 32$. `struct softirq_action` consists of a function pointer `action` which points to the function within the kernel that will act as the Interrupt Service Routine (ISR) for the respective SoftIRQ. A SoftIRQ calls the `open_softirq` to populate the fields of the `softirq_action` structure during registration. For example, for `NET_RX_SOFTIRQ` and `NET_TX_SOFTIRQ`, the call to `open_softirq` looks as follows:

net/core/dev.c:net_dev_init()

```
open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

The functions `net_tx_action` and `net_rx_action` are registered as the ISR's for `NET_TX_SOFTIRQ` and `NET_RX_SOFTIRQ` respectively. Thus when the kernel gets around to processing the pending `NET_RX_SOFTIRQ` raised on the receipt of a packet, the `net_rx_action` function is called. This functions main purpose is to see if a new packet can be picked up for processing. If there are no packets waiting to be processed or the ISR has used more than 10 ms of CPU time, the function raises the SoftIRQ interrupt and exits. Else, it will dequeue a packet, find the relevant network layer function and hand over the packet to that function. These 3 tasks are done by 2 functions, `process_backlog` and `netif_receive_skb`, both of which are defined in *net/core/dev.c.*

The main task of the process_backlog function is to dequeue the packet from the softnet_data input queue and to call the netif_receive_skb function. It keeps performing these tasks as long as there are more packets in the input queue and the total time that it has been executing is less than 1 jiffie (10msec). As the packets get processed by netif_receive_skb, process_backlog updates the variables that maintain the state of the backlog packets.

The main task of the netif_receive_skb function is to find the function that serves as the entry point for the relevant L3 protocol. To understand the working of this function, one needs to first understand how a network protocol is registered with the kernel. The packet_type data structure, declared in *include/linux/netdevice.h*, is used to register an L3 protocol with the kernel.

```
struct packet_type {
__be16                  type;
struct net_device       *dev;
int                     (*func) (struct sk_buff *,
                                 struct net_device *,
                                 struct packet_type *,
                                 struct net_device *);
struct sk_buff          *(*gso_segment)(struct sk_buff *skb,
                                        int features);
int                     (*gso_send_check)(struct sk_buff *skb);
void                    *af_packet_priv;
struct list_head        list;
};
```

The main fields of this structure are type and *func. The field type stores the protocol identifier of the protocol being registered and the field *func is a function pointer pointing to the function that will serve as the entry point for that protocol.

The protocol identifiers that the kernel supports are declared in *include/linux/if_ether.h*. Each protocol registered with the kernel is added to the hash `ptype_base` which is declared at the start of *net/core/dev.c* as

```
static struct list_head ptype_base[16];
```

For the hash `ptype_base`, the kernel uses a very simple hash function, `ntohs(type)&15`. As seen from the declaration, the kernel has support for 16 network protocols. Each protocol creates a variable of type `packet_type` with the relevant information and registers itself at boot time by calling the function `dev_add_pack`, defined in *net/core/dev.c*, on that variable. For example, the IP protocol registers itself as follows:
net/ipv4/af_inet.c

```
static struct packet_type ip_packet_type = {
        .type = __constant_htons(ETH_P_IP),
        .func = ip_rcv,
        .gso_send_check = inet_gso_send_check,
        .gso_segment = inet_gso_segment,
};
.
.
.
dev_add_pack(&ip_packet_type);
```

In addition to `ptype_base`, the kernel defines another variable, `ptype_all`, which is used to handle all types of packets. This is useful for packet sniffers which need to process all types of packets. Hence, while processing the packet, `netif_receive_skb` first runs the packet through the `ptype_all` list. If any protocol of type `ETH_P_ALL` is registered with `ptype_all`, their respective functions are first executed. The function then searches the `ptype_base` variable for the network protocol whose identifier is stored in `sk_buff->protocol`. Once found, the function pointed to by the function pointer `*func` is executed, which in case of IPv4 is `ip_rcv`. In case `sk_buff->protocol` does not match with any of the registered protocols, the packet is dropped.

4.2.3 Third Stop - Network Layer

The type of the packet determines the network layer protocol that will process the packet. Since a TCP/IP packet is being considered for the purpose of this discussion, the network layer processing will be done by the IPv4 protocol. As stated in the previous section, the `ip_rcv` function, defined in *net/ipv4/ip_input.c*, is the entry point for the IPv4 protocol.

The functions starts by performing some basic checks for the IP header. These include: (from *net/ipv4/ip_input.c:ip_rcv()*)

1. Length of the packet is at least the size of an IP header.
2. Version in the IP header is 4.
3. Checksums are correct.
4. The various length fields (of `sk_buff` and IP header) are not bogus.

Once the packet passes all these checks, it enters the Netfilter system through the call to the `NF_HOOK` macro (defined in *include/linux/netfilter.h*). Since the Netfilter system is explicitly used by the Split TCP implementation, Section 4.4 is dedicated to it.

The Netfilter system calls the `ip_rcv_finish` function which is defined in *net/ipv4/ip_input.c*. This function decides the course of action i.e. should the packet be forwarded or should it be sent to the L4 protocol for further processing. The function calls upon the routing algorithm of the kernel to make this decision. The decision is then stored in the variable `dst`, which is of type `struct dst_entry`, of the `sk_buff`.

As mentioned earlier, the dst variable is used by the routing algorithm to store protocol independent information. This consists of a pointer to the outgoing interface (if applicable), a variable of type struct neighbour which stores the information of the next hop, and function pointers, input·and output, to the function that needs to be called next, etc. Depending on whether the packet needs to be forwarded or whether it needs to be sent to the transport protocol, the function pointer input points to either ip_forward or ip_local_deliver respectively. This function pointer is called from within the ip_rcv_finish function, thus executing either of the above mentioned functions.

Anticipating whether the packet in question needs to be routed, the kernel branches off to the ip_forward function. The journey of the packet through the ip_local_deliver function and into the transport protocol is explained in Section 4.2.5.

4.2.4 Fourth Stop - Forwarding the Packet

Once it has been decided that the packet needs to be routed, the ip_forward function, defined in *net/ipv4/ip_forward.c*, is called upon. This function performs the following tasks - check the packet against the IPSec policies that are in place (if any), decrease the Time To Live (TTL) field of the IP header and forward the packet to the function registered at the NF_IP_FORWARD hook. The function also does some error checking before forwarding the packet. For example, if the original value of TTL is 1, it calls the icmp_send function to create and send an Internet Control Message Protocol (ICMP) time exceeded packet back to the source.

The kernel code registers the ip_forward_finish function, defined in *net/ipv4/ip_forward.* at the NF_IP_FORWARD hook. This function processes IP options (if any) and then calls the function pointed to by the function pointer output in the dst variable of the packet. For the packet path currently being discussed, the function pointer output points to ip_output function which is defined in *net/ipv4/ip_output.c*. Depending of the size of the packet and the Maximum Transfer Unit (MTU) of the output link, ip_output calls either ip_fragment to handle packet fragmentation or calls ip_finish_output to go on to the next stage of the packet path.

The ip_finish_output function, defined in *net/ipv4/ip_output.c*, is a very small function. It updates the outgoing interface information in the packet and sets the protocol field of the packet to IPv4. It then calls the function ip_finish_output2 that is registered at the last hook of the Netfilter system, NF_IP_POST_ROUTING.

The ip_finish_output2 function is the last function to process the packet at the network layer, after which the data link layer takes over. The main task of the function is to append the data link layer header to the packet. For this it first checks to see whether the sk_buff has enough headroom space for another header. If need be, the headroom is increased. As is known from the basics of networking, a network host relies on the ARP mechanism to find the details, like MAC address, of the next hop. The kernel is no different. The details of each reachable next hop are then stored in a variable of type struct neighbour, declared one per next hop. Within struct neighbour is stored the basic information as well as a huge array of statistics regarding a neighboring host. This structure consists of a variable hh of type struct hh_cache which is used to stored the data link layer header. The fields of this variable are filled in after the host receives the response message for the ARP request that it might have sent. The hh variables, which are one per neighboring host, are saved in a linked list by the kernel. This list is known as the "L2 header cache". Since the hh variable stores the entire data link layer header in its original form, the ip_finish_output2 function first traverses the header cache to find a match. This technique helps save CPU time. If a match is not found, it calls the neigh_resolve_output function, pointed to from struct neighbour, to invoke the ARP protocol.

Irrespective of how the kernel gets the data link layer header, the packet is next sent to the dev_queue_xmit function. The dev_queue_xmit function, defined in *net/core/dev.c*, is the entry point into the data link layer when the packet is on its way out. The main task of this function is to put the packet in the queue from where the device driver picks them up and transmits them on the wire. This queue is part of the Queueing Discipline (qdisc) subsystem of the kernel which serves as an interface between the network stack and the device driver. See Figure 4.7.

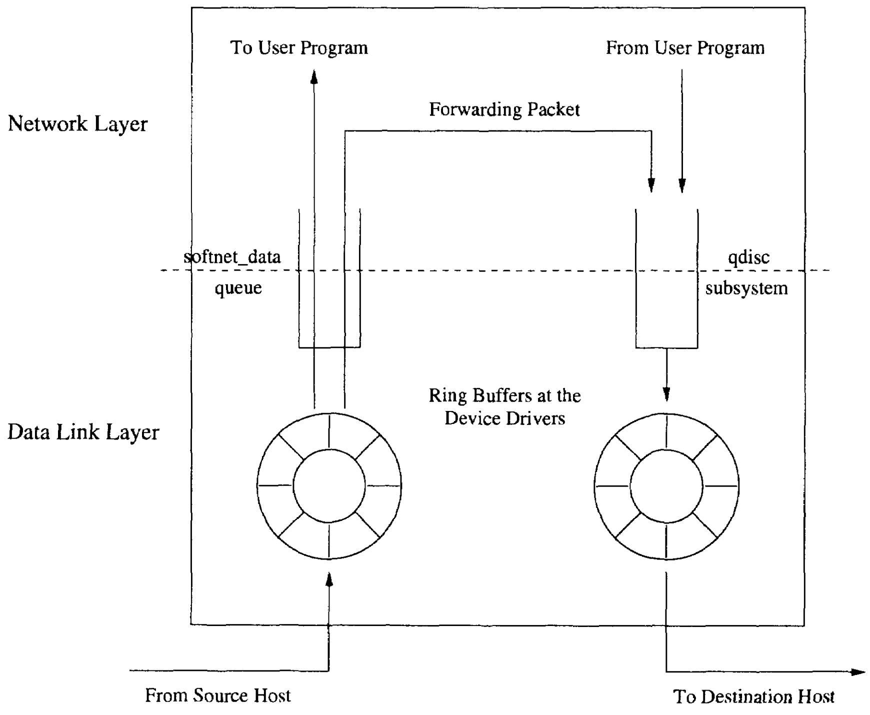

Figure 4.7 The Queuing Discipline interface.

The qdisc interface was introduced in the kernel to add advanced routing (e.g. Split access and load balancing over multiple interfaces) and traffic control capabilities (e.g. bandwidth control, policing etc) within it. This subsystem allows the use of the various queuing disciplines (First In First Out (FIFO), Stochastic Fairness Queuing etc) and traffic control algorithms (Token Bucket Filter etc) that have been proposed over the years. One can always write code to add one's own queuing discipline in the kernel. Using the qdisc interface, one can attach a single or multiple queues, where each queue may have a different queuing discipline, to a single interface. The multiple queues are arranged as a tree structure. The qdisc interface eventually calls the `tx_poll` function of the device driver which puts the packets on the wire. For more information on how qdisc are employed and used in the kernel, see [27].

Figure 4.8 shows the flow of an incoming TCP packet through important functions of L2 and L3 layer. The path of both a forwarded packet and a packet that is sent to a user program, is shown in the figure.

4.2.5 Alternate Fourth Stop - Local Delivery

In the previous section, it was assumed that the packet needs to be forwarded. However, if the routing algorithm determines that the packet is destined for this host, the kernel calls the `ip_local_deliver` function. The `ip_local_deliver` function, defined in *net/ipv4/ip_input.c* is a wrapper for the function registered at the Netfilter hook, NF_IP_LOCAL_IN. It starts by doing IP fragment reassembly (if required) followed by a call to the `ip_local_deliver_finish` function. The `ip_local_deliver_finish` function, defined in *net/core/ip_input.c*, has 2 main tasks: prepare the packet for the transport layer protocol, TCP in this case, and hand the packet to the function (also called as protocol handler) that acts as the entry point into TCP.

Before the packet is handed off to the protocol handler, the `data` pointer of the `sk_buff` needs to be adjusted so that it is pointing to the start of the L4 header. `ip_local_deliver_finish` achieves this by calling the `__skb_pull` function. It then uses the `protocol` field of the IP header to find the appropriate protocol handler as shown in the code abstract:

net/ipv4/ip_input.c:ip_local_deliver_finish()

```
hash = protocol & (MAX_INET_PROTOS - 1);

 .

 .
```

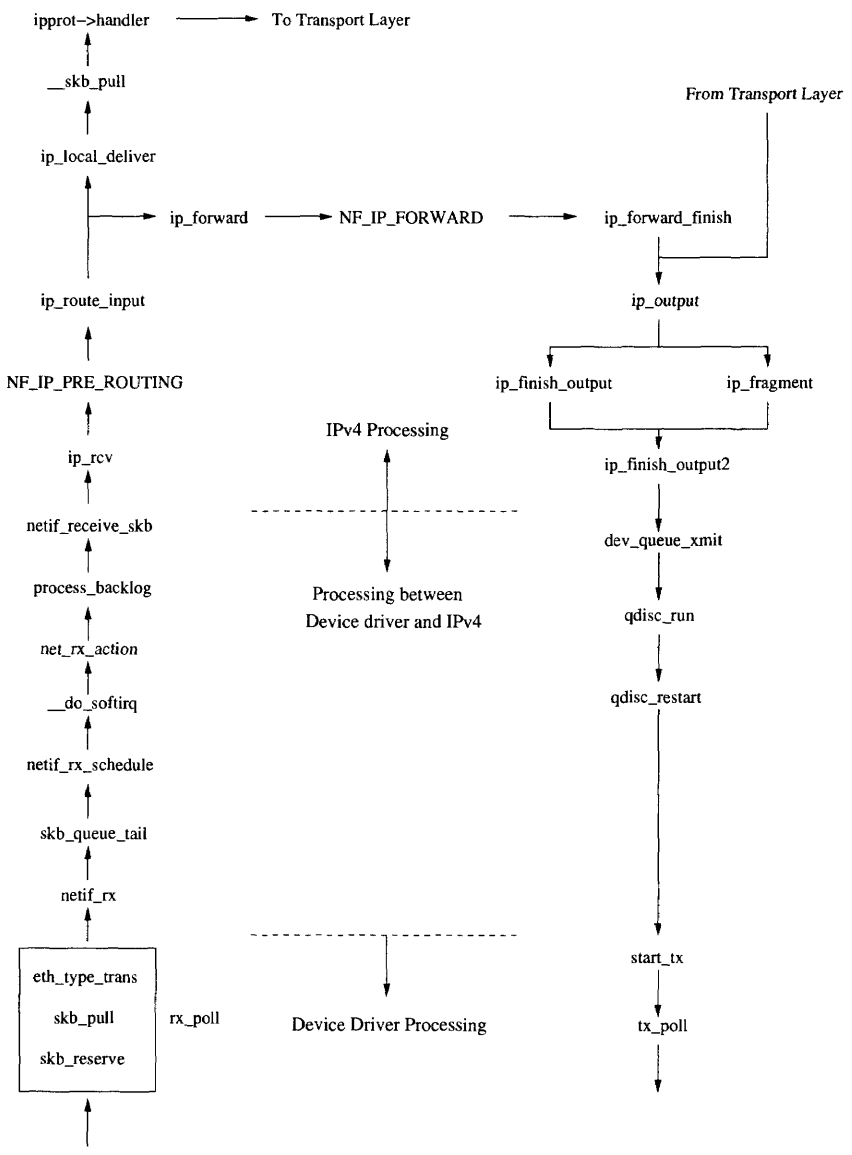

Figure 4.8 Flow of TCP packet till network layer.

```
.
if ((ipprot = rcu_dereference(inet_protos[hash])) != NULL) {
       .
       .
       .
       ret = ipprot->handler(skb);
       .
       .
       .
}
```

As seen from the code abstract, the kernel maintains an array, `inet_protos`, of the L4 protocols that it supports. The `inet_protos` is an array of type `struct net_protocol` and is declared in *net/ipv4/protocol.c*. It is used by the kernel to register a maximum of `MAX_INET_PROTOS` (=256) L4 protocols. The `inet_add_protocol` function is called by the L4 protocol to register itself with `inet_protos`. The following code abstract, taken from *net/ipv4/af_inet.c:inet_init()*, shows the User Datagram Protocol (UDP) and the TCP protocol being registered with the kernel:

```
if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
        printk(KERN_CRIT "inet_init: Cannot add UDP protocol\n");
if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
        printk(KERN_CRIT "inet_init: Cannot add TCP protocol\n");
```

The `inet_add_protocol` function, defined in *net/ipv4/ip_input.c*, takes 2 input parameters - a variable of type `struct net_protocol` and a variable containing the protocol ID of the L4 protocol. The `net_protocol` structure is used to record the function that will be used as the protocol handler. The following code abstract, taken from *net/ipv4/af_inet.c:inet_init()*, shows the values of `net_protocol` structure for TCP.

```
static struct net_protocol tcp_protocol = {
        .handler =      tcp_v4_rcv,
        .err_handler =  tcp_v4_err,
        .no_policy =    1,
};
```

Thus when the kernel makes a call to the protocol handler for the TCP protocol, the control is shifted to the `tcp_v4_rcv` function.

4.2.6 Fifth Stop - Transport Layer

The `tcp_v4_rcv` function, defined in *net/ipv4/tcp_ipv4.c*, starts off by running the packet through a few error checks like proper and consistent header length, correct checksum etc. From the protocol headers, it then notes some bookkeeping information about the TCP flow in the `cb` field of the `sk_buff` structure. Since the packet is being processed by the TCP protocol, it is understood that the final destination of the packet either is an application program or a service daemon. Both make use of sockets in order to send and receive data. Hence the `tcp_v4_rcv` function calls the `__tcp_v4_lookup` function to find the socket associated with the destination port number in the TCP header. The socket could either be in the established state or the listening state depending on whether it belongs to an application process or a service daemon respectively.

The `__tcp_v4_lookup` function, defined in *net/ipv4/tcp_ipv4.c*, is shown below.

```
static inline
struct sock *__tcp_v4_lookup(u32 saddr, u16 sport,
                             u32 daddr, u16 hnum, int dif)
{
   struct sock *sk = __tcp_v4_lookup_established(saddr, sport,
                                                  daddr, hnum, dif);
```

```
    return sk ? : tcp_v4_lookup_listener(daddr, hnum, dif);
}
```

As is seen from the code abstract, the function first calls the `__tcp_v4_lookup_established` function. This function traverses the list of established sockets to find a socket to which the packet needs to be sent. It calls the `tcp_v4_lookup_listener` function if a match was not found. The `tcp_v4_lookup_listener` function traverses the list of listening sockets to find a match. This sequence of function calls is even followed for an original SYN packet. Both these functions are defined in *net/ipv4/tcp_ipv4.c.*

If `__tcp_v4_lookup` does not return with a reference to a socket, `tcp_v4_rcv` sends a TCP reset back to the source and then frees the memory by calling `kfree_skb`. However, if the function did return a reference to a socket, `tcp_v4_rcv` checks if the packet can be processed now or at a later time. It calls `tcp_v4_do_rcv` if the packet can be processed now or calls `sk_add_backlog` to add the packet to the backlog queue for later processing.

Depending on the state of the socket, which is stored in the field `sk_state` of `struct sock`, the `tcp_v4_do_rcv` function branches off its execution into the TCP state machine. The kernel handles the `TCP_ESTABLISHED` state in a separate function as compared to the other states of the TCP state machine, which are collectively handled by one function. Within `tcp_v4_do_rcv` function, the kernel first calls the `tcp_rcv_established` function which is used to handle the `TCP_ESTABLISHED` state. The `tcp_rcv_established` function, defined in *net/ipv4/tcp_input.c*, is divided into two paths - fast path and slow path. The "fast path" is used when the TCP flow does not need to handle any special case like loss of packets, processing data with URG flag set, etc. During the fast path the following operations are performed in sequence

1. Processing of the timestamp option and calculation of the RTT based on it.

2. Processing TCP packets that have the ACK flag set. This involves updating the TCP bookkeeping variables, removing acknowledged data from the queue and advancing the congestion window if allowed.
3. Copy the payload of the TCP packet into the receive buffer of the socket to be sent to the user application.
4. Update the window that will be advertised to the remote host.
5. Schedule and/or send a TCP ACK packet (might be a delayed acknowledgment).

The "slow path" is taken by the kernel if any of the following conditions are true. *Taken from the comments in net/ipv4/tcp_input.c as it is*

```
- A zero window was announced from us - zero window probing
  is only handled properly in the slow path.
- Out of order segments arrived.
- Urgent data is expected.
- There is no buffer space left
- Unexpected TCP flags/window values/header lengths are received
  (detected by checking the TCP header against pred_flags)
- Data is sent in both directions. Fast path only supports pure
  senders or pure receivers (this means either the sequence number
  or the ack value must stay constant)
- Unexpected TCP option.
```

However, if the state of the socket is anything other than `TCP_ESTABLISHED`, the `tcp_rcv_state_process` function is called upon. The `tcp_rcv_state_process` function, defined in *net/ipv4/tcp_input.c*, starts off by checking whether the current packet is a part of the 3-way handshake for establishing a connection. If the packet is a connection request, the appropriate connection request handler is called, which in case of TCP is `tcp_v4_conn_request` (defined in *net/ipv4/tcp_ipv4.c*). If the packet has the ACK flag set, then the function processes the `TCP_SYN_RECV, TCP_FIN_WAIT1,`

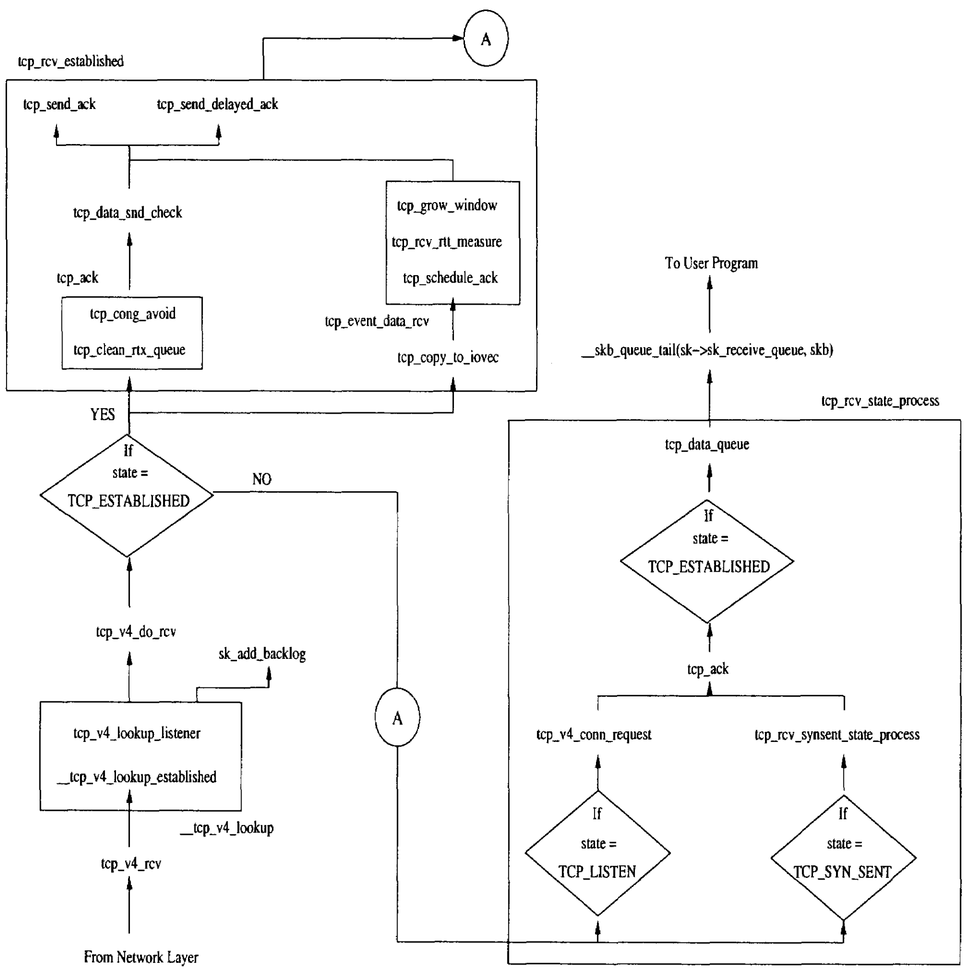

Figure 4.9 Flow of incoming TCP packet through TCP layer.

TCP_CLOSING and TCP_LAST_ACK states. While processing the TCP_SYN_RECV state, the kernel changes the state of the TCP flow to TCP_ESTABLISHED.

At the end of tcp_v4_do_rcv the packet is guaranteed to have gone through the TCP state machine and resulted in a positive outcome, like sending the data to the user application, or in a negative outcome, like a RST packet being send back to the source by the TCP state machine.

Figure 4.9 shows the flow of an incoming TCP packet through important functions of the TCP layer.

4.3 Outgoing Packet Flow Through the Kernel

The previous few sections concentrated on the journey of an incoming TCP packet. This section will describe the journey of a transmitted TCP packet i.e. an outgoing TCP packet.

4.3.1 First Stop - Transport Layer

The transport layer (TCP) receives the data from the user program or socket as a byte stream. Hence, one of the tasks of the L4 layer is to create packets i.e. allocate memory to and create data packets and `sk_buff`'s for those data packets out of the byte stream. This task is taken care by the `tcp_sendmsg` function defined in *net/ipv4/tcp.c*. The `sk_buff`'s, once created, are stored in the queue, `sk_write_queue`, associated with the socket. While creating packets, the kernel can either append the data into an existing packet buffer or it could fill the data in a new packet buffer. For the former option, the `tcp_sendmsg` function checks the last packet in `sk_write_queue` to see if the buffer has room for additional data. For the latter case, the `sk_stream_alloc_pskb` function is called upon to create and allocate memory to a new packet buffer and its corresponding `sk_buff`.

Once a reference to the appropriate packet buffer has been attained, `tcp_sendmsg` gets busy with copying the data into the packet buffer by calling the `skb_copy_to_page` function, defined in *include/net/sock.h*. It also updates the various pointers and counters of the `sk_buff`. Once the packet has been created, the `__tcp_push_pending_frames` function, defined in *include/net/tcp.h*, is called upon to check if it or any of the packets enqueued before it can be transmitted. If enabled, `__tcp_push_pending_frames` controls the flow of the TCP packets through Nagel's algorithm. The outcome of the algorithm is stored in the bookkeeping flags. These flags are then used by the `tcp_write_xmit` function. The `tcp_write_xmit` function, defined in *net/ipv4/tcp_output.c* and called from `__tcp_push_pending_frames`, traverses the `sk_write_queue` in sequence and picks packets for transmission if allowed by Nagel's algorithm and by the window. Since the queue is maintained and works in a FIFO manner, `tcp_write_xmit` has a while loop that always picks up the packet at the head of the queue for transmission. If selected, it calls the `tcp_transmit_skb` function and advances the pointer that points to the head of the queue.

The main task of the `tcp_transmit_skb` function, defined in *net/ipv4/tcp_output.c*, is to create and populate the TCP header for the packet being processed. For a TCP data packet, it calls the `tcp_select_window` function to calculate the window that will be advertised and the `tcp_build_and_update_options` functions to update (if required) and append any TCP options that have been negotiated for the TCP flow. `tcp_transmit_skb` gets called any time the kernel needs to transmit a TCP packet (normal transmission, retransmission, probing etc).

Once the TCP header has been populated, `tcp_transmit_skb` hands the packet off to the L3 layer by calling the respective protocol handler. This protocol handler is pointed to by the function pointer

```
tp->af_specific->queue_xmit
```

where `tp` is of type `struct tcp_opt` and `af_specific` is of type `struct tcp_func`. For IPv4, `queue_xmit` points to the function `ip_queue_xmit` as shown in the following code abstract:

net/ipv4/tcp_ipv4.c

```
struct tcp_func ipv4_specific = {
        .queue_xmit     =       ip_queue_xmit,
        .send_check     =       tcp_v4_send_check,
        .rebuild_header =       tcp_v4_rebuild_header,
        .conn_request   =       tcp_v4_conn_request,
        .syn_recv_sock  =       tcp_v4_syn_recv_sock,
        .remember_stamp =       tcp_v4_remember_stamp,
        .net_header_len =       sizeof(struct iphdr),
        .setsockopt     =       ip_setsockopt,
        .getsockopt     =       ip_getsockopt,
        .addr2sockaddr  =       v4_addr2sockaddr,
        .sockaddr_len   =       sizeof(struct sockaddr_in),
};
```

4.3.2 Second Stop - Network Layer

As mentioned in the previous section, the protocol handler for IPv4 is the `ip_queue_xmit` function. The `ip_queue_xmit` function, defined in *net/ipv4/ip_output.c*, starts off by checking if the packet is routable. If it is, `ip_queue_xmit` calls the `ip_route_output_flow` function to invoke the routing algorithm of the kernel and populate the `dst` field of the packet with the route details. The next step is to create, populate and add the IP header to the headroom in the packet. The `ip_options_build` function is called to add details of any IP options that are being used by the TCP flow. Once the IP header has been created, the kernel enters the Netfilter system through the `NF_IP_LOCAL_OUT` hook. Depending on the type of packet, the routing algorithm registers the appropriate function at the `NF_IP_LOCAL_OUT` hook. For example, for a multicast packet the `ip_mc_output` function is registered at the hook, whereas for a regular TCP packet, `ip_output` function, is registered at the hook.

The `ip_output` function, defined in *net/ipv4/ip_output.c*, performs just one task. If the length of the packet is more than the MTU of the outgoing link, the function calls the `ip_fragment` function to perform IP fragmentation. Else, it calls the `ip_finish_output` function, which as discussed in Section 4.2.4, prepares the packet to be sent to the function registered at the `NF_IP_POST_ROUTING` hook. From this point on, the packet follows the same path as described in Section 4.2.4.

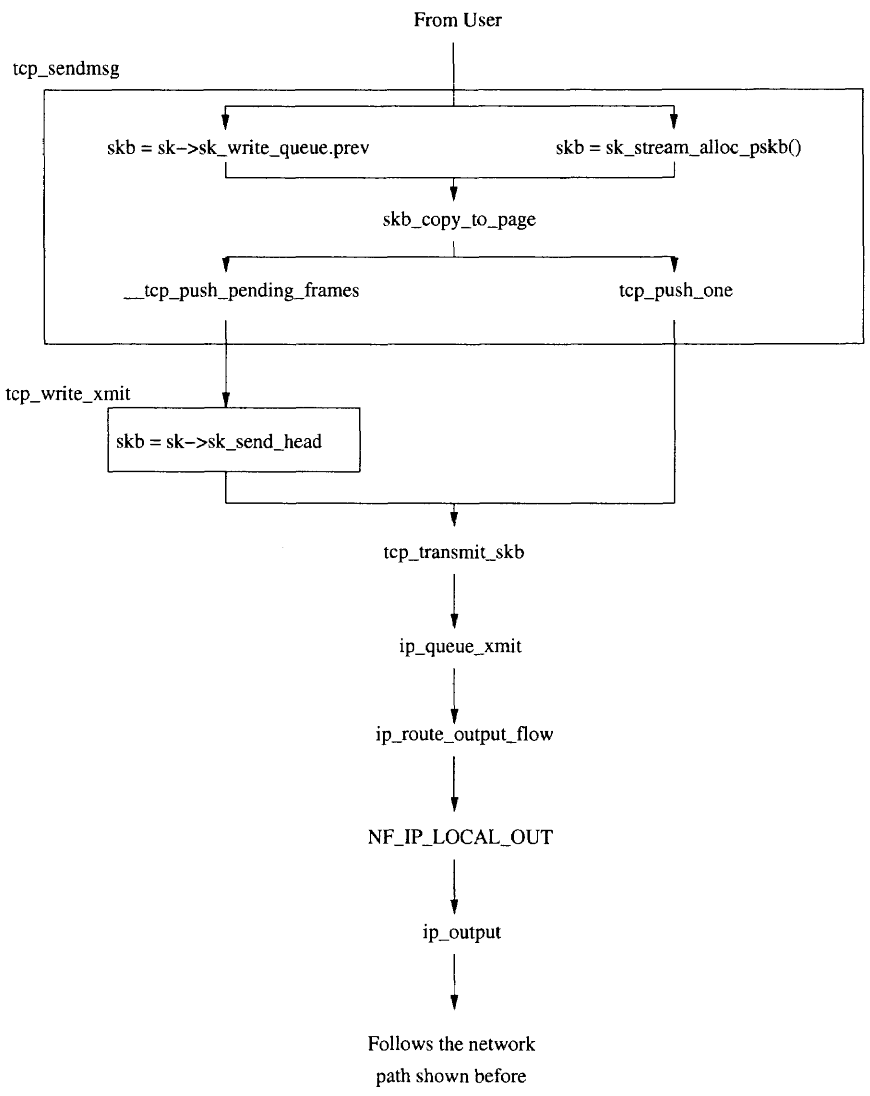

Figure 4.10 Flow of outgoing TCP packet through TCP layer.

Figure 4.10 shows the flow of an outgoing TCP packet through important functions of the TCP layer after which it follows the flow an outgoing TCP packet as shown in Figure 4.8.

4.4 Netfilter System

The material in this sub-section was gathered by studying the kernel code. It has also been verified against [28].

The Netfilter subsystem of the Linux kernel is used to load the Split TCP code in a HB. The purpose of the Netfilter system is to intercept the network traffic at various points within the network layer to perform network operations like packet filtering, Network Address Translation (NAT) and connection tracking. This is possible through the strategically placed "hooks" in the network layer. The hooks are placed so as to intercept the traffic flowing through the 3 possible paths in the network layer *viz.* packets destined for this host, packets that will be routed and packets that are generated locally. See Figure 4.11. These hooks are places where the kernel can, either compiled in or in the form of a loadable module, register functions to be called at the occurrence of specific network events. For IPv4, Netfilter defines 5 hooks as shown in Table 4.1. Figure 4.11 shows the Netfilter hooks as they appear and work within the Linux kernel.

Table 4.1 Netfilter Hooks for IPv4

Hook Name	**Network Event**
NF_IP_PRE_ROUTING	Before routing decisions are made
NF_IP_LOCAL_IN	If the packet is destined for this host
NF_IP_FORWARD	If the packet needs to be forwarded
NF_IP_LOCAL_OUT	Packets coming from a local socket
NF_IP_POST_ROUTING	Before packet is sent on the wire

As mentioned earlier, the function associated with a particular hook could either be compiled in the kernel source tree or can be loaded at a later time as a kernel module. For the first option, at the occurrence of the network event, the kernel calls the `NF_HOOK` macro which takes 2 input parameters - the name of the function that should be called at the occurrence of a network event and the name of the hook

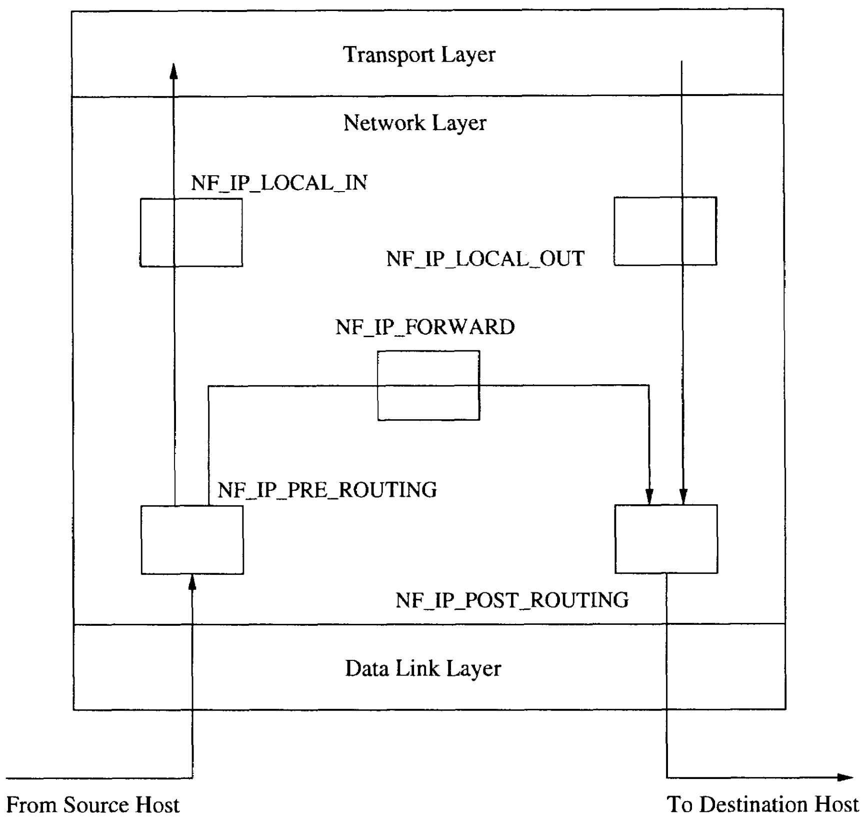

Figure 4.11 Netfilter subsystem for IPv4.

associated with that network event. Shown below is an example call to this macro from the ip_rcv function.

```
return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL, ip_rcv_finish);
```

Here ip_rcv_finish is the function to be called for the NF_IP_PRE_ROUTING hook. For the latter option, the function should be explicitly registered with the hook in question. The nf_register_hook function, defined in *net/core/netfilter.c*, is used for this purpose. This function takes as an input a variable of type struct nf_hook_ops. This structure is declared in *include/linux/netfilter.h* and has the following declaration.

```
struct nf_hook_ops
{
```

```
struct list_head list;


nf_hookfn *hook;

struct module *owner;

int pf;

int hooknum;

int priority;

};
```

Apart from the field `list` all others are initialized by the user and are explained below:

`hook`
: This is a function pointer initialized to the function that needs to be executed when the network event related to the hook occurs.

`owner`
: This field points to the module from where the `nf_register_hook` function is being called.

`pf`
: This field represents the protocol family for which the hook is being registered. As mentioned earlier, the Netfilter subsystem can be used by any network layer protocol.

`hooknum`
: In this field is stored the hook for which the function is being registered.

`priority`
: This field represents the priority of the function being registered. The Netfilter subsystem allows multiple functions to be registered with a single hook. At the occurrence of the respective network event, these functions are executed one by one in ascending order of their priority.

Once the module is registered for a particular hook, for each packet passing through the hook the callback function is called. At the completion of its execution the module returns a verdict for the future of the packet to the kernel. The possible return values are shown in Table 4.2.

Table 4.2 Possible Return Values for a Module

Return Value	**Meaning**
NF_ACCEPT	Continue with normal kernel processing
NF_DROP	Drop the packet silently. Do not process it anymore
NF_STOLEN	The module has taken over the packet. Do not process it anymore
NF_QUEUE	Enqueue the packet to userspace
NF_REPEAT	Repeat or call this hook again

When multiple functions are registered with a hook, the kernel calls upon the `nf_iterate` function, defined in *net/core/netfilter.c* to iterate through the functions and execute them in ascending order of their priority.

The HB is designed to be dual functional i.e. depending on the packet, it can either act as a regular router or it can provide Split TCP processing. This decision needs to be made for each packet received by the HB. As seen from Fig. 4.11, of the 5 hooks defined for IPv4, the function registered at the NF_IP_PRE_ROUTING hook will be executed for each incoming packet. Thus making the NF_IP_PRE_ROUTING hook the ideal location to place the Split TCP code in the Linux kernel. Hence the Split TCP design proposed in this research works at the network layer, IP in this case.

CHAPTER 5

DESIGN OF SPLIT TCP

This chapter describes the design details of the Split TCP mechanism in the HB. It also discusses the pros and cons of various viable design options, as well as the reasoning behind the design that was chosen.

As mentioned before, the Split TCP mechanism at the HB splits the end-to-end TCP connection into 2 almost independent TCP connections. It intercepts all TCP/IP and TCP/IP over IP (IPIP) traffic between the source hosts and the destination hosts, irrespective of the direction of data flow. For the purpose of Split TCP discussion, a source host is defined as the host from which the HB receives the first SYN packet. The Split TCP mechanism caches the data packets and forwards them to their respective destination hosts while at the same time sending ACK packets back to the source host. For each of the legs, Split TCP maintains the feedback, error control and congestion control TCP mechanisms.

Figure 5.1 shows the network scenario used for describing the Split TCP mechanism. It also shows the terminology used in this and the following chapters.

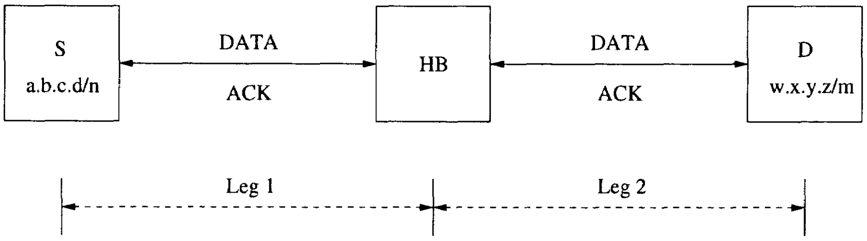

Figure 5.1 Assumed network scenario.

5.1 Split TCP Design Options

Before starting with actual implementation of the Split TCP mechanism in the HB, the following questions need to be answered:

1. Whether to implement a "Cache and Forward" mechanism or a mechanism which processes the packets on individual legs in parallel.
2. Whether to implement the connection establishment and tear down phase of a TCP connection end-to-end or on individual legs.
3. Whether to implement the Split TCP mechanism at the User level or the Kernel level.

A Split TCP like mechanism in the HB could work in several ways. In a very naive strategy, a file being transferred between the two end hosts could be transferred in its entirety over the first leg before it can be forwarded onto the next leg. This strategy will force the mechanism to wait for the FIN and FIN-ACK exchange of the previous leg, Leg 1, before the HB forwards the data on the next leg, Leg 2. It will break most of the services that have an interactive component in them. Even FTP will be broken since it has an interactive component in its control channel (port 21). For Split TCP to work in this mode, the HB will need to maintain a large buffer for transferring large files.

This strategy is also known as "Cache and Forward". An example of this strategy is the "Postcards from the Edge" project [24] which is geared more towards the mobile environment. In this project each MH is associated with a PO which caches all the files destined for that MH. At the same time a PO might also need to cache and forward files destined for other MH's. The need for the extra storage and higher processing power is justifiable because of the declining prices of the two.

The implementation chosen in this dissertation is more intuitive and performance based. In this strategy, the two network events of receiving and forwarding of a data packet are overlapped. The Split TCP mechanism will process an incoming data packet on Leg 1 (e.g. remove acknowledged packets from the buffer, do RTT calculations etc) while at the same time forward data packets on Leg 2 if the buffer is not empty and if allowed by the various windows (advertised window and congestion window). Since these operations are being performed in parallel, this strategy results in better performance while using less buffer space as compared to the previous approach. However, having a large buffer is still preferred for handling temporary mismatches in transmission rates of the legs, possibly due to high drop probabilities or fading channels. The mechanism should calculate the advertise window based on the free buffer space to prevent the HB from getting overwhelmed with data packets. This approach is also friendly to services having an interactive component.

A second design question that needs to be answered is whether to implement the connection establishment and tear down phase of a TCP connection end-to-end or individually over each leg. In the former approach, the HB will act like a regular router and shall simply forward the SYN and FIN packets between the end hosts. In the latter case, for each leg, the HB can negotiate special TCP options depending on the network conditions of that leg. The Split TCP mechanism designed in this work combines the two approaches to get maximum performance improvement. During the connection establishment phase, the mechanism mostly makes the HB act as a regular router. Thus, the connection establishment takes place end-to-end. During this phase the SYN and SYN-ACK packets are simply forwarded, but with some modifications. These modifications include negotiation of performance enhancing TCP options like window scaling and ECN and calculation of minimum possible MSS for the entire path. The end-to-end SYN exchange also allows for appropriate reaction in case the connection is refused (e.g.: destination or port unreachable). This is explained in more detail in Section 6.1. Unlike the connection establishment phase, the connection tear down is done over individual legs, independent of each other. Thus the TCP connection of Leg 1 may potentially close before the TCP connection of the Leg 2.

A third design question, one which is also raised in [18], that needs to be considered is whether to design Split TCP at the user level or at the kernel level. Each of these options has their own merits and demerits. Because of readily available resources on network programming, the user level design is more intuitive and easier to implement. It has the advantage of having a large buffer space for the Split TCP mechanism to work with. This buffer space is only limited by the available hard disk space in the HB. However, a serious drawback is that the design is not very suitable for services having an interactive component. The design will also waste time (CPU cycles) undoing and redoing packetization (copying to and from the user space) and hence may not be considered real time. There might also be a need for handling (reading, storing and writing) TCP flags of each TCP packet.

The kernel level design, although difficult, is more efficient and is the chosen approach. This approach allows the Split TCP mechanism to work with packets as compared to byte streams. Thus, there is no need for repacketization of data, thus saving CPU time. This approach also helps in preserving the TCP flags at no extra effort. It also preserves packet boundaries between different legs. This is important, for example, when the URG or PSH flag is set. The only limitation of this approach is the buffer space, which is limited by the available main memory in the HB. The Split TCP mechanism uses the main memory to store the data packets and the bookkeeping information for each TCP flow that it splits.

An advantage of the Split TCP design of this work is that it is completely transparent with respect to end hosts. Hence, no code modifications are required at the end hosts. The design can be extended to include special TCP mechanisms between the HB's (if ≥ 2 in a connection).

In this research, the Split TCP mechanism has been designed to run at the kernel level in Linux. The Split TCP software resides at the Network layer in the network stack of the HB as shown in Figure 5.2.

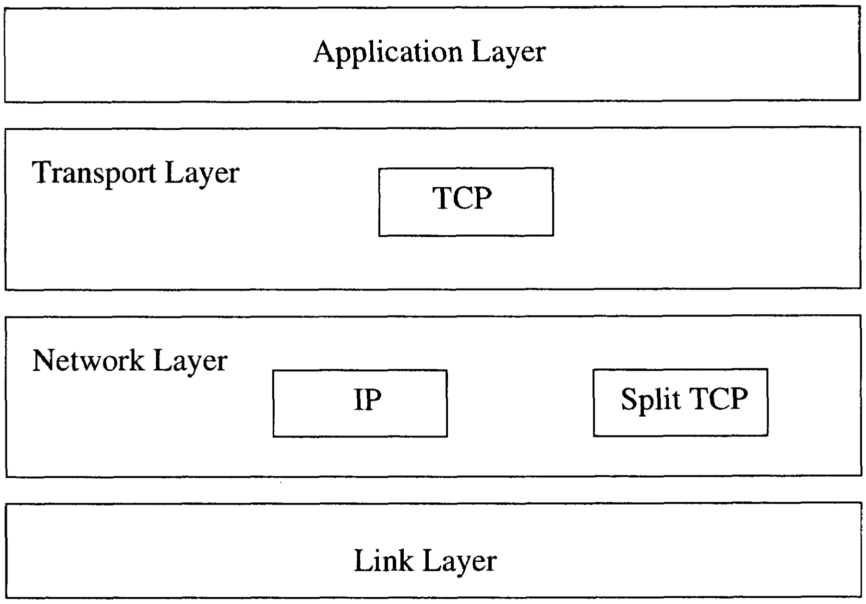

Figure 5.2 Network stack of the helper box.

The mechanism has been implemented as a Linux Kernel Loadable Module (LKLM) which is registered at the Netfilter hook, NF_IP_PRE_ROUTING. This will be discussed in more detail in Section 6.1.

5.2 Helper Box Design and Features

The main component of the Split TCP mechanism is the HB. The design chosen frees the end hosts of any code modifications and ensures minimal configuration changes (if any) in the end hosts and the networks to which they belong. For the HB to function properly, all TCP packets of a flow between the end hosts should be routed through it. This helps the HB in maintaining an accurate state of the TCP flow. As mentioned in Chapter 2, the HB makes use of IP over IP to communicate with other HB's to guarantee this.

The HB's were chosen to be Linux computers that satisfy the following two criteria: First, they have enough real time capacity to handle a sizeable number of connections between the campuses. And second, they should also have a large amount of main memory to handle such large number of flows. A large main memory also helps in buffering data packets in case Leg 2 is congested or if there is a mismatch in the network capabilities of Leg 1 and Leg 2. The available buffer space of each flow is used by the HB to calculate the window to be advertised in the acknowledgments, thus implementing back pressure.

The HB is designed to intercept all network traffic (i.e. TCP, ICMP, UDP, etc) passing through it. Every IP packet whose source and destination address belongs to the special IP address pools of the campuses, are picked up for special processing. All other packets are simply routed by the HB as if it were a regular router. This ensures that all traffic other than the one requiring Split TCP processing does not get dropped by the HB. In other words, the HB's are designed to be dual functional, thus reducing the need for additional components within the network.

For each TCP flow that the HB splits, it sets aside a pair of packet queues, one per direction of data flow. These queues are used to cache the TCP data packets in the respective direction until they are forwarded and acknowledged by the destination host. In addition to the data queues, the HB also maintains the state of the TCP flow in each direction. This is necessary for the accurate operation of the HB. In Section 6.1 the details of these design option are discussed.

An important feature that needed to be designed for the HB was Error Recovery. As mentioned previously, the HB acts as a proxy for both the source host and the destination host (or previous or next HB). Hence it needs to simulate the error recovery mechanism of both. When acting as a proxy for the source host, the HB maintains the Retransmission Timeout (RTO) timer and does the necessary RTT calculations. The expiration of the RTO timer indicates a lost packet, forcing the HB to retransmit the data packet and enter the error recovery phase. The TCP NewReno algorithm [29] has been implemented to handle the congestion avoidance phase of the HB.

However, there can be a packet loss in the backward leg of the Split TCP connection. The ability to accept data packets after the lost packet has been introduced in the HB. Just like a destination host does, the HB keeps sending duplicate acknowledgments until it receives the lost packet(s) that increase the acknowledgment number. The HB has been designed not to forward any data packet that lies after a lost packet in the packet queues. Thus, a HB is designed to re-order or re-sequence the data packet, transmit them in order and retransmit as needed.

A lacking feature of the current Split TCP implementation is that it cannot handle overlapped packets. In the current implementation, if the starting sequence number of the packet is less than the sequence number that the HB is expecting, the packet is dropped, even if the packet contains new data.

5.3 Design Components

The TCP protocol has different functions and mechanisms that are specifically defined for the source and the destination host. For an end-to-end connection, these mechanisms work at the respective end hosts. Since the HB acts as a proxy for the source host and the destination host, the source specific and destination specific TCP mechanisms needed to be integrated and made to work together in the HB. The following four components were designed to accomplish this:

1. Flow Tracker
2. Packet Queues
3. Statekeeper
4. TCP Finite State Machine

These components collectively maintain the state of an end-to-end TCP connection at the HB. For each TCP connection that the HB splits, it creates an instance of the Flow Tracker. The Flow Tracker, as the name suggests, is used to track the end-to-end

TCP connection. It contains within itself a pair of Statekeepers, one for Leg 1 and one for Leg 2. Similarly, the Flow Tracker contains a pair of Packet Queues, one per leg.

The Statekeeper contains variables that represent the state of the various TCP mechanisms at any given moment. For example, it contains variables for the sliding window protocol, RTT calculation, RTO estimation, window scaling, etc.

The Packet Queues buffer data packets for a particular direction of data flow. This helps the HB to re-sequence or re-order the data packets before they are forwarded towards the destination host. At any given time, the Packet Queues contains data packet that have been received but not forwarded yet as well as packets that have been forwarded but not acknowledged as of yet.

These will be discussed in more detail in the next chapter, Chapter 6.

CHAPTER 6

KERNEL IMPLEMENTATION OF SPLIT TCP

This chapter describes the kernel level implementation details of the Split TCP mechanism in the HB. As mentioned before, Split TCP has been implemented as a Linux Kernel Loadable Module (LKLM) which is registered at the NF_IP_PRE_ROUTING hook of the Netfilter system in Linux. The chapter also discusses the algorithm that is used to process the different type of TCP packets.

The terminology shown in Figure 5.1 is used in this chapter while discussing the implementation details of the Split TCP mechanism.

6.1 Helper Box Implementation

The Split TCP mechanism can be implemented on any operating system whose network stack can be modified. Since most of the operating systems (like Microsoft, Cisco etc) are proprietary, Linux was chosen for implementing Split TCP. Hence, all the HB's are Linux boxes. The networking code of Linux has been around for several years now and the author, like several others, found it to be a stable platform, both for experiments and regular use. One can look at the results at [30] to quantify the stability of the Linux kernel. Moreover, the source code for Linux kernels is freely available at [31] and the author does have experience in modifying the Linux source code. All these factors favored Linux as the ideal choice for this work.

The Split TCP mechanism implemented at the HB consists of 3 files. The main algorithm is implemented as a LKLM. The other 2 files contain supporting function and variable declarations and are compiled in the kernel source tree. These 3 files are described below:

- split_helper.h

This file contains the declaration of various bookkeeping variables that are used to maintain the state of various TCP mechanisms like feedback, error control, congestion control etc at the HB. It also contains the definition of the buffer structure that caches the data packet in flight. Various data structures that are used to organize multiple split TCP flows in the HB are also defined here.

This file is located at */usr/src/linux/include/linux/*.

- split_helper.c

  This file contains various function that are used to ease the use of the data structures defined in *split_helper.h*. These function implement various operations, like initialization, enqueue, insert, dequeue, search and memory cleanup. Some accessor and mutator functions are also implemented.

  This file is located at */usr/src/linux/net/ipv4/*.

- ip_in_intercept.c

  This file is the LKLM and contains all of the TCP/IP related packet processing required for the Split TCP mechanism. It is registered at the NF_IP_PRE_ROUTING hook of the Netfilter subsystem. The LKLM makes use of the data structures and functions defined in the previous 2 files along with the ones defined in the kernel.

  This file can be located anywhere with the file system. However, since its a LKLM, the users home directory is the best place.

As mentioned previously, since the LKLM is registered at the NF_IP_PRE_ROUTING hook, the HB is able to intercept and process (insofar required) all packets before they are handed to the IP and TCP layers. The LKLM implementation was chosen for its ease in integrating with the kernel without the need to recompile the entire kernel. The LKLM is ~3200 lines of C code, while the 2 kernel files together are ~750 lines of C code.

To help with the proper functioning of the HB and the ease of data organization, 4 components were designed and implemented. These are the Flow Tracker, Packet Queue, State Keeper and the TCP State Machine. Each of these components will be described in detail in the later sections.

The general operation of the HB is shown as a timing diagram in Figure 6.1. In the current implementation of the HB, a TCP flow from the source end host (in campus A or campus B) to the destination end host (in campus B or campus A respectively) is split into 2 independent TCP flows at the HB. At the receipt of each non-duplicate "original SYN" packet, the HB creates an instance of the "Flow Tracker" component, thus creating an instance per TCP flow that it splits. The HB caches the SYN packet in the respective "Packet Queue" before forwarding a copy toward the destination host. As shown in Figure 6.1, the HB does not send a SYN-ACK packet back to the source host on behalf of the destination host. However it waits for the SYN-ACK packet from the destination host, which when received, is forwarded to the source host. From this moment on, the HB starts acting as a proxy for either host thus sending and acknowledging the last ACK of the connection establishment phase. Once the connection has been established on Leg 1 and Leg 2, the HB acknowledges, caches and forwards the data packets from the end hosts. Buffering of the data packet helps in possible retransmission and data rate mismatch if any. From each data packet received, the HB extracts information that is used to update the variables in the respective "State Keeper". Once the source host has sent all the data packets, it initiates to close the connection by sending a FIN packet. The HB caches the FIN packet and responds with a FIN-ACK packet. This allows the TCP flow in either leg to close irrespective of the state of the other. At a later time, when all the data packets have been forwarded, the HB will forward the FIN packet thus initiating to close the connection of Leg 2. At the receipt of the FIN-ACK packet, the HB frees the memory occupied by the various components for the split TCP flows.

In the following sections the 4 components designed for the HB are explained.

6.1.1 Flow Tracker

All information concerning a flow between the end hosts is stored in this component. The HB creates an instance of this component for each non-duplicate original SYN

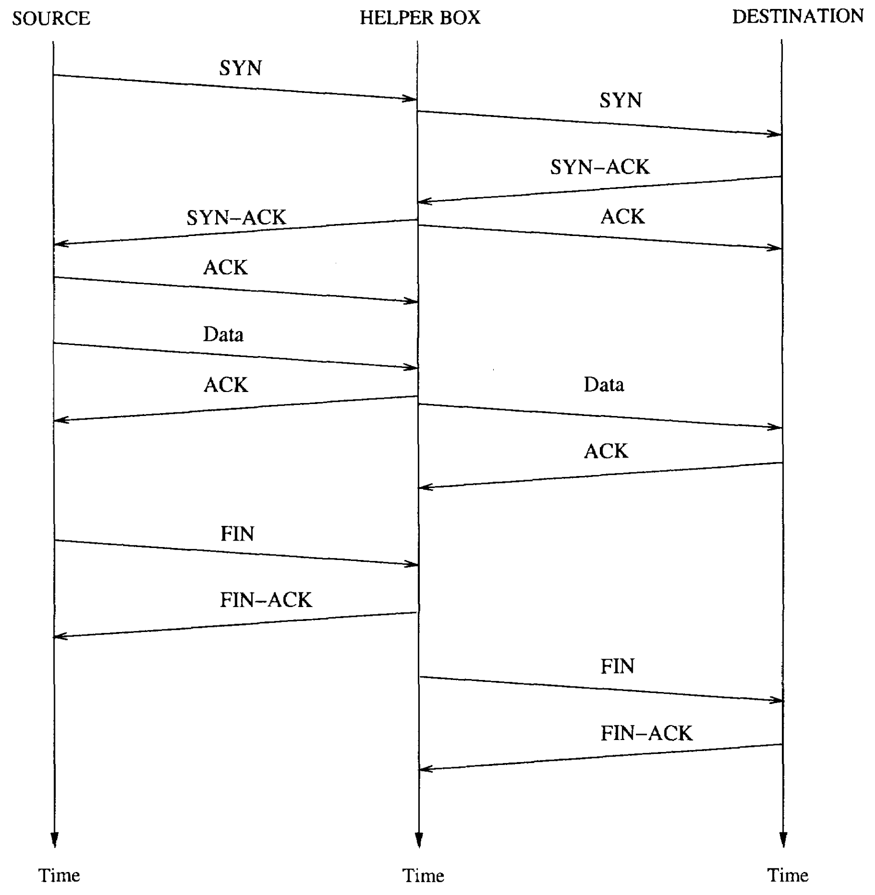

Figure 6.1 Timing diagram of HB operation.

packet that it receives for the supported end hosts IP address pool. The HB deletes an instance of the flow tracker and frees the memory associated with it, once all data packet including the FIN packets have been forwarded and acknowledged by the end hosts. Thus, an instance of the flow tracker is associated with each TCP flow that the HB splits. All such instances are arranged in a doubly linked list as shown in Figure 6.2.

Within the code, the flow tracker node is represented by `struct split_flow_info`. The linked list shown in Figure 6.2 is organized similar to the way linked lists are organized in the kernel i.e. the first node of the linked list acts as the head of the list and contains almost no useful information. The head of this list is called `sfi_list_head` within the code. Starting from node '1' the nodes within the list

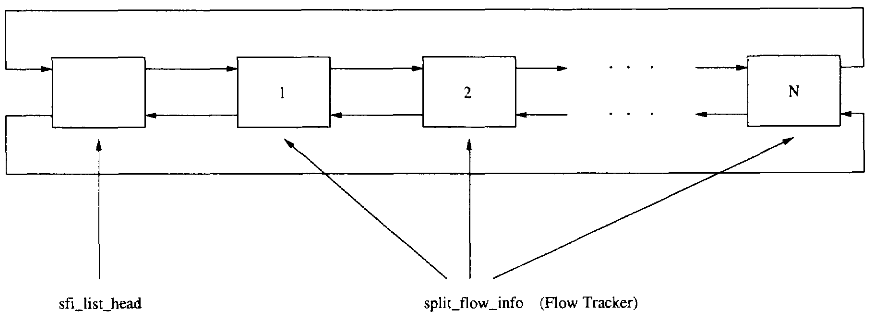

Figure 6.2 Arrangement of flow tracker nodes in the HB.

represent the TCP flows that the HB has split. For each TCP packet that the HB picks for processing, it traverses the list to search for an existing flow tracker node. If found, a pointer to the node is returned else the HB creates a new instance and adds the node to the tail of the list. Currently there is no limit on the maximum possible length of this list. Thus, there is no upper bound within the code to the number of TCP flows that the HB can split.

The structure `split_flow_info` is declared in *split_helper.h* and is shown below.

```
struct split_flow_info {
    struct split_flow_info *next;
    struct split_flow_info *prev;
    struct flow_detail *i2r_flow,
                       *r2i_flow,
                       *rep_in_flow;

    struct skbuff_list *i2r_queue;
    struct skbuff_list *r2i_queue;

    struct tcp_opt *prev_tp_opt;
    struct ethhdr *prev_flow_hw;

    struct ethhdr *fwd_flow_hw;


    struct tcp_state *lhs_tcp_state;

    struct tcp_state *rhs_tcp_state;


    int buff_clamp;

    int buff_curr;

};
```

In addition to the pointer variables `next` and `prev`, which have the traditional meaning with respect to linked list, the rest of the variables fall under 4 categories.

Of type `struct flow_detail`

This structure, as the name suggests, contains information that is used to identify a TCP flow. A TCP flow can be identified using the source host IP address, destination host IP address, source port number and destination port number.

As is seen from the declaration, the flow tracker consists of a pair of variables of type `flow_detail`. The variable `i2r_flow` contains the flow details for Leg 1 while the variable `r2i_flow` contains the details for the TCP flow of Leg 2.

This structure is declared in *split_helper.h*.

Of type `struct ethhdr`

This structure represents the Ethernet header of a packet. The main detail stored in this structure are the next hop source and destination MAC address of a leg. These values are used by the packet processing code while creating and appending the Ethernet header to the packet.

The flow tracker consists of a pair of variables of type `ethhdr`, one per leg. The variable `prev_flow_hw` contains the Ethernet header details for Leg 1 while the variable `fwd_flow_hw` contains the Ethernet header details for Leg 2.

This structure is declared within the kernel at *include/linux/if_ether.h*.

Of type `struct skbuff_list`

An instance of this structure represents a packet within the Split TCP mechanism. As will be explained later, within `skbuff_list` is a variable that points to the actual `sk_buff`. Instances of this structure are arranged in a doubly linked list to form the packet queues.

Since a TCP connection can potentially be bi-directional, each flow tracker node consists of a pair of variables of type `skbuff_list`. This pair represents the head node of the packet queues. The variable `i2r_queue` is the head node for the packet queue for Leg 1 and the variable `r2i_queue` is that for Leg 2.

This structure is defined in *split_helper.h*. the data packets.

Of type `struct tcp_state`

This structure is used to record and maintain the bookkeeping information for the various TCP mechanisms. The variables declared within it maintain the right sequence numbers, congestion window, advertised window, RTO logic, identify duplicate ACK's etc. This structure represents the statekeeper component of the HB design.

The flow tracker consists of a pair of variables of type `tcp_state`. The variable `lhs_tcp_state` maintains the TCP semantics for the backward leg and the variable `rhs_tcp_state` maintains the TCP semantics for the forward leg.

This structure is defined in *split_helper.h*.

In addition to these variables, the flow tracker contains two variables, `buff_clamp` and `buff_curr`, that represent the amount of free memory available for the per leg TCP flow. The variable `buff_clamp` stores the total memory that will be allocated for a TCP flow and the variable `buff_curr` represents how much of that memory is currently being used. This memory is used while caching the data packets and does not account for the memory being used by the various structures.

Figure 6.3 represents a flow tracker node at any given instance. As can be seen from the figure, the flow tracker node acts like a container and contains instances of the packet queues and the statekeepers within it.

6.1.2 Packet Queues

As shown in Figure 6.3, for each TCP flow that the HB splits, a pair of buffers, called Packet Queues, are created. One queue is used to store the data packets sent from the source host to the destination host and the second queue is used to store the data packets in the opposite direction. A copy of each new data packet handled by the

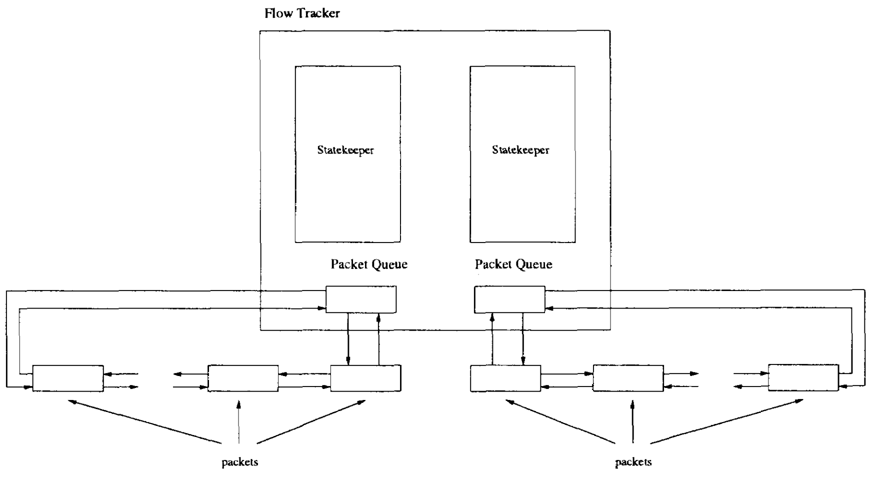

Figure 6.3 An instance of flow tracker.

HB is created and added to the respective queue. The HB traverses these queues and forwards the data packets to the destination host. It keeps a copy of the data packet until it receives an acknowledgment for the data in the packet from the next HB or end host. Any data packet that has been acknowledged by the next HB or the end host can, and soon will, be deleted from the buffer. In case the HB receives 3 duplicate acknowledgments or there is a time out, it retransmits the packet from the buffer. This approach of buffering helps the HB to isolate the network problems of Leg 1 from Leg 2.

The packets within the buffer are arranged as a circular linked list and work in the FIFO manner. The packet queue organization is similar to that shown in Figure 6.2. Within the code, the head node of the packet queues are labeled `i2r_queue` (for data flowing from source host to destination host) and `r2i_queue` (for data flowing from destination host to source host). All the nodes within the list, including the head node, are of type `struct skbuff_list`. This structure is declared in *split_helper.h* and is shown below.

```
struct skbuff_list {
     struct skbuff_list *next;
```

```
        struct skbuff_list *prev;


        struct skbuff_list *pkt_bfr_hole;

        struct sk_buff *sb_pkt;


        struct tcp_state *tps_ptr;

        __u32 snd_tstamp;

        int hole_in_queue;

        int pkt_state;

        int pkt_count;

        rwlock_t lock;

};
```

Some of the important variables of this structure are explained below:

`sb_pkt`

This variable stores the copy of the data packet that needs to be cached in the buffer. Except the head node, for each node within the packet queue this variable points to an instance of a `sk_buff`. The variable is initialized to NULL for the head node.

`pkt_state`

This variable is used by the HB to figure out whether a packet has been forwarded or not. The value of the variable can either be SENT or NOT_SENT. The variable is manipulated for each node within the packet queue except the head node. For the head node, the variable in initialized to SENT.

`hole_in_queue`

This is a boolean variable which is set (=1) if there are non-contiguous packets in the buffer. A packet is considered to be non-contiguous if the following condition holds true for it

$$\begin{aligned} &\textit{Current Packet Sequence Number} > \\ &\textit{Previous Packet Sequence Number} + \textit{Previous Packet Data Bytes} \end{aligned}$$

If this condition is found to be true for any data packet in the queue, the `hole_in_queue` variable of the head node of that packet queue is set to 1. Thus, the Split TCP mechanism needs to check the value of `hole_in_queue` for only the head node to see whether there is a hole in the packet queue.

The `hole_in_queue` variable of all other nodes within the packet queue is untouched.

`pkt_bfr_hole`

This variable points to the packet just before the first lost packet (packet hole) within the buffer. Thus it helps speed up the processing (finding the right place to insert within the buffer) of an incoming data packet after a lost packet. As is the case for the `hole_in_queue` variable, the `pkt_bfr_hole` of only the head node of a packet queue shall contain a valid pointer. Thus the Split TCP mechanism will read the `pkt_bfr_hole` variable of only the head node of a packet queue to get a pointer to the packet just before the lost packet.

The `pkt_bfr_hole` variable of all other nodes within the packet queue is initialized to NULL and remains untouched.

`pkt_count`

`pkt_count` is an integer variable and stores the number of data packets present in the packet queue. The value of this variable is incremented by 1 for each data packet added to the queue and decremented by 1 for each data packet that is acknowledged and released from the queue. The `pkt_count` variable is manipulated only for the head node of the packet queue.

Since each node of the packet queue contains a copy of the `sk_buff`, the packet boundaries and the TCP flags are maintained. As mentioned earlier, this enables the use of Split TCP even for interactive applications. At any given time, a queue will contain data packets that have arrived but have not been forwarded yet, as well as packets that have been forwarded but not acknowledged by the next HB or destination host.

6.1.3 Statekeeper

The Statekeeper is used to maintain the state of a TCP flow that the HB splits. Like the packet queues and as shown in Figure 6.3, there is a pair of statekeepers for each end-to-end TCP flow (one statekeeper for each direction of data flow). To enable the HB act as a proxy for both the source host and the destination host, many TCP mechanisms have been implemented. Each of these mechanisms rely on various variables for them to operate correctly. Within the statekeeper are stored all such variables. Variables specific to Split TCP were also introduced in the statekeeper. The statekeepers and the packet queues work together to maintain the transparency of the HB with respect to the end hosts.

Within the code, the statekeeper is represented by `struct tcp_state`. The structure is declared in *split_helper.h*. The corresponding structure within the Linux kernel is `tcp_opt`.

Figure 6.4 shows the traffic flowing in one direction i.e. from the host in campus A to the host in campus B. A similar setup exists for the traffic in the opposite direction. However, for simplicity of language only one direction of traffic is discussed.

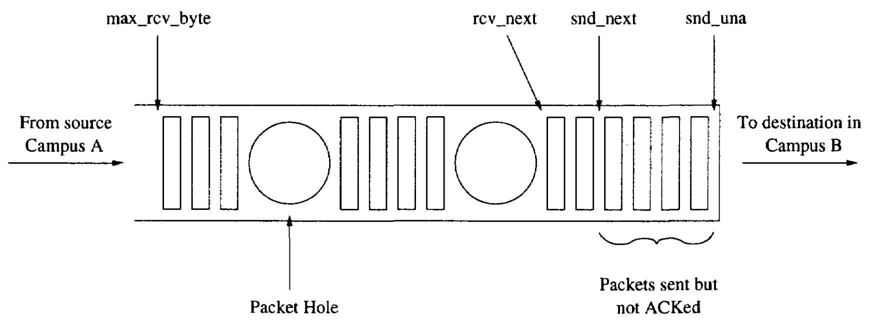

Figure 6.4 Packet Queue instance and important statekeeper variables.

The above figure also shows some of the variables that are used to implement the flow control mechanism of TCP. An important variable that is updated while receiving data packets from the source host is

$$rcv\_next$$

The *rcv_next* variable corresponds to the sequence number of the byte that the HB is next expecting from the source host. This helps the HB in sending correct acknowledgments to the source host. It also helps in identifying an out of sequence data packet. This variable is accessed by the HB when acting in the capacity of a destination host.

Important variables that the HB maintains for forwarding the data packets while acting in the capacity of a source host are

$$snd\_next, snd\_una$$

snd_next contains the sequence number of the byte that needs to be forwarded next to the destination host. *snd_una* contains the sequence number of the oldest byte unacknowledged by the destination host. For a classical TCP connection, these 3 variables are associated with the receive queue at the destination host and the send queue at the source host respectively. However for Split TCP, these 3 variables are associated with the same packet queue within the HB. The se variables ensure the proper exchange of data packets between the end hosts and the HB [32].

Another variable, specific to Split TCP, that the HB maintains is

$$max\_rcv\_byte$$

max_rcv_byte contains the sequence number of the highest byte that the HB has received from the source host. It helps when the HB is dealing with lost packets from the source host. It also helps in maintaining the correct sequence number in the acknowledgments once the HB receives all the lost packets.

These variables are declared in `struct tcp_state` as shown below.

```
__u32 rcv_next;

__u32 snd_next;

__u32 snd_una;

__u32 max_rcv_byte;
```

Another important TCP functionality is the congestion control mechanism. Over the years many algorithms have been suggested. In this work, the NewReno algorithm [29] is chosen to implement the congestion control mechanism of TCP. The variables

$$cwnd, ssthresh$$

of the statekeeper are used to control the flow of data during the congestion phase of TCP. The variable *cwnd* is increased once per RTT during the congestion avoidance phase. The variable *ssthresh* is initialized to 4 times the negotiated MSS for the TCP flow, as stated in [32], and then modified during fast recovery. The variable

$$recover$$

is introduced to implement the NewReno modification for the Fast Recovery algorithm [29]. These variables are declared in `tcp_state` as shown below.

```
__u16 ssthresh;
__u32 cwnd;
__u32 recover;
```

The RTO and RTT estimation algorithm is another source end TCP functionality that helps in dealing with lost packets. The RTT estimation algorithm proposed by V. Jacobson is implemented [33]. The following variables, declared in `tcp_state` are used to estimate the RTT and RTO.

```
__u32 rtt_seq;
__u32 srtt;
__u32 mdev;
__u32 mdev_max;
__u32 rttvar;
__u32 rto;
struct timer_list rto_timer;
```

Of the many TCP options, the window scaling option is implemented in this work. As mentioned in Chapter 2, for the scenario under consideration, the throughput of the TCP connection largely depends upon the advertised window and the congestion window and is often the minimum of the two. The advertised window is calculated based on the free buffer space available for the TCP flow. Since the TCP header field *window* is 16 bits long, the advertised window cannot grow beyond 65535 ($2^{16} - 1$). Thus for networks that support large data rate and offer favorable network conditions, the advertised window becomes the limiting factor, resulting in lower TCP throughput. The window scaling option was designed to remove this constraint and it allows the advertised window to grow up to 1 Gbyte. The following variables, declared in *split_helper.h*, are used to implement the window scaling option.

```
char wscale_ok;
__u8 snd_wscale;
__u8 rcv_wscale;
```

`wscale_ok` is a boolean variable which when set (=1) allows the use of the window scale option between the end hosts of a particular leg. It is set (or unset) during the connection establishment phase. For a given leg, `snd_wscale` stores the window scaling factor used by the remote host (end host or HB), while `rcv_wscale` stores the window scaling factor used by this HB. The HB's are configured to use the window scaling option by default.

Another TCP/IP mechanism that is implemented in the HB is Explicit Congestion Notification (ECN). A classical TCP connection gradually increases the window size thus allowing the queue at the bottleneck router to grow. Once the queue is full, the router will start dropping packets thus causing the end hosts to half their congestion window. The ECN mechanism promotes the idea of marking the packets as opposed to dropping them as the queue at the router builds up. This timely feedback from the router allows the end hosts to gradually slow down its sending rate thus proactively avoiding network congestion. The following variables, declared in *split_helper.h*, implement ECN in the HB.

```
int ecn_capable;

int ece;

int cwr;

int ce;

int ecn_flags;

int demand_cwr;

int do_cwr;

int prev_do_cwr;
```

These variables provide ECN functionality for both ___ in which the HB operates i.e. while acting as a proxy for the source host and while acting as a proxy for the destination host.

`ecn_capable` is a boolean variable which when set (=1) provides ECN capability for a particular leg (HB to HB or HB to an end host). The variables `ece`, `cwr` and `ce` correspond to the ECN-Echo (ECE), Congestion Window Reduced (CWR) and the Congestion Experienced (CE) bits of the TCP and IP headers. `demand_cwr` is a boolean variable and is used when the HB acts as a proxy for the destination host. If set (=1), the CE bit is marked in all packets being forwarded (data packets) or being send (ACKs) towards the source host. Similarly, `do_cwr` is a boolean variable and used when the HB acts as a proxy for the source host. If set (=1), the congestion window for the forward leg, Leg 2 in case of a 1 HB scenario, is reduced and the CWR bit is set in packets being forwarded (data packets) or sent (ACKs) towards the destination host.

As explained in Chapter 2, IP over IP is used for transporting the data packets between two HB's. The following variables, declared in the statekeeper `tcp_state`, are used for this purpose.

```
__u32 next_hb_addr;
int forwarding_option;
```

`forwarding_option` dictates whether the data packets should be wrapped in another IP header or not. And `next_hb_addr` contains the IP address of the next hop HB. As mentioned previously, a statekeeper is used to maintain the semantics of a particular leg. Hence the `next_hb_addr` always corresponds to the next hop HB of the leg whose state is maintained by that statekeeper instance. `forwarding_option` is set to IP_OVER_IP and `next_hb_addr` contains the IP address of that HB. Else it is set NO_IP_OVER_IP and `next_hb_addr` to NULL. Because of these two variables, a HB can have difference next hop HBs for different flows.

A TCP connection goes through various stages during its lifetime. These stages and the transition between them is governed by the TCP state machine. Split TCP works in a similar fashion and its state machine closely simulates the classical TCP state machine. In order to record the current state of the TCP connection of each leg, the variable

$$state$$

is used. The various states of the Split TCP state machine and the use of this variable will be explained in Section 6.1.4.

6.1.4 Split TCP State Machine

Just like classical TCP, a Split TCP connection works according to a finite state machine. The finite state machine tracks the various stages of a Split TCP connection. It also details the events that allow the connection to move between different states. This helps the HB to execute the proper code while processing a packet. Unlike the packet queues and the statekeepers, that work on individual legs, the state machine works on the two legs collectively. It is closely modeled around the classical TCP state machine, but with fewer states.

The Split TCP finite state machine is shown in Figure 6.5 and is modeled using the following 7 states:

- TCP_LISTEN: HB is waiting for a SYN packet.
- TCP_SYN_RCVD: HB received a SYN packet.
- TCP_SYN_SENT: HB forwarded the SYN packet.
- TCP_SYNA_SENT: HB received a SYN-ACK packet.
- TCP_CONNECTED: Each leg has an established connection. Data transfer in progress.
- TCP_FIN_SENT: HB forwarded a FIN packet.
- TCP_FIN_WAIT: HB waiting on a missing data packet even though it received a FIN packet
- TCP_CLOSING: Both legs are done with data transfer.

At any given moment, the TCP flow is in one of these states. Here, a TCP flow is defined as the TCP connection of a leg. Hence it might happen that the TCP connection between the HB and the source host and the TCP flow between the HB and the destination host are in different states at a given instance.

For the purpose of this discussion, the TCP flow between the HB and the source host (or previous HB) is called "F1" and the TCP flow between the HB and the destination host (or next HB) is called "F2". It is always assumed that the first original SYN packet was received by the HB on "F1". Initially, the TCP connection of both F1 and F2 is in the TCP_LISTEN state. This is the initial state of a flow during which the HB is waiting for a non-duplicate SYN packet. When the first non-duplicate original SYN packet is received on F1, the state of F1 changes from TCP_LISTEN to TCP_SYN_RCVD. Once the HB forwards the SYN packet on F2, the state of F2 is changed to TCP_SYN_SENT. At this moment the HB waits for the corresponding SYN-ACK packet from the destination host, upon the receipt of which it sends the last ACK of the connection establishment phase on F2 and changes

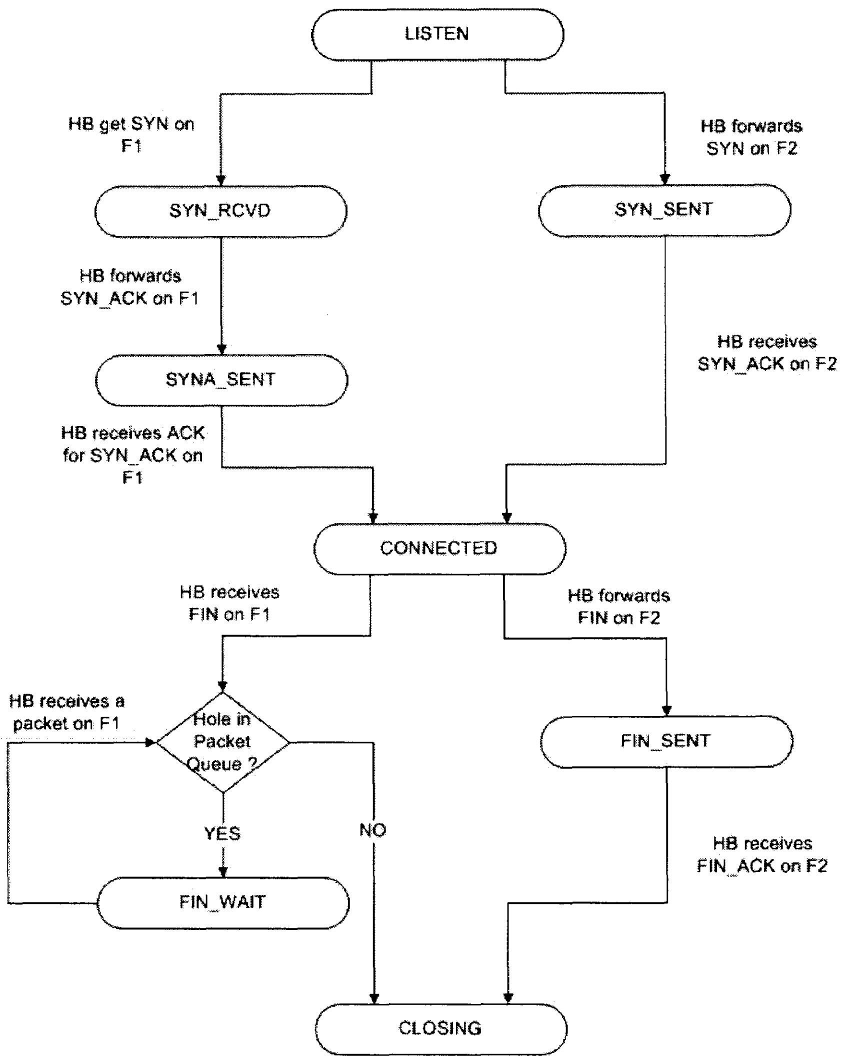

Figure 6.5 Split TCP finite state machine.

the state of F2 to TCP_CONNECTED. After the SYN-ACK packet is forwarded on F1, its state is changed to TCP_SYNA_SENT. The state of F1 is changed to TCP_CONNECTED as soon as the HB receives on F1, the last ACK packet of the connection establishment phase. Once both F1 and F2 are in TCP_CONNECTED, the HB processes and forwards the data packet among the legs. Once the source has sent all the data packets, it will send the FIN packet. On receipt of a FIN packet, the HB checks whether all data packets have been received from the source. If yes, the state of F1 is changed to TCP_CLOSING and the HB sends the FIN-ACK packet back towards the source host. However, if there are missing data packets, the state of F1 is changed to TCP_FIN_WAIT. Once all missing data packets are received, the HB changes the state of F1 to TCP_CLOSING and sends the FIN-ACK packet to the source host. After the HB has forwarded all data packets including the FIN packet on F2, it changes the state of F2 to TCP_FIN_SENT. On receipt of a FIN-ACK packet on F2, the state of F2 is changed to TCP_CLOSING. When both F1 and F2 are in the TCP_CLOSING state, the HB frees all the memory that was used to maintain the end-to-end TCP connection.

6.1.5 Packet Processing

The previous sections discussed about the 4 components that were designed for implementing Split TCP at the HB. This section describes the algorithm implemented at the HB for processing the packets of a TCP flow. The HB processes different types of TCP packets like SYN, SYN-ACK, FIN, FIN-ACK, ACK, RST and data packets. As mentioned earlier, the entire packet processing code of Split TCP is implemented as a LKLM (*ip_in_intercept.c*) which is registered at the NF_IP_PRE_ROUTING hook of the Netfilter subsystem. Thus allowing the LKLM to intercept all incoming network traffic.

The function, `ip_in_hook_filter()`, acts as the entry point into the LKLM. It behaves as the `main()` function of a C program. The main task of this function is to filter TCP packets from the incoming network traffic and decide which of these TCP packets will be processed by the Split TCP code and which ones should be let inside the kernel for regular network processing. The HB maintains a table of the supported IP address pools along with the IP address of the next hop HBs for each direction of flow. This helps in filtering of the relevant TCP traffic. The following tasks are performed, in order, by the function:

1. Since IP over IP is being used for transporting the data packets, the first check performed by the function is to find out whether it received a simple IP packet

or an encapsulated IP packet. The *protocol* field of the IP header is used for this. If the packet received is an encapsulated IP packet, the function checks whether the source IP address in the header belongs to a supported network IP pool and whether the destination IP address in the header corresponds to one of its own IP addresses. If yes, it removes the outer IP header and updates the `data` pointer of the `sk_buff` to point to the start of the inner IP header before processing any further.

   It must be noted that the HB will skip this step and directly go to the next one in case the packet received does not contain an encapsulated IP packet.

2. The HB then consults the table containing the supported IP address pools once again to check if the packet being processing is supported by checking the source and destination IP address from the IP header. If it is, the packet is processed by the LKLM else it is forwarded to the kernel for normal network processing. It also determines the direction of flow of the packet i.e. is the packet flowing from the source host to the destination host or in the opposite direction. This is important while updating the variables of the relevant statekeeper.

3. If the packet belongs to a supported TCP flow, the HB traverses the flow tracker linked list to find the appropriated flow tracker instance. If this search fails for any packet other than the SYN packet, the packet is dropped by the HB and a RST is sent in the direction of the source of the packet.

   However, if the search fails for a SYN packet, which will happen for a non-duplicate SYN packet, the HB creates an instance of the flow tracker and enqueues it.

4. Next the MAC header details are extracted from the incoming packet. This helps the HB bypass the kernel ARP mechanism while sending ACK packets to the source of the data packets.

Any packet filtered for Split TCP processing is next forwarded to the function `process_in_pkt()`. For the filtered packets, the LKLM returns NF_STOLEN to the kernel. For all other packets it returns with NF_ACCEPT.

The `process_in_pkt()` function is the heart of the packet processing code. The function is responsible for identifying the different type of TCP packets and invoking the appropriate packet processing code. Since the kernel module works at the IP layer, the function starts off by populating the TCP header of the `sk_buff`. The function then calls the `tcp_parse_option` to process any TCP options that are present in the

TCP header. Depending on the value of the TCP flags the function processes the packet as follows:

SYN packets: A non duplicate SYN packet is enqueued in the appropriate packet queue without performing any checks. The receipt of the SYN packet is used to initialize the TCP variables of one leg, say Leg 1 (or F1), of the TCP connection at the HB. The HB checks for the MSS, window scaling and ECN option. If set, it updates the respective variables of the statekeeper. Depending on the source of the packet, end host or another HB, a flag is updated to indicate if the packets flowing towards the source need to be encapsulated or not.

Before forwarding the SYN packet on Leg 2 (or F2), the HB first finds the IP address of the next hop HB, if any. This information is used while processing the data packets. The function then calls the `prepare_tcp_syn` function to forward the SYN packet towards the destination host. It also initializes the window parameters for the Sliding Window protocol of TCP. The HB ensures that IP over IP, if it will be used, is reflected in the MSS option of a given direction of data flow.

SYN-ACK packets: At the receipt of a SYN-ACK packet, the HB initializes the variables of Leg 2 of the TCP connection. It checks the TCP options that are being negotiated and updates the respective variables for the MSS, window scaling and ECN option. It also updates a flag to indicate if the packets flowing towards the destination host need to be encapsulated or not.

Before forwarding the SYN-ACK packet, the HB first finds the IP address of the next hop HB, if any. This information is used while processing the data packets. The function then calls the `prepare_tcp_synack` function to forward the SYN-ACK packet towards the source host. It also calls the `prepare_tcp_ack` function to send the last ACK of the connection establishment phase back to the destination host. It also initializes the window parameters for the Sliding Window protocol for Leg 2.

Data and ACK packets: These type of packets are received when the TCP connections of Leg 1 and Leg 2 are in the established state. The data and ACK packets require similar processing and hence the `process_in_pkt` function treats them as a single case.

The receipt of both data and ACK packets is an opportunity to forward unsent data packets sitting in the packet queues. The ACK for a corresponding data packet is always piggybacked on an outgoing data packet, if there is one. The `prepare_fwd_data()` function, which will be described later, is called to process and forward data packets. The `prepare_tcp_ack()` function is called to create and send out the ACK packet if there is no data packet to piggyback it on.

FIN and FIN-ACK packets: These packets indicate the teardown of a TCP connection. However, there is a possibility that a FIN packet contains some data. Hence it is treated as a data packet by the HB. Depending on the TCP state of the leg on which the FIN (FIN-ACK) packet was received, the HB processes it as follows.

If the packet received is the first FIN packet, the state of that leg is changed to TCP_CLOSING provided there are no missing data packets. Else the HB changes the state of the leg to TCP_FIN_WAIT and calls the `prepare_tcp_ack()` function to inform the source host of the missing data packets.

The receipt of a FIN-ACK packet indicates that the next host (HB or destination) has received the FIN packet and is ready to close the connection. The HB changes the state of the leg to TCP_CLOSING from TCP_FIN_SENT. It calls upon the `prepare_tcp_ack()` function to send out the last ACK of the connection. It deletes the RTO timers for both the legs and initiates the TIME_WAIT timer as required by [32].

A task common while processing SYN and SYN-ACK packets is to calculate, if required, the window scale factor for the respective legs. This task is performed by the function `calc_rcv_wscale` as shown in the following code abstract:

```
void calc_rcv_wscale
(int __space, __u32 mss, __u32 window_clamp, int wscale_ok, __u8 *wscale)
{
unsigned int space = (__space < 0 ? 0 : __space);

if(window_clamp == 0)
window_clamp = (65535 << 14);

space = min(window_clamp, space);

(*wscale) = 0;
if(wscale_ok) {
while(space > 65535 && (*wscale) < 14) {
space >>= 1;
(*wscale)++;
}
}
}
```

The `rcv_calc_wscale` function calculates the window scale facotr based on the values of the `__space` and `window_clamp` parameters. `__space` represents the available buffer space. Since this function is called during the connection establishment phase, the entire buffer of 1MB is available. Hence `__space` is always initialized to 1MB. `window_clamp` represents the upper limit for the acceptable advertised window. If no value is passed for it, `window_clamp` is initialized to the theoretically maximum possible value ($65535 \ll 14 = 2^{30}$). See [10] for an explanation of why 14 is the upper limit for the shift count. Based on the value of `__space` and `window_clamp`, the window scale factor is calculated in the `while` loop. This logic was adapted from the function that calculates the window scale factor in the Linux kernel.

A common task while processing all of above types of packet is to update the variables of the Sliding Window protocol *viz.* `snd_next`, `snd_una`, `rcv_next` and `mac_rcv_byte`. This is important as these variables dictate the flow of the data stream in a given direction. This task, among others, is performed by the `process_tcp_ack()` function. This function is called for each of the packet types listed above. Another task common for most of the packets is to queue them in the respective packet queues while at the same time release the data packets that have been acknowledged. These 2 tasks are performed by the `enqueue_packet` and `prepare_fwd_data` functions, respectively. These functions will be described later in this section.

The `process_tcp_ack()` function starts off by finding the direction of flow of the packet being processed. It is then able to retrieve the values of the right variables that were updated by the previous packet in the same direction of flow. The function executes the following steps in order:

1. If the acknowledgment sequence of the packet is less than the `snd_una` variable, the packet is not processed any further. Such a packet corresponds to an out of order ACK packet.

2. The ECN flags are processed next by the function. The 3 ECN flags are combined into one variable as shown below

   ```
   tp->ecn_flags=((tp->ce << 2) | (tp->ece << 1) | (tp->cwr));
   ```

   where `tp` is a pointer to the respective statekeeper. `process_tcp_ack` then calls the function `process_ecn_flags`, which uses the variable `ecn_flags` to determine if the HB needs to reduce the congestion window or ask the previous host (HB or source host) to reduce its congestion window or both or none.

3. If the starting byte of the received packet is less than `rcv_next`, the packet is not queued in the respective packet queue. Since the current implementation cannot

handle packets containing overlapped data bytes, such a packet corresponds to an out of order retransmission of a data packet. Hence the HB drops the packet (frees memory used by the packet) and sends back an ACK packet to help the previous host (HB or source host) to keep up to date with the current status of the data stream.

If the starting byte of the received packet is greater than or equal to `rcv_next`, `process_tcp_ack` calls the `enqueue_packet()` function to queue the packet in the respective packet queue.

4. The function then checks whether the received packet is a duplicate ACK or not. If it is, it increments the duplicate ACK counter by 1. The duplicate ACK counter is used while forwarding data packets to the next host (HB or destination host).

   As mentioned earlier, the NewReno algorithm is implemented for fast retransmission and fast recovery. Hence for every good ACK, if the TCP connection was in fast recovery, the HB checks if the ACK is a partial ACK or one that acknowledges all outstanding data packets. In either case, the duplicate ACK counter is initialized to 0. The congestion window is updated accordingly by calling the `tcp_update_cwnd()` function.

5. The last task performed by the function is to update the state of a TCP connection, if need be. If the TCP connection is in the TCP_SYN_SENT or TCP_SYNA_SENT state, the function releases the first packet from the packet queue. Since neither of the legs is in the TCP_CONNECTED state, the first packet in the packet queues will either be a SYN or a SYN-ACK. The function then updates the state of the connection to TCP_CONNECTED. If the TCP connection is in the TCP_CLOSING state and there are holes in the packet queue (missing data packet), the function updates the state to TCP_FIN_WAIT. On the other hand, if the TCP connection is in the TCP_FIN_WAIT state and has received all missing data packets, it updates the state to TCP_CLOSING.

   Thus, the `process_tcp_ack` function makes sure that a split TCP connection follows the Split TCP state machine by changing its state.

Another task common for all packets, except pure ACK's, is to queue the packet in the right packet queue. The function, `enqueue_packet`, is responsible for this task. Since the HB reorders and forwards the data packets, `enqueue_packet` is designed to not only place the data packet in the right queue, but at the correct position within the queue. As mentioned previously, the packet queues are designed to work as a FIFO queue. Hence, in the best case scenario a data packet is enqueued at the tail of the queue. However if the data packet is about to create a packet hole or fill one, extra processing needs to be done. The `hole_in_queue` and the `pkt_bfr_hole` variables are used by the function to process data packets arriving after a lost data packet. The `hole_in_queue` variable is initialized to 0 indicating that there no holes in the queue and the `pkt_bfr_hole` variable is initialized to NULL.

The `enqueue_packet` function starts off by finding out the sequence number and the amount of data bytes of the last packet in the packet queue. This helps in deciding whether the packet is in sequence or is a missing data packet or is about to create a hole in the packet queue. `enqueue_packet` performs the following steps in order:

1. If there are no packets in the queue and the starting byte of the packet is equal to `rcv_next`, the packet is enqueued without performing any further checks. This scenario occurs when all the previous data packets have been forwarded and acknowledged and the current data packet is in sequence. However, if the starting byte is not what the HB is expecting, the packet is considered to be out of sequence. The function jumps to the code that handles data packets that create holes in the queue (Step 2, Case 2).

   As mentioned previously, the original SYN and SYN-ACK packets are blindly enqueued in their respective packet queues.

2. If there are data packets already present in the queue, the function first reads the value of the `hole_in_queue` variable. Depending on the value of this variable, there are 2 possible cases that the function handles:

   Case 1: There are no holes in the queue i.e. `hole_in_queue = 0`

   This case implies that either all the data packet received so far have been in sequence or the previous out of sequence data packets have been taken care of. Hence the new data packet could either be in sequence or it might create a hole in the queue. The packet will create a hole in the queue if the following condition is true.

$$\begin{aligned} Startingbyteofpacket &> Previouspacketsequence \\ &+ Previouspacketdatabytes \end{aligned}$$

   If the condition is true, the variables `hole_in_queue`, `pkt_bfr_hole` and `max_rcv_byte` are updated. The `hole_in_queue` variable is set to 1 to indicate the presence of a hole. `pkt_bfr_hole` is made to point to the packet currently at the tail of the queue. And `max_rcv_byte` is set to the last data byte of the packet being processed. The packet is then enqueued in the packet queue. Figure 6.6 illustrates this scenario.

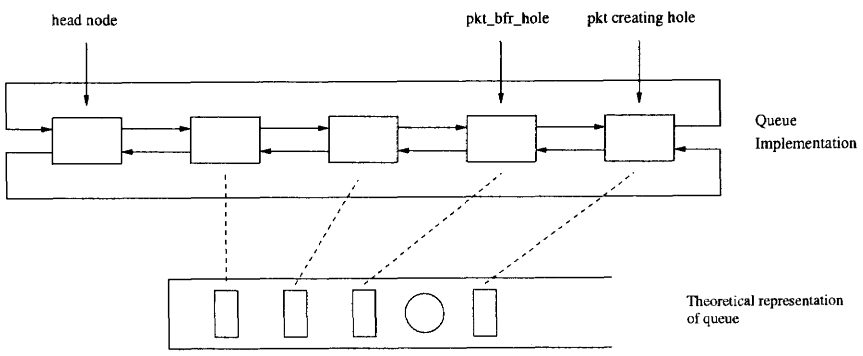

Figure 6.6 Representation of a packet queue containing packet holes.

Case 2: There is a hole(s) in the queue i.e. `hole_in_queue = 1`

Handling of a data packet when there are holes in the packet queue can be tricky. A point worth noting here is that in this scenario there is at least one hole in the packet queue. Also, the variable `pkt_bfr_hole` always points to the packet just before the first hole, in sequence, within the packet queue. There following 3 sub-cases are consider.

In the simplest of case, the starting byte of the current packet is equal to `max_rcv_byte + 1`. The packet is in sequence and hence is queued at the tail of the packet queue. The `max_rcv_byte` variable is updated to reflect this.

It is possible that the current data packet is the missing data packet that will advance `rcv_next`. In this case, the data packet is inserted just after the packet pointed to by `pkt_bfr_hole`. Since there might be additional packet holes, `enqueue_packet` calls the `find_new_hole_update` function to find the next hole in sequence and update the `pkt_bfr_hole` and `rcv_next` variables accordingly.

As a final case, it is possible that the current packet fills another hole within the packet queue. It must be recalled that the `pkt_bfr_hole` variable of only the head node of a packet queue contains a valid pointer. Hence at any given time, the Split TCP code can find the packet before the first hole in one operation. However, if there are additional holes, like in this sub-case, the code will traverse the packet queue starting at the packet pointed to by `pkt_bfr_hole`. The `find_hole_and_enqueue` is called by `enqueue_packet` to perform this task and insert the new data packet at its correct position. No variables are updated for this case.

Figure 6.7 illustrates the 3 possibilities discussed above.

After the packet has been enqueued, the next step for the HB is to forward it towards the destination host. The `prepare_fwd_data` function is responsible for this task. As mentioned earlier in this section, `prepare_fwd_data` is called on receipt of a data packet as well as at the receipt of a pure ACK packet. Hence there are two tasks

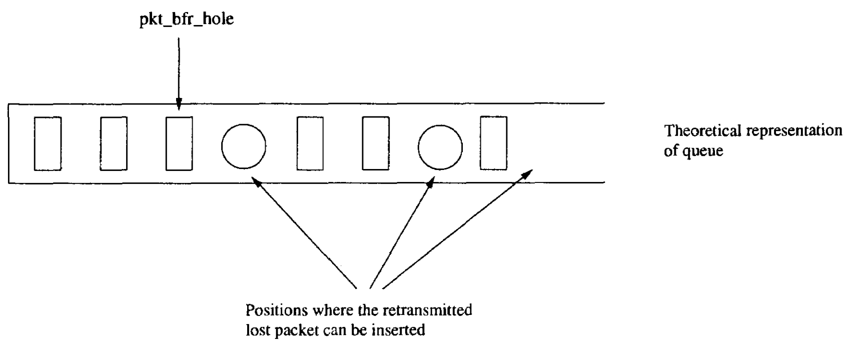
pkt_bfr_hole
Theoretical representation of queue
Positions where the retransmitted lost packet can be inserted

Figure 6.7 Possible positions to insert the data packet in a packet queue containing holes.

performed by the function - do ACK related processing for the current packet and do the processing related to forwarding a data packet. `prepare_fwd_data` performs the following steps in order:

1. The `prepare_fwd_data` function starts by releasing all data packets that the packet currently being processed acknowledges. As long as there are packets in the queue, the function will always pick the packet at the head of the queue to check if it is being acknowledged. If yes, the packet is released. This process is repeated for all data packets that have been forwarded or until the acknowledgment sequence number of the packet being processed is less than the starting sequence number of the packet currently at the head of the queue. The buffer space and congestion window are updated accordingly.

   For the packet whose sequence number is equal to the ACK sequence number of the packet being processed, the RTT estimation and RTO update logic is called upon. However, the ACKs for data packets that were sent during the congestion avoidance phase are not considered for the RTT estimation and RTO update logic.

   Once all acknowledged data packets have been released, the `snd_una` variable is updated.

2. The function then checks if any data packets can be forwarded towards the destination host. It finds the minimum of the congestion window and the advertised window and compares this value with the current number of packets in flight. This value is expressed in packets as opposed to bytes.

3. The function then checks whether there is a need to retransmit a data packet. The `prepare_fwd_retrans_data` function is responsible for retransmitting lost data packets. The function is called from `prepare_fwd_data` for the following 3 scenarios - the duplicate ACK count is equal to 3 and the TCP connection is not in fast recovery or the TCP connection is in fast recovery but the packet

corresponds to a partial ACK or the ECE flag in the IP header is set. While dealing with partial ACK's, the congestion window is updated.

The `prepare_fwd_retrans_data` is called when the RTO timer expires. This functionality is implemented using the `timer_list` structure of the kernel.

4. The function is now ready to forward data packets, if allowed. It finds the first non-sent in sequence data packet that lies before a packet hole. If the data packet is being forwarded to another HB, the function encapsulates the original packet in an IP packet. The encapsulated data packet replaces the original packet that was queued in the packet queue. This saves processing time if the packet needs to be retransmitted. For all data packets being forwarded, the ACK sequence value is replaced by the current `rcv_next` value for the TCP flow.

The HB treats a FIN packet like a data packet. Once received, it is enqueued in the appropriate packet queue and forwarded once all the data packets in front of it have been forwarded.

The HB also contains functions for generating various types of TCP packets. The functions are named as `prepare_tcp_x` where "x" corresponds to the type of packet and could be either syn, synack, ack, finack, probe or reset. These functions start off by creating a new `sk_buff`. Once the TCP, IP and MAC headers have been populated, the HB forwards the packet to the kernel function `dev_queue_xmit` for putting it on the wire.

CHAPTER 7

EXPERIMENTS AND RESULTS

The Split TCP implementation in the HB was tested against two modes of operation - bulk transfer and real time user response. Telnet and SSH were used to conduct real time response test while files of varying sizes were transferred for the bulk transfer mode. Each experiment was conducted twice, one with the Split TCP mechanism enabled in the HB and one without. The findings for the Telnet and SSH experiments are discussed towards the end of this chapter.

The implementation of Split TCP is done on desktop computers consisting of either a Pentium 4, 2.4 GHz processor or an AMD Athlon XP 1700+/2000+ processor. The Linux kernel version on these computers varied from 2.4.20 to 2.6.15. These were the HB's. All computers had 512 MB of RAM. 100 Mbps Ethernet links were used for communication.

The implementation was tested under various scenarios by varying the network parameters, HB capabilities or the location of the HB in the network. The network parameters of RTT and drop probability were implemented using NistNet [34]. The individual experiments are discussed below.

7.1 1 HB Scenario

The first set of experiments were performed with 1 HB between the source and destination host. Figure 7.1 shows the setup used for these experiments.

For the first experiment, the network parameters were chosen so as to reflect a HB that is placed at the worst location within the network. Measurements were taken with $RTT_2 \gg RTT_1$ and $p_2 > p_1$. For this scenario, Section 2.2 predicts that Split TCP will do only marginally better than the end-to-end TCP connection. This is indeed what was found.

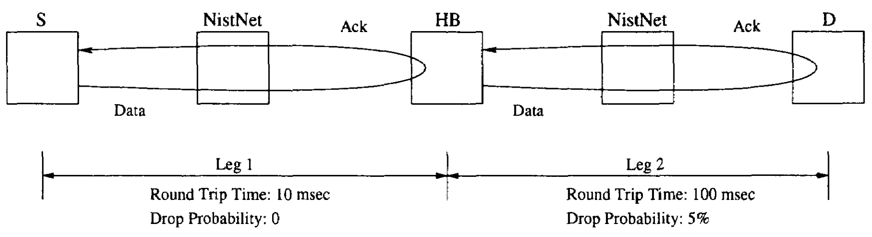

Figure 7.1 1 HB network setup.

The network parameters chosen for this experiment are as follows: $RTT_1 = 10ms$, $p_1 = 0\%$, $RTT_2 = 100ms$ and $p_2 = 5\%$. Various files were transferred between the source and destination host to take measurements. The file size varied from 25 MB to 450 MB with increments of 25 MB. Each file was transferred 3-5 times to get a good approximation of the transfer time. Figure 7.2 compares the average transfer times of the files with Split TCP and with classical TCP. In the graph, the X-axis represents the size of the file being transferred, while the Y-axis represents the average transfer time in minutes. Though Split TCP does better than classical TCP, the factor of improvement is rather low and ranges between 1.26 and 1.54.

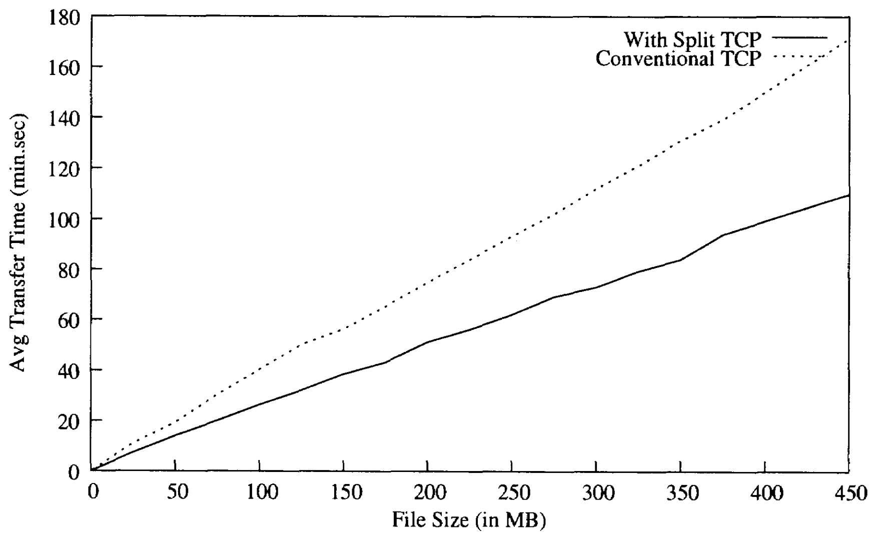

Figure 7.2 Measurement for $RTT_2 \gg RTT_1$ and $p_1 < p_2$.

For the second experiment, only minor modifications were made to the setup for the first experiment. All network parameters except p_1 were kept the same. p_1 was set to 5%, equal to p_2. The same experiment was performed. Figure 7.3 compares the transfer times of the files under these conditions. Once again, Split TCP does marginally better than classical TCP. The factor of improvement ranged from 1.16 and 2.55.

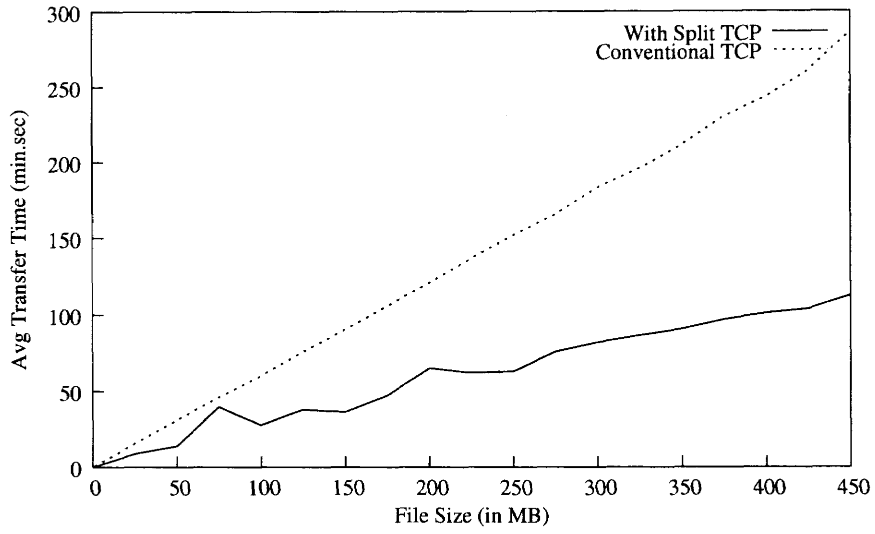

Figure 7.3 Measurement for $RTT_2 \gg RTT_1$ and $p_1 = p_2$.

7.2 Varying Drop Probability Scenario

For the previous two experiments, all the network parameters were kept constant during the course of the experiment. In this experiment, the drop probability of the leg near the destination was varied. The experiment was conducted to find the drop probability at which the performance of Split TCP would start to degrade.

The setup of Figure 7.1 was used for the experiment. The network parameters were set to the following values: $RTT_1 = 200ms$, $p_1 = 0\%$ and $RTT_2 = 10ms$. The value of p_2 was varied from 5% to 20% in increments of 1. This resembles the situation involving 1 HB placed near the earthstation of the developing country which is considered to be a "good HB location". A 100 MB file was transferred between the end hosts.

Figure 7.4 compares the average transfer time of the 100 MB file with Split TCP and with classical TCP. As seen from the graph, a flow with Split TCP does much better and the factor of improvement ranged from 6.18 to 12.70.

The three experiments just described, prove the hypothesis stated in Section 2.2 i.e. the factor of improvement is largest when the network problems are localized one per leg. To state it explicitly, in the first experiment, the factor of improvement for a 100 MB file transfer was 1.48. This is very small when compared to 12.54, the factor of improvement observed in this experiment for the same file and network parameters ($p_2 = 5\%$).

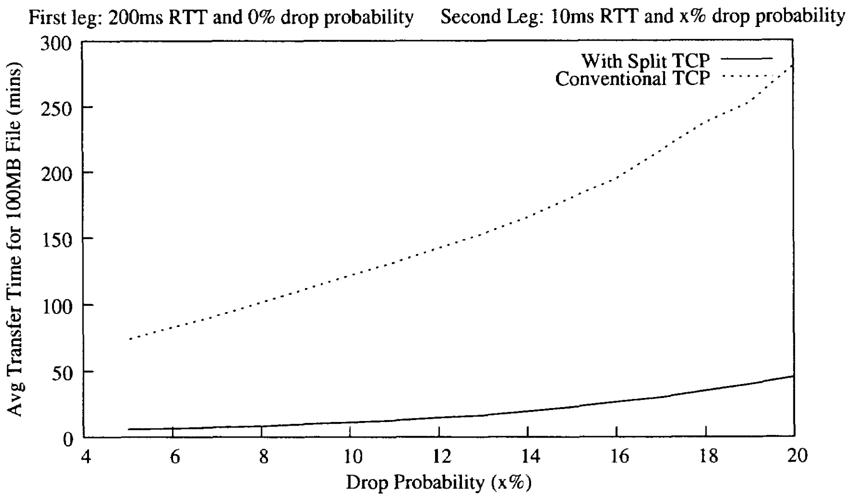

Figure 7.4 Measurements for varying drop probability in 1 HB setup.

7.3 3 HB Scenario

This scenario reflects the campus scenario discussed in Section 2.3. The network setup of Figure 7.5 was used for conducting this experiment. The drop probability between HB_I and HB_B is varied from 5% to 15% in increments of 1. A 100 MB file was transferred between the end hosts.

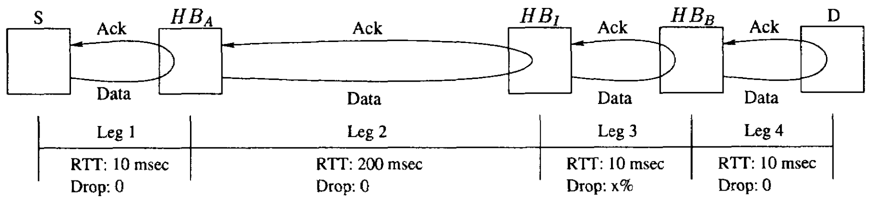

Figure 7.5 3 HB network setup.

Figure 7.6 reports the measurements obtained for this experiment. It compares the average transfer time of a 100 MB file for Split TCP and classical TCP. The factor of improvement increased with increasing drop probability and ranged from 2.60 to 6.54. One can expect larger difference at higher drop probabilities.

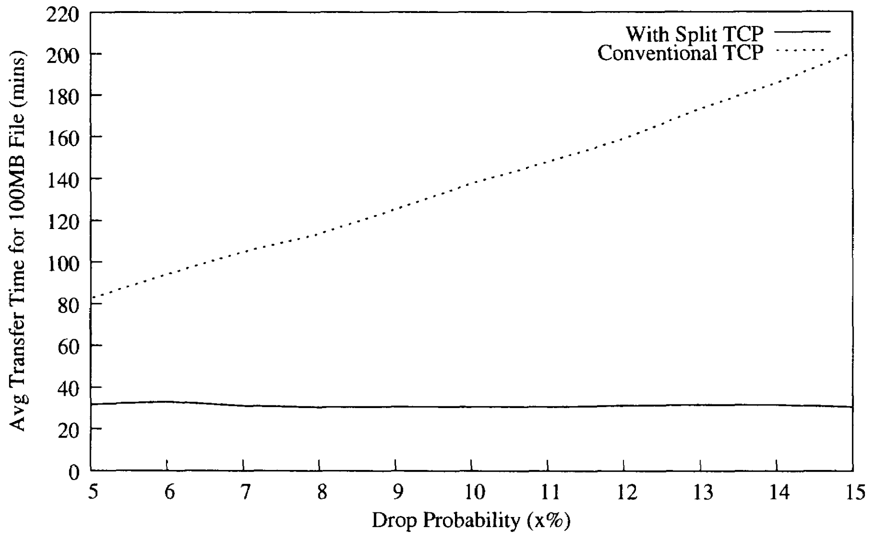

Figure 7.6 Measurements for varying drop probability in 3 HB setup.

An interesting observation is the flat line characteristic of Split TCP with increasing drop probability. This implies that a TCP connection with Split TCP is almost independent of the drop probability. It also means that there is a good possibility that the new bottleneck is either the advertised window of one of the legs or the linkspeed of one of the legs. A study of the tcpdump [35] outputs showed that the bottleneck was the advertised window (without window scaling: at most 65535 bytes) between HB_A and HB_I (i.e. the window advertised by HB_I to HB_A).

This indicates that use of the window scaling option between the HBs will improve performance of Split TCP. The next experiment confirms this analysis.

7.4 3 HB Scenario With Window Scaling

In all the previous experiments, the window scaling option was not used. This limits the effective window to 65535 bytes instead of the higher congestion window. For this experiment, the window scaling option was used in the HB. Note that the end hosts need not understand the window scaling option. See Chapter 5 for an explanation of how the window scale factor was computed. The network setup of Figure 7.5 is used for this experiment with the drop probability of "Leg 3" fixed at 5%. Various files were transferred between the source and destination host to obtain the measurements. The file size varied from 25 MB to 250 MB with increments of 25 MB. Each file was transferred 3-5 times to get a good approximation of the transfer time.

Each file was transferred under 2 different conditions - One with Split TCP enabled in the HB and window scaling being used only between the HBs, and one with Split TCP disabled in the HB and window scaling being used by the end hosts. Figure 7.7 compares the average transfer times of the files under these 2 conditions. As seen from the figure, a flow with Split TCP and window scaling option performs better as compared to a corresponding regular TCP flow with end hosts using window scaling. The factor of improvement ranged from 2.20 to 5.26.

When compared with the previous experiment, under the same network conditions (drop probability of 5%), the average transfer time of a 100 MB file with Split TCP and window scaling improved by a factor of 2.

The previous experiment was conducted for a fixed drop probability of 5% in "Leg 3". However, to get a clear understanding of the improvement due to window scaling, the drop probability of "Leg 3" must be varied, as was done for the experiment of Section 7.3. Thus for this experiment, the network setup of Figure 7.5 is used with

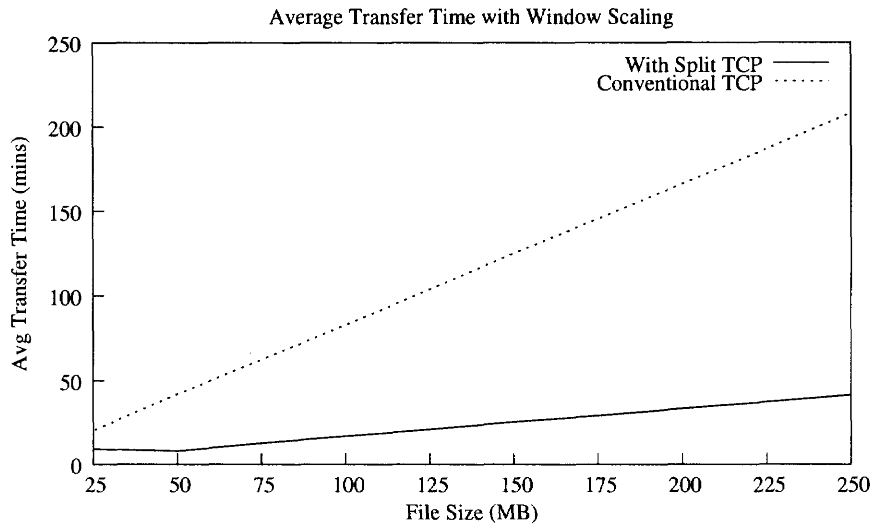

Figure 7.7 Measurements with window scaling in 3 HB setup with fixed drop probability of 5%.

the drop probability of "Leg 3" varying from 5% to 15%. For each drop probability, a 100 MB file was transferred 3-5 times with and without Split TCP. For the end-to-end TCP connection, window scaling was turned on in both the end hosts. Whereas for Split TCP, though the HB's used the window scaling option among themselves, it was turned off at the end hosts.

Figure 7.8 compares the average transfer time of the 100 MB file over various drop probabilities. As is seen from the figure, the TCP flow with Split TCP once again out performs regular TCP with the factor of improvement ranging from 4.90 to 12.44. This factor of improvement is twice the factor of improvement that was obtained for the experiment of Section 7.3.

The use of window scaling option by the end hosts, as anticipated, did not change the transfer time of the 100 MB file when classical TCP was used. Hence from Figure 7.6 and 7.8 it can be inferred that as the network conditions worsen (increasing drop probabilities), the factor of improvement because of Split TCP will increase and the use of window scaling with make this factor even larger.

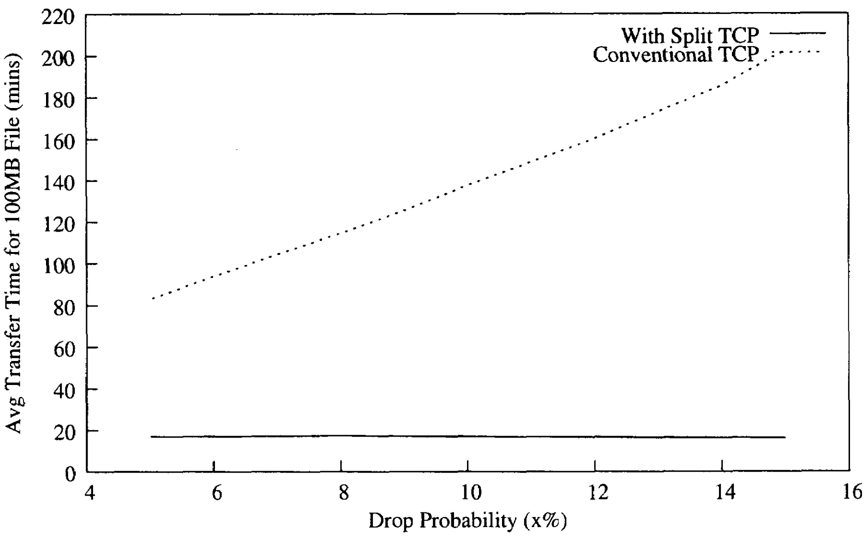

Figure 7.8 Measurements with window scaling in 3 HB setup with varying drop probability.

Once again, it is interesting to note the flat line characteristic of Split TCP with increasing drop probability. This implies that Split TCP is independent of drop probability and that the bottleneck is either the advertised window of one of the legs or the linkspeed. Looking at the tcpdumps for the experiment, it was confirmed that the advertised window between the source and HB_A was the bottleneck. Since the window scaling option was not negotiated between the source host and HB_A, the advertised window of this leg could not grow beyond 65535.

7.5 Wireless Network Scenario

This section reports on measurements taken over an actual heterogeneous network. Figure 7.9 shows the network setup used for this experiment.

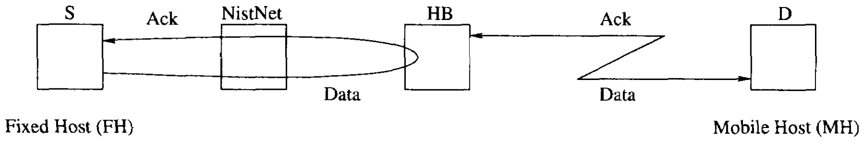

Figure 7.9 Wireless network setup with 1 HB.

As shown in the figure, the HB acts as the base station where the wired and wireless leg of the end-to-end connection meet. This allows the HB to deal with the network problems of the wired leg independent of the network problems of the wireless link and vice versa. In the setup shown, the HB and the destination host (Mobile Host (MH)) communicate using the IEEE 802.11b protocol and is the only wireless hop in this experiment. The 802.11 protocols use CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) as the basic access mechanism. In CSMA/CA each host needs to check if the channel is idle before it can start transmission. In case of contention, an exponential random backoff mechanism is used, at the expiration of which the host tries to transmit again. The standard also defines MAC layer acknowledgment, retransmissions and fragmentation which work in addition to the corresponding mechanisms at the transport layer. For some wireless NIC's, these MAC layer mechanisms can be enabled, disabled or configured at will.

The destination host was a laptop running Linux, kernel version 2.6.15. It had an Intel(R)/Wireless LAN PCI Adapter for connectivity. At the HB, an off the shelf Netgear Wireless NIC having an Atheros chipset was used for communication on the wireless leg. The HB and the laptop were connected in ad hoc mode at a data rate of 11 Mbps. For the purpose of the experiment, the laptop was moved within and to different floors of the GITC building at NJIT. For the leg between the source host (Fixed Host (FH)) and the HB, the RTT and the drop probability were set to 200ms and 0% respectively. Since the experiments were taken on an actual heterogeneous network, the RTT and the drop probability of the wireless link were not under control and varied drastically. For most of the runs of the following two experiments, wireless signal loss and TCP disconnects were experienced, both of which added to the drop probability of the wireless link.

In this dissertation, the results of experiment with and without MAC retransmissions are reported.

In the first experiment the MAC retransmission were left enabled. A 50 MB file was transferred between the FH and the MH. The file was transferred 3 times to get a good approximation of the transfer time. Figure 7.10 compares the average transfer time of the file with Split TCP and with classical TCP.

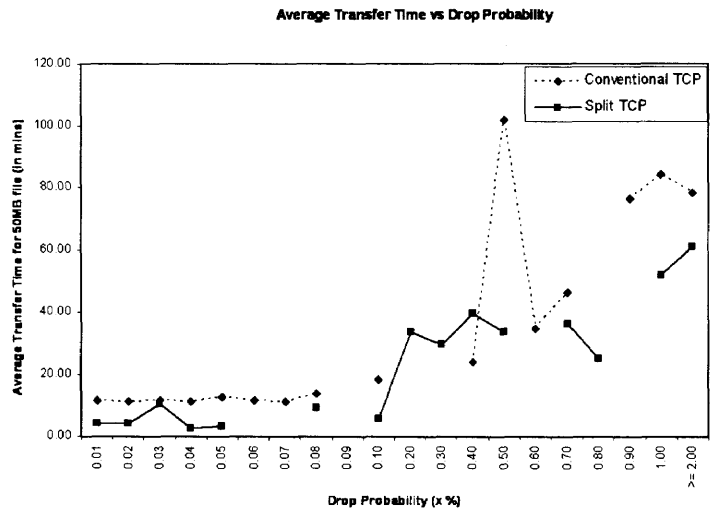

Figure 7.10 Measurements for heterogeneous network setup with MAC retransmissions.

In Figure 7.10, the y-axis represents the average transfer time while the x-axis represents the drop probability % that was measured from the tcpdump outputs of the experiment. As seen from the figure, the maximum drop probability achieved was less that 3%. Also not for all measured drop probabilities, are there comparable entries for Split TCP and classical TCP. However, for most entries in the figure, Split TCP does comparably better than classical TCP. It is also interesting to note that as the drop probability increases and gets close to the ones that were used for experiments with wired legs, the factor of improvement because of Split TCP starts to increase.

In the second experiment, MAC retransmissions were disabled. A 100 MB file was transferred between the FH and the MH. The file was transferred 3 times to get a good approximation of the transfer time. Figure 7.11 compares the average transfer time of the file with Split TCP and with classical TCP.

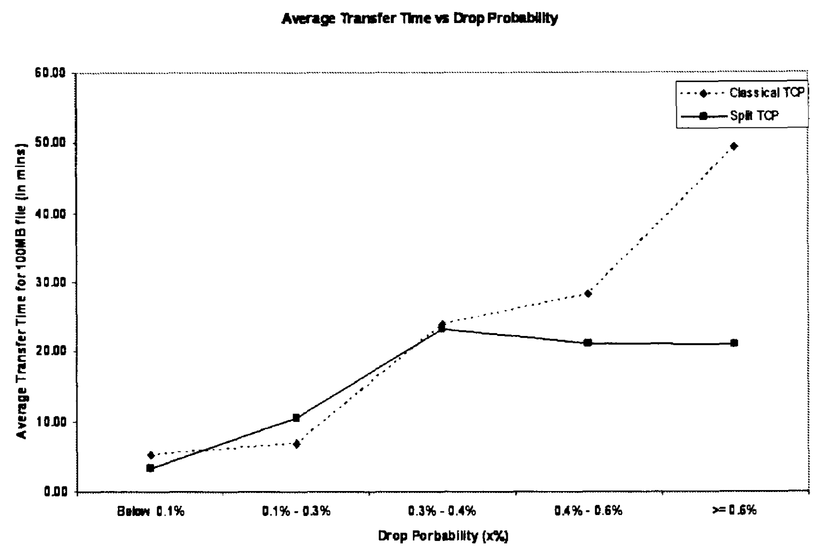

Figure 7.11 Measurements for heterogeneous network setup without MAC retransmissions.

In Figure 7.11, the y-axis represents the average transfer time and the x-axis represents the drop probability % measured from the tcpdump outputs of the file transfers. As seen from the figure, the maximum drop probability achieved for this experiment was less that 1%. Once again it is interesting to see that although Split TCP does not do well for lower drop probabilities, the factor of improvement starts to increase for larger drop probabilities.

It should be noted that the drop probabilities for the experiments over an heterogeneous environment were observed to be much smaller than the drop probabilities used for the experiments over a wired environment.

7.6 Interactive Connections

This section discusses the findings for an interactive connection, like Telnet or SSH, with Split TCP. This is unlike the previous sections which discussed the findings for performance improvement with Split TCP by transferring files of varying sizes between the end hosts.

The successful SSH connections over Split TCP demonstrates that an interactive secure channel is not broken because of the introduction of the HB in the network. Any type of authentication required by SSH is done end-to-end. The results presented in this section are of subjective qualitative nature.

The network setup of Figure 7.1, is used for this experiment with the following network parameters: $RTT_1 = 10ms$, $p_1 = 0\%$, $RTT_2 = 200ms$ and $p_2 = 10\%$. For these settings, minimal to no improvement was perceived with Split TCP in place. However, when the drop probability, p_2, was increased to 20%, a considerable improvement in the response time of the connection was perceived.

7.7 Scalability

In the previous experiments, at any given time, the HBs were managing 1 end-to-end TCP connection. Although they have shown to perform well in the previous scenarios, their performance while handling multiple end-to-end TCP connections was unknown. The first experiment of this section addresses this matter.

For this experiment the network setup of Figure 7.1 was used. For each concurrent TCP connection, a 100 MB file was transferred between the source and destination host. The file was transferred 3 times each to get a good approximation of the transfer time. Figure 7.12 compares the average transfer time of Split TCP with classical TCP for increasing number of TCP connections through a HB.

In Fig. 7.12, the x-axis represents the number of TCP flows handled by the HB and the y-axis represents the average transfer time of a 100 MB file. As is seen from the x-axis, the number of flows through the HB is increased from 1 to 10. It is interesting to note that while managing 10 flows, the HB performs as if it

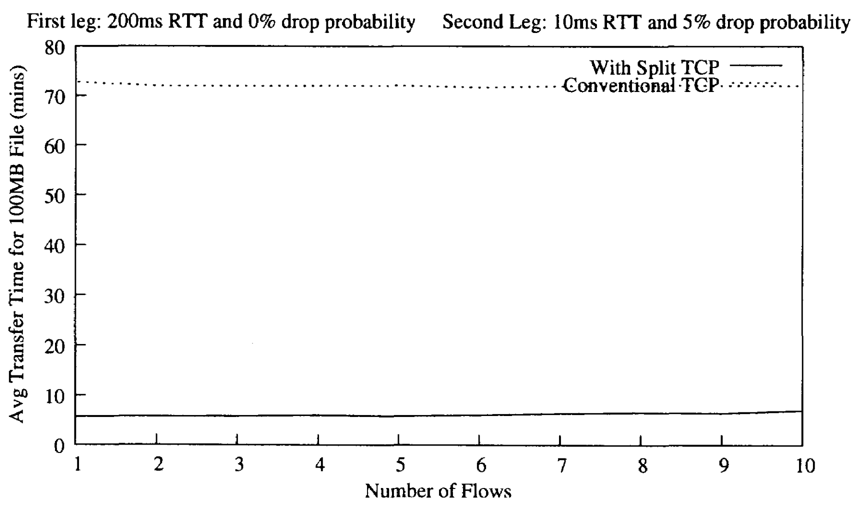

Figure 7.12 HB performance for multiple TCP flows.

was managing 1 end-to-end TCP flow. This experiment shows that with increasing number of TCP connections, the HB scales well.

CHAPTER 8

CONCULSIONS

8.1 Main Contribution

This dissertation aims at improving the performance of TCP connections under unfavorable network conditions of large RTT and high loss probability. To improve TCP performance under such network environments, the dissertation proposes the use of Split TCP where proxies (called Helper Boxes) are introduced in the network to split the end-to-end TCP connection into multiple TCP connections. The dissertation provides a mathematical guarantee for the improvement in TCP performance with Split TCP. Using the mathematical result, it is deduced that the optimal location of the HBs along a network path is such that the network problems get localized one per leg. This guarantees maximum achievable improvement in TCP performance with Split TCP.

The proposed solution, Split TCP, has been implemented in the Linux kernel as a Linux Kernel Loadable Module (LKLM) using the Netfilter system. This kernel level implementation is a first of its kind. The module works at the IP layer and provides both IP and TCP functionalities. The dissertation also proposes the use of IP over IP for exchanging data packets between HBs. Various components were designed which collectively maintain the state of the end-to-end TCP connection at the HB.

The implementation was tested on an actual network setup. The mathematical deduction about the optimal position of the HBs was verified through experiments. The Split TCP mechanism was tested on both wired and heterogeneous network environments. For all experiments, Split TCP was found to improve the end-to-end

TCP performance with the factor of improvement increasing for worsening network conditions.

8.2 Lessons Learned

Many challenging problems of varying degree of difficulty were faced and overcome during the course of this work. This section lists some of the interesting ones.

Since a kernel level implementation of Split TCP was opted for, it became necessary to find out how the kernel handles incoming and outgoing packets. At the time when this project was started, there was (and still is quite true) very minimal documentation regarding the network stack of the Linux kernel. None of them covers the entire network stack and most of them gives a birds-eye description of specific parts of the network stack. The only way to overcome this problem was to read the source code of the network stack of the kernel. This effort has been documented in Chapter 4 of this dissertation.

The implementation of Split TCP was started on kernel version 2.4.x. As the newer stable kernel version 2.6.x emerged, the Split TCP code was migrated onto it. The network stack of 2.6.x is different in many ways when compared with the one in 2.4.x. This resulted in various issues while compiling and executing the Split TCP code on a 2.6.x kernel. While the compilation errors were easily taken care of, the most challenging errors were the kernel panics that occurred while testing the Split TCP module. A kernel panic or a kernel oops, as called in Linux, is an error message dumped by the operating system when it encounters an error from which the operating system cannot recover. An oops message normally consists of the contents of the register and the function call trace information. Since for most oops the system freezes, a serial console was used to capture these kernel panics. Using the offset information displayed in the oops for the faulting function, the faulty instruction can be identified in the object file of the concerned C file. `objdump` [36] was used to create the object file of the respective C file.

8.3 Dissertation Summary

The network conditions of large RTT and high drop probability are known to affect the TCP performance drastically. In Chapter 1 there is an explanation why. These network conditions are present in various network scenarios like, long range TCP connections, wireless environment, interplanetary network, etc. This dissertation proposes Split TCP as the solution for such network environments. Split TCP works by breaking the end-to-end TCP connection into multiple independent TCP connections, each of which is called a leg. This is achieved by introducing proxies, called Helper Box (HB), in the network path. The Split TCP code will reside in these HBs. The goal is to isolate the network problems one per leg, thus leading to improved performance. Chapter 2 provides a mathematical proof for the same. The result of this proof can be used to find out the optimal location of the HB within the network path. Chapter 2 also describes the design environment that is of primary interest. This environment consists of two campuses connected through either a transoceanic link or a satellite link and where one of the campuses is in a third world country.

Split TCP is a fairly known technique and some research has been conducted to evaluate its benefit for mobile and cellular networks. Chapter 3 discusses some of the related research work along with their pros and cons. It also compares Split TCP against other similar techniques, specifically the "Cache and Forward" techniques.

For the purpose of this dissertation, Split TCP was implemented in the Linux kernel by making use of the Netfilter system. The design allows the HB to act as a proxy for the destination host while communicating with the source host and as a proxy for the source host while communicating with the destination host. This is achieved through acknowledgment spoofing and caching of relevant data packets.

The design also allows the use of IP over IP for exchanging data packets between the HBs. This guarantees the flow of data packets through the same sequence of HBs. This is also discussed in Chapter 2. The proposed design makes the HB completely transparent to the end hosts. Hence no code modifications whatsoever are required at the end hosts. The design also supports interactive applications.

A kernel level design is chosen since it allows Split TCP to work with packets as opposed to byte stream. It also preserves the TCP flags at no extra cost. The only disadvantage of a kernel design is the memory constraint, which as seen from the results of Chapter 6, is not a drastic one. Split TCP is designed as mild variant of the "Cache and Forward" technique. Instead of caching the entire file before forwarding, Split TCP only caches those packets for which it has not received an acknowledgment from the destination host.

For the proper functioning of the HB, 4 components were designed: Flow Tracker, Statekeeper, Packet Queues and TCP State Machine. For each TCP flow that the HB splits, it creates an instance of the flow tracker component. The flow tracker node is used to maintain the TCP semantics of the split connection. It accomplishes this by creating instances of the statekeeper and the packet queues. For each end-to-end TCP connection, a pair of statekeeper and packet queue are created, one per leg. The statekeeper is designed to maintain various variables which represent the state of the TCP connection for a given leg. Whereas the packet queues are used for caching data packets that will be forwarded to the respective destination host. The TCP state machine is designed for the proper functioning of the Split TCP engine. It consists of 8 states and is closely modeled to the classical TCP state machine.

Chapter 5 discusses the details of the actual implementation of Split TCP. Split TCP is implemented as 3 C files - two of which are compiled in the kernel source tree and the third one is the kernel module. The 2 files compiled in the kernel contain supporting functions (initialization, assessor and mutator functions, etc.) and variable and structure declarations. The kernel module contains the core packet processing code which provides TCP functionalities at IP layer. Various TCP features have been implemented in the Split TCP code. These include, sliding window protocol, sending acknowledgments back to the source host, RTT estimation, RTO calculation, MSS option, window scaling option, ECN, etc to name a few.

The Split TCP implementation and its features were tested in a LAN environment with the two network conditions of large RTT and high loss probability being introduced through NistNet. Files of varying sizes (in MB) were transferred between the source host and the destination host. The transfer time was recorded as a measure of performance. In the 1 HB scenario with varying drop probability the maximum factor of improvement was 12.70. For a 3 HB scenario (this scenario reflects the network environment of primary interest) with varying drop probability and without window scaling, the maximum factor of improvement achieved was 6.54. For the same setup with window scaling enabled between the HBs, the maximum factor of improvement jumped to 12.44. It was interesting to note that, in both experiments, the transfer time with Split TCP was almost independent of the drop probability. The implementation was also tested in a heterogeneous environment with the leg between the destination host and the HB being the wireless leg. With MAC retransmissions disabled at the HB, the maximum factor of improvement with Split TCP was 2.37. Analyzing the results for the heterogeneous setup was challenging since there was no control over the drop probability. The maximum drop probability that was achieved was only 3%. The Split TCP module was monitored and found to introduce less than 10% overhead on the Linux system with respect to CPU and cache usage.

8.4 Possible Enhancements

The current design and implementation of Split TCP duplicates, with required modifications, various TCP mechanisms. However, the current implementation does not support all end host TCP mechanisms. In order to make the Split TCP implementation complete, one can add the following TCP/IP features:

- The current implementation frees the HB from doing IP fragmentation by manipulating the MSS option during the SYN exchange. However the intermediate routers may fragment the data packet which the HB cannot handle currently.

  This is because of the lack of TCP headers in all the fragments except the first. The current implementation assumes that all data packets picked for processing have all the protocol headers up to the transport layer.

  One way of handling the fragmented packets is to maintain a fragment queue at the HB, one per direction of data flow. All incoming fragments will be placed in this queue. Once all the fragments that make up a data packet have been received, the HB can create a new `sk_buff` that replicates the original data packet. This data packet can then be inserted at its rightful position in the respective packet queue.

- In the current implementation, the HB sends an ACK packet for each data packet received. In other words, delayed acknowledgments have not been implemented.

  Delayed acknowledgments help reduce the ACK traffic considerably for unidirectional data flow. When data flow is bidirectional, the acknowledgment is always piggybacked on the outgoing data packet.

- The current implementation supports the window scaling and ECN TCP options. Another TCP option that might be of interest is SACK.

  The SACK option helps the destination host clearly convey to the source host which packets have been lost. This is particularly helpful for connections with lossy links. Since the network scenario of interest for this dissertation does contain a lossy leg, it would be interesting to see if the use of SACK in that leg improves the end-to-end throughput.

In addition to the above mention TCP/IP features, the following features may also be designed and implemented for completeness:

- **Load Balancing**: The real world scenario shown in Figure 2.4 contains just one intermediate HB between the campuses. For large number of Split TCP flows, it might become the bottleneck. A logical expansion of this scenario is to have multiple intermediate HBs between the two campuses. For each new TCP connection, the HBs within the campuses can then select the intermediate HB with the least load at that moment. This will prevent any one intermediate HB from being the bottleneck node.

- **Packet Marking**: As mentioned previously, the current implementation of Split TCP has support for the ECN mechanism i.e. a packet containing ECN flags will be processed appropriately at the IP and TCP layers of the HB. However, it was found through experiments that the packet marking mechanism ((**RED!**) algorithm) in the Linux kernel is faulty. Settings for which the RED algorithm should have marked at least a few packets, none were actually marked.

  Thus a possible enhancement is to design and implement a new packet marking technique. It can then be loaded either dynamically or statically as a qdisc discipline for the desired interface. Once the technique has been proven to be robust, it may also be merged as patch to the main Linux kernel.

8.5 Future Work

This dissertation proposes Split TCP as a solution for network environments having large RTT and high probability of packet loss. However there are many other network environments that will benefit from Split TCP. Some of these have been researched (Mobile and Cellular networks) while some are still open for discussion. This section presents the possible avenues of future research with Split TCP as a solution.

- **Wireless Environment**:

  Some research has been conducted outlining the advantages of Split TCP in a wireless environment. Some of this has been through simulations while others required restructuring the underlying infrastructure. The implementation presented in this dissertation was tested in a heterogeneous environment containing a single hop wireless leg. A natural extension is to have multiple wireless hops in the setup. This will allow the use of HBs within the wireless part of the connection.

- **Interplanetary Network**:

  Interplanetary network is a new project under taken by NASA with the vision of exchanging information between earth and terrestrial objects or among terrestrial objects. The network environment of the interplanetary network has both the large RTT and high probability of packet loss. Hence, Split TCP is a natural solution. However, the network environment makes it very difficult to localize the network problems one per leg. Although there is a mathematical guarantee of improvement in end-to-end performance with Split TCP, it would be interesting to find out the actual factor of improvement through experiments. It will also be interesting to compare the performance of the current design of Split TCP with a conventional "Cache and Forward" technique for such scenarios.

- **Reliable Multicast**:

  With sharable application getting popular in workplace, various issues concerning IP multicast have come into light. One of them is how to avoid the ACK implosion problem while at the same time provide reliable multicast.

  The ACK implosion problem is quite prominent in a one-to-many multicast scenario as shown in Figure 8.1. In the setup shown, the source host, S, broadcasts a data packet to all the members, receiver hosts R, in the group. Each member will then send an acknowledgment, for the data packet received, back to the source. As the number of members increases, the number of ACK packets destined for the source host increases, thus overwhelming it. This is known as the ACK implosion problem. Various method have been proposed to overcome this problem including the use of negative acknowledgments, hierarchical delivery tree, etc.

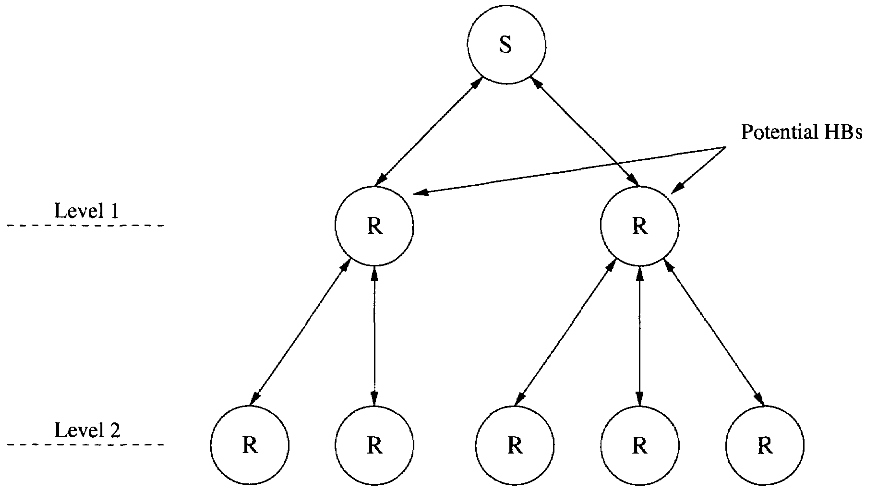

Figure 8.1 Reliable multicast hierarchical delivery tree.

  Considering a hierarchical delivery tree structure as shown in Figure 8.1, Split TCP can be used to minimize the ACK implosion problem while providing reliable multicast. The receivers at "Level 1" can be converted into HBs, thus greatly reducing the number of ACK packets that the source needs to handle. In there is a "Level 3", the receiver hosts at "Level 2" can also be converted into HBs. The only change required in the current implementation of Split TCP would be to modify the code that checks the supported destination IP addresses.

- **Mobile TCP/IP**:

  Currently a lot of research is being focused on mobile TCP/IP over cellular networks. Cellular networks face the same network problems as wireless networks. Thus Split TCP can be and has been [2] used to increase the data rate of data traffic over cellular networks. This is achieved by converting the cell towers (or access points) into HBs.

An interesting issue is to provide a reliable data rate even when the MH does cell switching. The current implementation of Split TCP can be enhanced to support this feature. An inter HB protocol can be designed to exchange the state of a TCP connection when the respective MH switches cells. Thus the new HB will take over the TCP connection and continue to provide an increased data rate.

APPENDIX A

LIST OF ACRONYMS

cwnd	Congestion Window
qdisc	Queueing Discipline
skb	[sk_buff]Socket Buffer
AH	Authentication Header
ARP	Address Resolution Protocol
BER	Bit-Error Rate
BGP	Border Gateway Protocol
BH	Bottom Halve
CE	Congestion Experienced
CWR	Congestion Window Reduced
DNS	Domain Name System
ECE	ECN-Echo
ECN	Explicit Congestion Notification
ESP	Encapsulating Security Protocol
FEC	Forward Error Correction
FH	Fixed Host
FIFO	First In First Out

FTP	File Transfer Protocol
HardIRQ	Hardware Interrupt Request
HB	Helper Box
ICMP	Internet Control Message Protocol
IEPM	Internet End-to-end Performance Monitoring
IP	Internet Protocol
ISR	Interrupt Service Routine
LAN	Local Area Network
LKLM	Linux Kernel Loadable Module
MAC	Media Access Control
MH	Mobile Host
MSR	Mobile Support Routers
MSS	Maximum Segment Size
MTU	Maximum Transfer Unit
NAT	Network Address Translation
OSI	Open Standards Interconnect
OSPF	Open Shortest Path First
PO	Post Office
RIP	Routing Information Protocol
RTO	Retransmission Timeout

RTT Round Trip Time

SACK Selective Acknowledgement

SH Supervisor Host

SNR Signal to Noise Ratio

SoftIRQ Software Interrupt Request

SRP Selective Repeat Protocol

TCP Transmission Control Protocol

TTL Time To Live

UDP User Datagram Protocol

WAN Wide Area Network

APPENDIX B

INSTALLATION MANUAL

This appendix is a HOWTO manual for installing and making use of the Split TCP software in a Linux system.

B.0.1 Getting the software

The Split TCP software is available in source format at the following webpage:

http://web.njit.edu/r̃bj2

Download the file *splittcp.tar.gz.*

B.0.2 What does the tar file contain

The *splittcp.tar.gz* file contains the following 5 files:

1. *ip_in_intercept.c*

   This file is the kernel module and contains the core packet processing code.

2. *split_helper.c*

   This file contains support functions for the Split TCP code.

3. *split_helper.h*

   This file contains variable and structure declarations used in Split TCP code.

4. *Makefile*

   This is the makefile used to compile the kernel module.

5. *README*

   This file contains the material presented in this appendix.

B.0.3 Installing the software

Unzip and untar the file to any directory in the Linux system, preferably in */home.* Off the 3 C files, the *split_helper.** files need to be compiled in the kernel source tree.

Assuming that the kernel source files are located at */usr/src/linux/*, do the following steps.

1. Copy *split_helper.c* in */usr/src/linux/net/ipv4/* directory.
2. Modify the *Makefile* in */usr/src/linux/net/ipv4/* to include "*split_helper.o*" at the end of the string for "*obj-y*".
3. Copy *split_helper.h* in */usr/src/linux/include/linux/* directory.
4. Compile a fresh copy of the kernel. Unless a change is made to the *split_helper.** files, these steps need to be repeated.

B.0.4 Configuring the kernel module

Currently the Split TCP code works for networks with a

16 subnet mask. This is reflected through the variables DEST_IP_MASK (currently 0xffffff) and SRC_IP_MASK (currently 0xffffff). These variables should be modified according to the subnet mask of the source and destination network address.

B.0.5 Configuring the Makefile

The *Makefile* contains the mapping between the end hosts and HBs. It feeds this mapping as command line arguments to the kernel module. Thus depending of the network setup, this mapping should be changed. Consider the network setup of Fig. B.1 to understand the format in which this mapping is written.

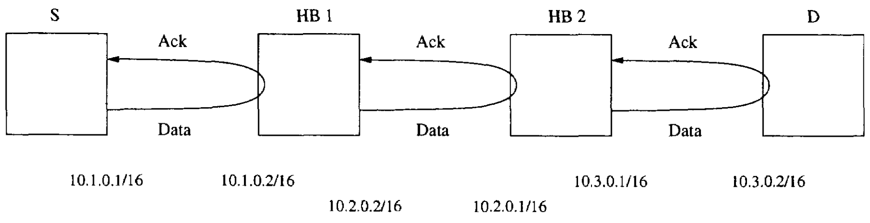

Figure B.1 Sample network setup.

The mapping is stored in a 4x4 table. Each row represents the following fields in order - source network address, destination network address, next HB IP address and previous HB IP address. In case there is no next or previous HB, a value of 0x0 is stored instead. The mapping stored at HB 1 is shown below:

```
neigh_hb_table=0x10a000,0x30a000,0x100020a,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
0x0,0x0,0x0,0x0
```

where

Source Net ID	Destination Net ID	Next HB IP address	Previous HB IP address
10.1.0.0	10.3.0.0	10.2.0.1	None
0x10a000 (left shifted by 12)	0x30a000 (left shifter by 12)	0x100020a	0x0

Similarly the mapping stored at HB 2 is shown below:

```
neigh_hb_table=0x10a000,0x30a000,0x0,0x200020a,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
0x0,0x0,0x0,0x0
```

APPENDIX C

SPLIT TCP CODE

This appendix lists the Split TCP code introduced in the Linux system.

C.1 split_helper.h

This file contains variable and structure declarations that are used in the Split TCP code.

```
/*
 * Created:     02/01/2005
 *
 * Author:      Rahul Jain
 *
 * Filename:    split_helper.c
 *
 * Comment:
 *    This file contains functions that will manipulate
 *    the structs defined in split_helper.h
 *
 */

#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/split_helper.h>
#include <net/inet_ecn.h>

/* Global Variable */
struct split_flow_info *sfi_list_head;

/*
 *      Function used to initialize the head_pkt linked
 *      list.
 */
void init_skbuff_list(struct split_flow_info *sfi)
{
    struct skbuff_list *list_head;

    /* Initializing the i2r queue */
    sfi->i2r_queue = kmalloc(sizeof(struct skbuff_list),
                                GFP_ATOMIC);
    list_head = sfi->i2r_queue;
    list_head->next = list_head->prev = list_head;
    list_head->pkt_state = SENT;
    list_head->hole_in_queue = 0;
    list_head->sb_pkt = NULL;
    list_head->pkt_bfr_hole = NULL;
    list_head->pkt_count = 0;
    list_head->tps_ptr = sfi->rhs_tcp_state;
    list_head->lock = RW_LOCK_UNLOCKED;

    /* Initializing the r2i queue */
    sfi->r2i_queue = kmalloc(sizeof(struct skbuff_list),
                                GFP_ATOMIC);
    list_head = sfi->r2i_queue;
    list_head->next = list_head->prev = list_head;
    list_head->pkt_state = SENT;
    list_head->hole_in_queue = 0;
    list_head->sb_pkt = NULL;
    list_head->pkt_bfr_hole = NULL;
    list_head->pkt_count = 0;
    list_head->tps_ptr = sfi->lhs_tcp_state;
    list_head->lock = RW_LOCK_UNLOCKED;

}

/*
 *      Function used to add a new packet to the list.
 *      List is arranged in FIFO manner so the new
 *      node is added at the end of the list.
 */
void enqueue_skbuff_list(struct split_flow_info *sfi,
                         struct skbuff_list *newpkt,
                         struct skbuff_list *queue_head)
{
    struct skbuff_list *prev , *next;
    struct skbuff_list *head = queue_head;

    prev = head->prev;
    next = head;

```

```
   prev->next = newpkt;
   newpkt->prev = prev;
   newpkt->next = next;
   next->prev = newpkt;

   newpkt->pkt_state = NOT_SENT;
   ++head->pkt_count;
}

/*
 *      Function used to insert a new packet after the
 *      given pointer.
 */
void insert_skbuff_list(struct skbuff_list *after,
                        struct skbuff_list *new_pkt,
                        struct skbuff_list *queue_head)
{
   struct skbuff_list *before = after->next;

   after->next = new_pkt;
   before->prev = new_pkt;
   new_pkt->prev = after;
   new_pkt->next = before;

   new_pkt->pkt_state = NOT_SENT;
   queue_head->pkt_count += 1;
}

/*
 *      Function used to dequeue a packet from the list.
 *      The list is arranged in FIFO manner so the node
 *      is removed from the head of the list.
 */
struct skbuff_list*
dequeue_skbuff_list(struct split_flow_info *sfi,
                    struct skbuff_list *queue_head)
{
   struct skbuff_list *prev , *next, *ret_node;
   struct skbuff_list *head = queue_head;

   prev = head;
   next = head->next->next;
   ret_node = head->next;

   prev->next = next;
   next->prev = prev;

   ret_node->prev = ret_node->next = NULL;
   --head->pkt_count;
   return(ret_node);
}

/*
 *      Function used to dequeue and free memory for a
 *      packet from the list. The list is arranged in
 *      FIFO manner so the node is removed from the
 *      head of the list.
 */
void free_head_skbuff_list(struct split_flow_info *sfi,
                           struct skbuff_list *queue_head)
{
   struct skbuff_list *prev , *next, *ret_node;
   struct skbuff_list *head = queue_head;
   struct sk_buff *skb;

   /* Initialize the pointers */
   prev = head;
   next = head->next->next;
   ret_node = head->next;

   /* Adjust the pointers */
   prev->next = next;
   next->prev = prev;

   /* Update pointers and packet count */
   ret_node->prev = ret_node->next = NULL;
   --head->pkt_count;

   /* Unlink the skb from the list and free skb memory */
   skb = ret_node->sb_pkt;
// if(skb->list)
//    __skb_unlink(skb, skb->list);

   /* Taking care of dst_release BUG */
   if(skb->dst) {
      if(atomic_read(&skb->dst->__refcnt) < 1)
         atomic_set(&skb->dst->__refcnt, 1);
   }

   kfree_skb(skb);

   /* Free skbuff_list node */
   kfree(ret_node);
}

/*
 *      This function peeks inside the list and returns
 *      the skb at the head of the forward queue.
 */
struct skbuff_list*
head_peek_skb_list(struct skbuff_list *queue_head)
{
   struct skbuff_list *head = queue_head;

   if(head->next != NULL &&
      head->next != head)
        return head->next;

   return NULL;
}

/*
 *      This function checks if the forward-queue is empty.
 */
int get_queue_pkt_count(struct skbuff_list *queue_head)
{
   return queue_head->pkt_count;
}

/*
 *      Function used to initialize the doubly linked list
 *      containing split_flow_info nodes.
 */
void init_head_sfi()
{
   sfi_list_head = kmalloc(sizeof(struct split_flow_info),
                              GFP_ATOMIC);
   sfi_list_head->prev = sfi_list_head->next
                       = sfi_list_head;
}

/*
 *      Function returns a pointer to the variable
 *      sfi_list_head
 */
```

```
struct split_flow_info* get_head_sfi()
{
   return sfi_list_head;
}

/*
 *      Function returns a pointer to the member in_flow
 */
struct flow_detail*
get_in_flow(struct split_flow_info *node, int flag)
{
   if(flag == PREV_FLOW) {
      if(node->i2r_flow != NULL)
         return node->i2r_flow;
   }
   else if(flag == FWD_FLOW) {
      if(node->r2i_flow != NULL)
         return node->r2i_flow;
   }
   return NULL;
}

/*
 *      Function returns a pointer to the member
 *      rep_in_flow
 */
struct flow_detail*
get_rep_in_flow(struct split_flow_info *node)
{
   if(node->rep_in_flow != NULL)
      return node->rep_in_flow;
   return NULL;
}

/*
 *      Function enqueues a new node to the list. The
 *      linked list is arranged as a FIFO, hence the
 *      node is added at the tail of the list.
 */
struct split_flow_info*
enqueue_sfi(struct split_flow_info *head)
{
   struct split_flow_info *prev, *next;
   struct split_flow_info *new_sfi =
         (struct split_flow_info *)
```

```
            kmalloc(sizeof(struct split_flow_info),
                    GFP_ATOMIC);
    new_sfi->lhs_tcp_state = (struct tcp_state *)
            kmalloc(sizeof(struct tcp_state), GFP_ATOMIC);
    new_sfi->rhs_tcp_state = (struct tcp_state *)
            kmalloc(sizeof(struct tcp_state), GFP_ATOMIC);

    next = head;
    prev = head->prev;

    new_sfi->prev = prev;
    prev->next = new_sfi;
    new_sfi->next = next;
    next->prev = new_sfi;

    new_sfi->buff_clamp = 1048576;
    new_sfi->buff_curr = 0;

    /* Initialize the state for the connections */
    new_sfi->lhs_tcp_state->state = TCP_LISTEN;
    new_sfi->rhs_tcp_state->state = TCP_LISTEN;

    /* Initializing the rto to 3 secs and srtt to 0 */
    new_sfi->lhs_tcp_state->rto =
    new_sfi->rhs_tcp_state->rto = TCP_RTO_INIT;
    new_sfi->lhs_tcp_state->srtt =
    new_sfi->rhs_tcp_state->srtt = 0;
    new_sfi->lhs_tcp_state->ack_seq_tstamp = 0;
    new_sfi->rhs_tcp_state->ack_seq_tstamp = 0;
    new_sfi->lhs_tcp_state->rtt_seq_tstamp = 0;
    new_sfi->rhs_tcp_state->rtt_seq_tstamp = 0;
    new_sfi->lhs_tcp_state->finack_retrans = 0;
    new_sfi->rhs_tcp_state->finack_retrans = 0;
    new_sfi->lhs_tcp_state->in_fast_recovery = 0;
    new_sfi->rhs_tcp_state->in_fast_recovery = 0;
    new_sfi->lhs_tcp_state->cwnd_cnt = 0;
    new_sfi->rhs_tcp_state->cwnd_cnt = 0;
    new_sfi->lhs_tcp_state->pkts_in_flight = 0;
    new_sfi->rhs_tcp_state->pkts_in_flight = 0;
    new_sfi->lhs_tcp_state->first_good_ack = 0;
    new_sfi->rhs_tcp_state->first_good_ack = 0;
    new_sfi->lhs_tcp_state->probes_out = 0;
    new_sfi->rhs_tcp_state->probes_out = 0;
    new_sfi->lhs_tcp_state->local_ipip_addr = 0;
    new_sfi->rhs_tcp_state->local_ipip_addr = 0;
```

```
   new_sfi->lhs_tcp_state->data_pkt_seen = 0;
   new_sfi->rhs_tcp_state->data_pkt_seen = 0;

   /* Initializing wscale variables */
   new_sfi->lhs_tcp_state->wscale_ok =
   new_sfi->rhs_tcp_state->wscale_ok = 0;
   new_sfi->lhs_tcp_state->snd_wscale = 0;
   new_sfi->lhs_tcp_state->rcv_wscale = 0;
   new_sfi->rhs_tcp_state->snd_wscale = 0;
   new_sfi->rhs_tcp_state->rcv_wscale = 0;

   return new_sfi;
}

/*
 *      Function used to dequeue a node from the list.
 *      The linked list is arranged as a FIFO, hence
 *      the node is removed from the head of the list.
 */
struct split_flow_info*
dequeue_sfi(struct split_flow_info *head)
{
   struct split_flow_info *prev, *next, *ret_node;

   prev = head;
   next = head->next->next;

   ret_node = prev->next;
   prev->next = next;
   next->prev = prev;
   ret_node->next = ret_node->prev = NULL;
   return(ret_node);
}

/*
 *      This function frees up the memory allocated to
 *      a sfi node
 */
void delete_sfi(struct split_flow_info *sfi)
{
   struct skbuff_list *skb_curr, *skb_prev;

   /* Rearrange the prev and next pointers */
   sfi->prev->next = sfi->next;
   sfi->next->prev = sfi->prev;
   sfi->next = sfi->prev = NULL;

   skb_curr = sfi->i2r_queue->next;

   while(skb_curr != sfi->i2r_queue) {
      skb_prev = skb_curr;
      skb_curr = skb_curr->next;
      kfree_skb(skb_prev->sb_pkt);
      kfree(skb_prev);
   }
   kfree(skb_curr);

   skb_curr = sfi->r2i_queue->next;

   while(skb_curr != sfi->r2i_queue) {
      skb_prev = skb_curr;
      skb_curr = skb_curr->next;
      kfree_skb(skb_prev->sb_pkt);
      kfree(skb_prev);
   }
   kfree(skb_curr);

   kfree(sfi);
}

/*
 *      Function used to search for a split_flow_info.
 *      The function returns a 1 on success and -1 on
 *      failure. Search is done using the incoming flow
 *      details.
 */
struct split_flow_info*
search_sfi(struct split_flow_info *head,
               struct flow_detail *fd, int flag)
{
   struct split_flow_info *curr;
   struct flow_detail *sfi_flow;
   __u32 saddr, daddr;
   __u16 sport, dport;

   saddr = fd->saddr;
   daddr = fd->daddr;
   sport = fd->sport;
   dport = fd->dport;
   curr = head->next;

   if(flag == PREV_FLOW) {
      while(curr != head) {
         sfi_flow = curr->i2r_flow;
         if(sfi_flow->saddr == saddr &&
            sfi_flow->daddr == daddr &&
            sfi_flow->sport == sport &&
            sfi_flow->dport == dport)
               return curr;
         curr = curr->next;
       }
   }
   else if(flag == FWD_FLOW) {
      while(curr != head) {
         sfi_flow = curr->r2i_flow;
         if(sfi_flow->saddr == saddr &&
            sfi_flow->daddr == daddr &&
            sfi_flow->sport == sport &&
            sfi_flow->dport == dport) {
               return curr;
         curr = curr->next;
      }
   }

   printk(KERN_INFO "Search failed\n");
   return NULL;
}

/*
 *      This function frees up the memory allocated to
 *      the doubly linked list
 */
void cleanup_sfi_list(struct split_flow_info *head)
{
   struct split_flow_info *sfi_curr, *sfi_prev;
   struct skbuff_list *skb_curr, *skb_prev;
   struct tcp_state *tps;

   if(head != NULL) {
      sfi_curr = head->next;

      while(sfi_curr != head) {
         skb_curr = sfi_curr->i2r_queue->next;

         while(skb_curr != sfi_curr->i2r_queue) {
            skb_prev = skb_curr;
            skb_curr = skb_curr->next;
            kfree(skb_prev);
         }
         kfree(skb_curr);

         skb_curr = sfi_curr->r2i_queue->next;

         while(skb_curr != sfi_curr->r2i_queue) {
            skb_prev = skb_curr;
            skb_curr = skb_curr->next;
            kfree(skb_prev);
         }
         kfree(skb_curr);

         /* Deleting the various timers */
         tps = sfi_curr->lhs_tcp_state;
         if(timer_pending(&(tps->rto_timer)))
            del_timer_sync(&(tps->rto_timer));

         if(timer_pending(&(tps->tw_timer)))
            del_timer_sync(&(tps->tw_timer));

         if(timer_pending(&(tps->probe_timer)))
            del_timer_sync(&(tps->probe_timer));

         tps = sfi_curr->rhs_tcp_state;
         if(timer_pending(&(tps->rto_timer)))
            del_timer_sync(&(tps->rto_timer));

         if(timer_pending(&(tps->tw_timer)))
            del_timer_sync(&(tps->tw_timer));

         if(timer_pending(&(tps->probe_timer)))
            del_timer_sync(&(tps->probe_timer));

         sfi_prev = sfi_curr;
         sfi_curr = sfi_curr->next;
         kfree(sfi_prev);
      }
      kfree(sfi_curr);
   }
}

MODULE_LICENSE("GPL");
```

```

EXPORT_SYMBOL(init_skbuff_list);
EXPORT_SYMBOL(enqueue_skbuff_list);
EXPORT_SYMBOL(insert_skbuff_list);
EXPORT_SYMBOL(dequeue_skbuff_list);
EXPORT_SYMBOL(free_head_skbuff_list);
EXPORT_SYMBOL(head_peek_skb_list);
EXPORT_SYMBOL(get_queue_pkt_count);
EXPORT_SYMBOL(init_head_sfi);
EXPORT_SYMBOL(get_head_sfi);
EXPORT_SYMBOL(get_in_flow);
EXPORT_SYMBOL(get_rep_in_flow);
EXPORT_SYMBOL(enqueue_sfi);
EXPORT_SYMBOL(dequeue_sfi);
EXPORT_SYMBOL(delete_sfi);
EXPORT_SYMBOL(search_sfi);
EXPORT_SYMBOL(cleanup_sfi_list);

/*
 *  This code below was written to implement our own ECN
 *  marking software. It still needs to be tested to be
 *  integrated with the working code of Split TCP
 */

/****************************************************/
/*          Function for RED Qdisc operation        */
/****************************************************/
/* Update: 04/20/07 Temporarily not used. Relying on
 * kernel support for marking */


//int packet_count;

/*
 * This function marks the packet ECN style
 */
/*
static int split_red_ecn_mark(struct sk_buff *skb)
{
   if (skb->nh.raw + 20 > skb->tail)
      return 0;

   switch (skb->protocol) {
      case __constant_htons(ETH_P_IP):
         if (!INET_ECN_is_capable(skb->nh.iph->tos)) {
                printk(KERN_ALERT "Flow not ECN capable \n");
                return 0;
            }

            if(INET_ECN_is_not_ect(skb->nh.iph->tos))
                IP_ECN_set_ce(skb->nh.iph);
                return 1;
        case __constant_htons(ETH_P_IPV6):
            if (!INET_ECN_is_capable(ipv6_get_dsfield(skb->nh.ipv6h)))
                return 0;
            IP6_ECN_set_ce(skb->nh.ipv6h);
            return 1;
        default:
            return 0;
    }
}*/

/*
 * This function contains the main alogrithm of RED
 * Enqueue packet with or without mark or drop the packet
 */
//int
//split_red_enqueue(struct sk_buff *skb, struct Qdisc *sch)
//{
//   struct split_red_sched_data *q =
//                  (struct split_red_sched_data *)sch->data;
//   struct split_red_sched_data *q = qdisc_priv(sch);
///  psched_time_t now;


     /* Processing for queue idle time */
/*   if(!PSCHED_IS_PASTPERFECT(q->qidlestart)) {
         long us_idle;
         int shift;

         PSCHED_GET_TIME(now);
         us_idle = PSCHED_TDIFF_SAFE(now, q->qidlestart,
                                     q->Scell_max, 0);
         PSCHED_SET_PASTPERFECT(q->qidlestart);
*/
         /* Do avg = (1-Wq)^m * avg here */
/*       index = ;
         shift = q->Stab[index];
         if(shift)
             q->qave <<= shift;
        }
*/
        /* Queue is not empty */
//      else {
//         q->qave +=
//               (sch->qstats.backlog - q->qave) >> q->Wlog;
//      }

        /* Marking decision */
//      if(q->qave <= q->qth_min) {
           /* Enqueue the packet */
//         q->qcount = -1;
//enqueue:
//         if(sch->qstats.backlog + skb->len <= q->limit) {
//            __skb_queue_tail(&sch->q, skb);
//            sch->qstats.backlog += skb->len;
//            sch->bstats.bytes += skb->len;
//            sch->bstats.packets++;
//            sch->qstats.qlen++;
//            packet_count++;
//            printk(KERN_ALERT "Enqueue blindly \n");
//            return NET_XMIT_SUCCESS;
//         }
//         q->st.pdrop++;
//         kfree_skb(skb);
//         sch->qstats.drops++;
//         return NET_XMIT_DROP;
//      }
//      else if(q->qave > q->qth_min &&
//               q->qave <= q->qth_max) {
           /* Mark the packet with random probability */
//         printk(KERN_ALERT "minth < qave < maxth \n");
//         if(++q->qcount) {
//            if((((q->qave - q->qth_min)>>q->Plog)/
//                 (q->qth_max - q->qth_min))*q->qcount < q->qR)
//               goto enqueue;

//            q->qcount = 0;
//            q->qR = net_random()&q->Rmask;
//            sch->qstats.overlimits++;
//            goto mark;
//         }
//         if(q->qcount == 0) {
//            q->qR = net_random()&q->Rmask;
//            sch->qstats.overlimits++;
//      }
//   }
//   else if(q->qave > q->qth_max) {
        /* Mark the packet */
//      q->qcount = -1;
//      sch->qstats.overlimits++;
//mark:
//      if(!(q->flags&TC_RED_ECN) ||
//         !split_red_ecn_mark(skb)) {
//         if(!split_red_ecn_mark(skb)) {
//            q->st.early++;
//            goto drop;
//         }
//         q->st.marked++;
//         goto enqueue;
//      }
//drop:
//   kfree_skb(skb);
//   sch->qstats.overlimits++;
//   return NET_XMIT_CN;
//}

/*
 * This function requeues the packet
 */
/*
int
split_red_requeue(struct sk_buff *skb, struct Qdisc* sch)
{
   struct split_red_sched_data *q = qdisc_priv(sch);

   PSCHED_SET_PASTPERFECT(q->qidlestart);

   __skb_queue_head(&sch->q, skb);
   sch->qstats.backlog += skb->len;
   return 0;
}
*/

/*
 * This function will dequeue and get a packet ready for
 * transmission
 */
/*
struct sk_buff* split_red_dequeue(struct Qdisc* sch)
```

```
{
   struct sk_buff *skb;
   struct split_red_sched_data *q = qdisc_priv(sch);

   skb = __skb_dequeue(&sch->q);


   if(skb) {
//    if(skb->h.th != NULL) {
//       if(skb->len < 1000 || skb->h.th->fin || skb->h.th->syn ||
//          sch->qstats.packets < 10 || sch->stats.qlen > 10) {
send_packet:
            packet_count--;
            sch->qstats.qlen--;
            sch->qstats.backlog -= skb->len;
            return skb;
//       }
//    }
//    else
//       goto send_packet;
   }

   PSCHED_GET_TIME(q->qidlestart);
   return NULL;
}
*/

/*
 * This function drops the packet
 */
/*
unsigned int split_red_drop(struct Qdisc* sch)
{
   struct sk_buff *skb;
   struct split_red_sched_data *q = qdisc_priv(sch);

   skb = __skb_dequeue_tail(&sch->q);
   if (skb) {
      unsigned int len = skb->len;
      sch->qstats.backlog -= len;
      sch->qstats.drops++;
      q->st.other++;
      kfree_skb(skb);
      packet_count--;
      sch->qstats.qlen--;
```

```
        return len;
    }
    PSCHED_GET_TIME(q->qidlestart);
    return 0;
}

void split_red_reset(struct Qdisc* sch)
{
    struct split_red_sched_data *q = qdisc_priv(sch);

    __skb_queue_purge(&sch->q);
    sch->qstats.backlog = 0;
    PSCHED_SET_PASTPERFECT(q->qidlestart);
    q->qave = 0;
    q->qcount = -1;
}
*/

/*
 * This function sets the parameters used by RED
 */
/*
int split_red_change(struct Qdisc *sch)
{
    struct split_red_sched_data *q = qdisc_priv(sch);

    packet_count = 0;

    sch_tree_lock(sch);
    q->flags = 0;
    q->Wlog = 9;
    q->Plog = 5;
    q->Rmask = q->Plog < 32 ? ((1<<q->Plog) - 1) : ~0UL;
//  q->Scell_log = ctl->Scell_log;
//  q->Scell_max = (255<<q->Scell_log);
    q->qth_min = 5000;
    q->qth_max = 15000;
    q->limit = 75000;
    q->qcount = -1;
    q->qave = 0;
    sch->qstats.backlog = 0;
    sch->bstats.packets = 0;
    sch->qstats.qlen = 0;

    if (skb_queue_len(&sch->q) == 0)
```

```
        PSCHED_SET_PASTPERFECT(q->qidlestart);

    sch_tree_unlock(sch);
    return 0;
}

int split_red_init(struct Qdisc* sch, struct rtattr *opt)
{
    return split_red_change(sch);
}

int split_red_copy_xstats(struct sk_buff *skb,
                                struct tc_red_xstats *st)
{
    RTA_PUT(skb, TCA_XSTATS, sizeof(*st), st);
    return 0;

rtattr_failure:
    return 1;
}

int split_red_dump(struct Qdisc *sch, struct sk_buff *skb)
{
    struct split_red_sched_data *q = qdisc_priv(sch);
    unsigned char    *b = skb->tail;
    struct rtattr *rta;
    struct tc_red_qopt opt;

    rta = (struct rtattr*)b;
    RTA_PUT(skb, TCA_OPTIONS, 0, NULL);
    opt.limit = q->limit;
    opt.qth_min = q->qth_min>>q->Wlog;
    opt.qth_max = q->qth_max>>q->Wlog;
    opt.Wlog = q->Wlog;
    opt.Plog = q->Plog;
    opt.Scell_log = q->Scell_log;
    opt.flags = q->flags;
    RTA_PUT(skb, TCA_RED_PARMS, sizeof(opt), &opt);
    rta->rta_len = skb->tail - b;

    if (split_red_copy_xstats(skb, &q->st))
        goto rtattr_failure;

    return skb->len;

```

```
rtattr_failure:
   skb_trim(skb, b - skb->data);
   return -1;
}

void split_red_destroy(struct Qdisc *sch)
{
}

static int __init split_red_module_init(void)
{
   return register_qdisc(&split_red_qdisc_ops);
}
static void __exit split_red_module_exit(void)
{
   unregister_qdisc(&split_red_qdisc_ops);
}

module_init(split_red_module_init)
module_exit(split_red_module_exit)
MODULE_LICENSE("GPL");

EXPORT_SYMBOL(split_red_enqueue);
EXPORT_SYMBOL(split_red_dequeue);
EXPORT_SYMBOL(split_red_requeue);
EXPORT_SYMBOL(split_red_drop);
EXPORT_SYMBOL(split_red_init);
EXPORT_SYMBOL(split_red_reset);
EXPORT_SYMBOL(split_red_destroy);
EXPORT_SYMBOL(split_red_change);
EXPORT_SYMBOL(split_red_dump);
EXPORT_SYMBOL(split_red_qdisc_ops);
*/
```

C.2 split_helper.c

This file contains various support functions that are used by the kernel module.

```
/*
 * Created:     02/01/2005
 *
 * Author:      Rahul Jain
 *
 * Filename:    split_helper.c
 *
 * Comment:
 *    This file contains functions that will manipulate
 *    the structs defined in split_helper.h
 *
 */

#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/split_helper.h>
#include <net/inet_ecn.h>

/* Global Variable */
struct split_flow_info *sfi_list_head;

/*
 *      Function used to initialize the head_pkt linked
 *      list.
 */
void init_skbuff_list(struct split_flow_info *sfi)
{
   struct skbuff_list *list_head;

   /* Initializing the i2r queue */
   sfi->i2r_queue = kmalloc(sizeof(struct skbuff_list),
                            GFP_ATOMIC);
   list_head = sfi->i2r_queue;
   list_head->next = list_head->prev = list_head;
   list_head->pkt_state = SENT;
   list_head->hole_in_queue = 0;
   list_head->sb_pkt = NULL;
   list_head->pkt_bfr_hole = NULL;
   list_head->pkt_count = 0;
   list_head->tps_ptr = sfi->rhs_tcp_state;
   list_head->lock = RW_LOCK_UNLOCKED;

   /* Initializing the r2i queue */
   sfi->r2i_queue = kmalloc(sizeof(struct skbuff_list),
                            GFP_ATOMIC);
   list_head = sfi->r2i_queue;
   list_head->next = list_head->prev = list_head;
   list_head->pkt_state = SENT;
   list_head->hole_in_queue = 0;
   list_head->sb_pkt = NULL;
   list_head->pkt_bfr_hole = NULL;
   list_head->pkt_count = 0;
   list_head->tps_ptr = sfi->lhs_tcp_state;
   list_head->lock = RW_LOCK_UNLOCKED;

}

/*
 *      Function used to add a new packet to the list.
 *      List is arranged in FIFO manner so the new
 *      node is added at the end of the list.
 */
void enqueue_skbuff_list(struct split_flow_info *sfi,
                         struct skbuff_list *newpkt,
                         struct skbuff_list *queue_head)
{
   struct skbuff_list *prev , *next;
   struct skbuff_list *head = queue_head;

   prev = head->prev;
   next = head;

   prev->next = newpkt;
   newpkt->prev = prev;
   newpkt->next = next;
   next->prev = newpkt;

   newpkt->pkt_state = NOT_SENT;
   ++head->pkt_count;
}

/*
 *      Function used to insert a new packet after the
 *      given pointer.
 */
void insert_skbuff_list(struct skbuff_list *after,
                        struct skbuff_list *new_pkt,
                        struct skbuff_list *queue_head)
{
   struct skbuff_list *before = after->next;

   after->next = new_pkt;
   before->prev = new_pkt;
   new_pkt->prev = after;
   new_pkt->next = before;

   new_pkt->pkt_state = NOT_SENT;
   queue_head->pkt_count += 1;
}

/*
 *      Function used to dequeue a packet from the list.
 *      The list is arranged in FIFO manner so the node
 *      is removed from the head of the list.
 */
struct skbuff_list*
dequeue_skbuff_list(struct split_flow_info *sfi,
                    struct skbuff_list *queue_head)
{
   struct skbuff_list *prev , *next, *ret_node;
   struct skbuff_list *head = queue_head;

   prev = head;
   next = head->next->next;
   ret_node = head->next;

   prev->next = next;
   next->prev = prev;

   ret_node->prev = ret_node->next = NULL;
   --head->pkt_count;
   return(ret_node);
}

/*
 *      Function used to dequeue and free memory for a
 *      packet from the list. The list is arranged in
 *      FIFO manner so the node is removed from the
 *      head of the list.
 */
void free_head_skbuff_list(struct split_flow_info *sfi,
                           struct skbuff_list *queue_head)
{
   struct skbuff_list *prev , *next, *ret_node;
   struct skbuff_list *head = queue_head;
   struct sk_buff *skb;

   /* Initialize the pointers */
   prev = head;
   next = head->next->next;
   ret_node = head->next;

   /* Adjust the pointers */
   prev->next = next;
   next->prev = prev;

   /* Update pointers and packet count */
   ret_node->prev = ret_node->next = NULL;
   --head->pkt_count;

   /* Unlink the skb from the list and free skb memory */
   skb = ret_node->sb_pkt;
// if(skb->list)
//    __skb_unlink(skb, skb->list);

   /* Taking care of dst_release BUG */
   if(skb->dst) {
      if(atomic_read(&skb->dst->__refcnt) < 1)
         atomic_set(&skb->dst->__refcnt, 1);
   }

   kfree_skb(skb);

   /* Free skbuff_list node */
   kfree(ret_node);
}

/*
 *      This function peeks inside the list and returns
 *      the skb at the head of the forward queue.
 */
struct skbuff_list*
head_peek_skb_list(struct skbuff_list *queue_head)
{
   struct skbuff_list *head = queue_head;

   if(head->next != NULL &&
      head->next != head)
        return head->next;

   return NULL;
}

/*
```

```
 *      This function checks if the forward-queue is empty.
 */
int get_queue_pkt_count(struct skbuff_list *queue_head)
{
   return queue_head->pkt_count;
}

/*
 *      Function used to initialize the doubly linked list
 *      containing split_flow_info nodes.
 */
void init_head_sfi()
{
   sfi_list_head = kmalloc(sizeof(struct split_flow_info),
                           GFP_ATOMIC);
   sfi_list_head->prev = sfi_list_head->next
                       = sfi_list_head;
}

/*
 *      Function returns a pointer to the variable
 *      sfi_list_head
 */
struct split_flow_info* get_head_sfi()
{
   return sfi_list_head;
}

/*
 *      Function returns a pointer to the member in_flow
 */
struct flow_detail*
get_in_flow(struct split_flow_info *node, int flag)
{
   if(flag == PREV_FLOW) {
      if(node->i2r_flow != NULL)
         return node->i2r_flow;
   }
   else if(flag == FWD_FLOW) {
      if(node->r2i_flow != NULL)
         return node->r2i_flow;
   }
   return NULL;
}

```

```
/*
 *      Function returns a pointer to the member
 *      rep_in_flow
 */
struct flow_detail*
get_rep_in_flow(struct split_flow_info *node)
{
   if(node->rep_in_flow != NULL)
      return node->rep_in_flow;
   return NULL;
}

/*
 *      Function enqueues a new node to the list. The
 *      linked list is arranged as a FIFO, hence the
 *      node is added at the tail of the list.
 */
struct split_flow_info*
enqueue_sfi(struct split_flow_info *head)
{
   struct split_flow_info *prev, *next;
   struct split_flow_info *new_sfi =
           (struct split_flow_info *)
           kmalloc(sizeof(struct split_flow_info),
                   GFP_ATOMIC);
   new_sfi->lhs_tcp_state = (struct tcp_state *)
           kmalloc(sizeof(struct tcp_state), GFP_ATOMIC);
   new_sfi->rhs_tcp_state = (struct tcp_state *)
           kmalloc(sizeof(struct tcp_state), GFP_ATOMIC);

   next = head;
   prev = head->prev;

   new_sfi->prev = prev;
   prev->next = new_sfi;
   new_sfi->next = next;
   next->prev = new_sfi;

   new_sfi->buff_clamp = 1048576;
   new_sfi->buff_curr = 0;

   /* Initialize the state for the connections */
   new_sfi->lhs_tcp_state->state = TCP_LISTEN;
   new_sfi->rhs_tcp_state->state = TCP_LISTEN;

```

```
    /* Initializing the rto to 3 secs and srtt to 0 */
    new_sfi->lhs_tcp_state->rto =
    new_sfi->rhs_tcp_state->rto = TCP_RTO_INIT;
    new_sfi->lhs_tcp_state->srtt =
    new_sfi->rhs_tcp_state->srtt = 0;
    new_sfi->lhs_tcp_state->ack_seq_tstamp = 0;
    new_sfi->rhs_tcp_state->ack_seq_tstamp = 0;
    new_sfi->lhs_tcp_state->rtt_seq_tstamp = 0;
    new_sfi->rhs_tcp_state->rtt_seq_tstamp = 0;
    new_sfi->lhs_tcp_state->finack_retrans = 0;
    new_sfi->rhs_tcp_state->finack_retrans = 0;
    new_sfi->lhs_tcp_state->in_fast_recovery = 0;
    new_sfi->rhs_tcp_state->in_fast_recovery = 0;
    new_sfi->lhs_tcp_state->cwnd_cnt = 0;
    new_sfi->rhs_tcp_state->cwnd_cnt = 0;
    new_sfi->lhs_tcp_state->pkts_in_flight = 0;
    new_sfi->rhs_tcp_state->pkts_in_flight = 0;
    new_sfi->lhs_tcp_state->first_good_ack = 0;
    new_sfi->rhs_tcp_state->first_good_ack = 0;
    new_sfi->lhs_tcp_state->probes_out = 0;
    new_sfi->rhs_tcp_state->probes_out = 0;
    new_sfi->lhs_tcp_state->local_ipip_addr = 0;
    new_sfi->rhs_tcp_state->local_ipip_addr = 0;
    new_sfi->lhs_tcp_state->data_pkt_seen = 0;
    new_sfi->rhs_tcp_state->data_pkt_seen = 0;

    /* Initializing wscale variables */
    new_sfi->lhs_tcp_state->wscale_ok =
    new_sfi->rhs_tcp_state->wscale_ok = 0;
    new_sfi->lhs_tcp_state->snd_wscale = 0;
    new_sfi->lhs_tcp_state->rcv_wscale = 0;
    new_sfi->rhs_tcp_state->snd_wscale = 0;
    new_sfi->rhs_tcp_state->rcv_wscale = 0;

    return new_sfi;
}

/*
 *      Function used to dequeue a node from the list.
 *      The linked list is arranged as a FIFO, hence
 *      the node is removed from the head of the list.
 */
struct split_flow_info*
dequeue_sfi(struct split_flow_info *head)
{
   struct split_flow_info *prev, *next, *ret_node;

   prev = head;
   next = head->next->next;

   ret_node = prev->next;
   prev->next = next;
   next->prev = prev;
   ret_node->next = ret_node->prev = NULL;
   return(ret_node);
}

/*
 *      This function frees up the memory allocated to
 *      a sfi node
 */
void delete_sfi(struct split_flow_info *sfi)
{
   struct skbuff_list *skb_curr, *skb_prev;

   /* Rearrange the prev and next pointers */
   sfi->prev->next = sfi->next;
   sfi->next->prev = sfi->prev;
   sfi->next = sfi->prev = NULL;

   skb_curr = sfi->i2r_queue->next;

   while(skb_curr != sfi->i2r_queue) {
      skb_prev = skb_curr;
      skb_curr = skb_curr->next;
      kfree_skb(skb_prev->sb_pkt);
      kfree(skb_prev);
   }
   kfree(skb_curr);

   skb_curr = sfi->r2i_queue->next;

   while(skb_curr != sfi->r2i_queue) {
      skb_prev = skb_curr;
      skb_curr = skb_curr->next;
      kfree_skb(skb_prev->sb_pkt);
      kfree(skb_prev);
   }
   kfree(skb_curr);

   kfree(sfi);
}

/*
 *      Function used to search for a split_flow_info.
 *      The function returns a 1 on success and -1 on
 *      failure. Search is done using the incoming flow
 *      details.
 */
struct split_flow_info*
search_sfi(struct split_flow_info *head,
              struct flow_detail *fd, int flag)
{
   struct split_flow_info *curr;
   struct flow_detail *sfi_flow;
   __u32 saddr, daddr;
   __u16 sport, dport;

   saddr = fd->saddr;
   daddr = fd->daddr;
   sport = fd->sport;
   dport = fd->dport;
   curr = head->next;

   if(flag == PREV_FLOW) {
      while(curr != head) {
         sfi_flow = curr->i2r_flow;
         if(sfi_flow->saddr == saddr &&
            sfi_flow->daddr == daddr &&
            sfi_flow->sport == sport &&
            sfi_flow->dport == dport)
               return curr;
         curr = curr->next;
       }
   }
   else if(flag == FWD_FLOW) {
      while(curr != head) {
         sfi_flow = curr->r2i_flow;
         if(sfi_flow->saddr == saddr &&
            sfi_flow->daddr == daddr &&
            sfi_flow->sport == sport &&
            sfi_flow->dport == dport) {
               return curr;
         curr = curr->next;
      }
```

```
    }

    printk(KERN_INFO "Search failed\n");
    return NULL;
}

/*
 *      This function frees up the memory allocated to
 *      the doubly linked list
 */
void cleanup_sfi_list(struct split_flow_info *head)
{
    struct split_flow_info *sfi_curr, *sfi_prev;
    struct skbuff_list *skb_curr, *skb_prev;
    struct tcp_state *tps;

    if(head != NULL) {
        sfi_curr = head->next;

        while(sfi_curr != head) {
            skb_curr = sfi_curr->i2r_queue->next;

            while(skb_curr != sfi_curr->i2r_queue) {
                skb_prev = skb_curr;
                skb_curr = skb_curr->next;
                kfree(skb_prev);
            }
            kfree(skb_curr);

            skb_curr = sfi_curr->r2i_queue->next;

            while(skb_curr != sfi_curr->r2i_queue) {
                skb_prev = skb_curr;
                skb_curr = skb_curr->next;
                kfree(skb_prev);
            }
            kfree(skb_curr);

            /* Deleting the various timers */
            tps = sfi_curr->lhs_tcp_state;
            if(timer_pending(&(tps->rto_timer)))
                del_timer_sync(&(tps->rto_timer));

            if(timer_pending(&(tps->tw_timer)))
                del_timer_sync(&(tps->tw_timer));
```

```

          if(timer_pending(&(tps->probe_timer)))
             del_timer_sync(&(tps->probe_timer));

          tps = sfi_curr->rhs_tcp_state;
          if(timer_pending(&(tps->rto_timer)))
             del_timer_sync(&(tps->rto_timer));

          if(timer_pending(&(tps->tw_timer)))
             del_timer_sync(&(tps->tw_timer));

          if(timer_pending(&(tps->probe_timer)))
             del_timer_sync(&(tps->probe_timer));

          sfi_prev = sfi_curr;
          sfi_curr = sfi_curr->next;
          kfree(sfi_prev);
       }
       kfree(sfi_curr);
    }
}

MODULE_LICENSE("GPL");

EXPORT_SYMBOL(init_skbuff_list);
EXPORT_SYMBOL(enqueue_skbuff_list);
EXPORT_SYMBOL(insert_skbuff_list);
EXPORT_SYMBOL(dequeue_skbuff_list);
EXPORT_SYMBOL(free_head_skbuff_list);
EXPORT_SYMBOL(head_peek_skb_list);
EXPORT_SYMBOL(get_queue_pkt_count);
EXPORT_SYMBOL(init_head_sfi);
EXPORT_SYMBOL(get_head_sfi);
EXPORT_SYMBOL(get_in_flow);
EXPORT_SYMBOL(get_rep_in_flow);
EXPORT_SYMBOL(enqueue_sfi);
EXPORT_SYMBOL(dequeue_sfi);
EXPORT_SYMBOL(delete_sfi);
EXPORT_SYMBOL(search_sfi);
EXPORT_SYMBOL(cleanup_sfi_list);

/*
 *  This code below was written to implement our own ECN
 *  marking software. It still needs to be tested to be
 *  integrated with the working code of Split TCP
 */

/******************************************************/
/*         Function for RED Qdisc operation           */
/******************************************************/
/* Update: 04/20/07 Temporarily not used. Relying on
 * kernel support for marking */


//int packet_count;

/*
 * This function marks the packet ECN style
 */
/*
static int split_red_ecn_mark(struct sk_buff *skb)
{
   if (skb->nh.raw + 20 > skb->tail)
      return 0;

   switch (skb->protocol) {
      case __constant_htons(ETH_P_IP):
         if (!INET_ECN_is_capable(skb->nh.iph->tos)) {
            printk(KERN_ALERT "Flow not ECN capable \n");
            return 0;
         }

         if(INET_ECN_is_not_ect(skb->nh.iph->tos))
            IP_ECN_set_ce(skb->nh.iph);
            return 1;
      case __constant_htons(ETH_P_IPV6):
         if (!INET_ECN_is_capable(ipv6_get_dsfield(skb->nh.ipv6h)))
            return 0;
         IP6_ECN_set_ce(skb->nh.ipv6h);
         return 1;
     default:
         return 0;
   }
}*/

/*
 * This function contains the main alogrithm of RED
 * Enqueue packet with or without mark or drop the packet
 */
//int
```

```
//split_red_enqueue(struct sk_buff *skb, struct Qdisc *sch)
//{
//   struct split_red_sched_data *q =
//                  (struct split_red_sched_data *)sch->data;
//   struct split_red_sched_data *q = qdisc_priv(sch);
///  psched_time_t now;


     /* Processing for queue idle time */
/*   if(!PSCHED_IS_PASTPERFECT(q->qidlestart)) {
        long us_idle;
        int shift;

        PSCHED_GET_TIME(now);
        us_idle = PSCHED_TDIFF_SAFE(now, q->qidlestart,
                                  q->Scell_max, 0);
        PSCHED_SET_PASTPERFECT(q->qidlestart);
*/
        /* Do avg = (1-Wq)^m * avg here */
/*      index = ;
        shift = q->Stab[index];
        if(shift)
           q->qave <<= shift;
     }
*/
     /* Queue is not empty */
//   else {
//      q->qave +=
//           (sch->qstats.backlog - q->qave) >> q->Wlog;
//   }

     /* Marking decision */
//   if(q->qave <= q->qth_min) {
        /* Enqueue the packet */
//      q->qcount = -1;
//enqueue:
//      if(sch->qstats.backlog + skb->len <= q->limit) {
//         __skb_queue_tail(&sch->q, skb);
//         sch->qstats.backlog += skb->len;
//         sch->bstats.bytes += skb->len;
//         sch->bstats.packets++;
//         sch->qstats.qlen++;
//         packet_count++;
//         printk(KERN_ALERT "Enqueue blindly \n");
//         return NET_XMIT_SUCCESS;
//      }
//      q->st.pdrop++;
//      kfree_skb(skb);
//      sch->qstats.drops++;
//      return NET_XMIT_DROP;
//   }
//   else if(q->qave > q->qth_min &&
//           q->qave <= q->qth_max) {
        /* Mark the packet with random probability */
//      printk(KERN_ALERT "minth < qave < maxth \n");
//      if(++q->qcount) {
//         if((((q->qave - q->qth_min)>>q->Plog)/
//               (q->qth_max - q->qth_min))*q->qcount < q->qR)
//            goto enqueue;

//         q->qcount = 0;
//         q->qR = net_random()&q->Rmask;
//         sch->qstats.overlimits++;
//         goto mark;
//      }
//      if(q->qcount == 0) {
//         q->qR = net_random()&q->Rmask;
//         sch->qstats.overlimits++;
//      }
//   }
//   else if(q->qave > q->qth_max) {
        /* Mark the packet */
//      q->qcount = -1;
//      sch->qstats.overlimits++;
//mark:
//      if(!(q->flags&TC_RED_ECN) ||
//         !split_red_ecn_mark(skb)) {
//         if(!split_red_ecn_mark(skb)) {
//            q->st.early++;
//            goto drop;
//         }
//         q->st.marked++;
//         goto enqueue;
//      }
//drop:
//   kfree_skb(skb);
//   sch->qstats.overlimits++;
//   return NET_XMIT_CN;
//}
```

```
/*
 * This function requeues the packet
 */
/*
int
split_red_requeue(struct sk_buff *skb, struct Qdisc* sch)
{
   struct split_red_sched_data *q = qdisc_priv(sch);

   PSCHED_SET_PASTPERFECT(q->qidlestart);

   __skb_queue_head(&sch->q, skb);
   sch->qstats.backlog += skb->len;
   return 0;
}
*/

/*
 * This function will dequeue and get a packet ready for
 * transmission
 */
/*
struct sk_buff* split_red_dequeue(struct Qdisc* sch)
{
   struct sk_buff *skb;
   struct split_red_sched_data *q = qdisc_priv(sch);

   skb = __skb_dequeue(&sch->q);


   if(skb) {
//    if(skb->h.th != NULL) {
//       if(skb->len < 1000 || skb->h.th->fin || skb->h.th->syn ||
//          sch->qstats.packets < 10 || sch->stats.qlen > 10) {
send_packet:
            packet_count--;
            sch->qstats.qlen--;
            sch->qstats.backlog -= skb->len;
            return skb;
//       }
//    }
//    else
//       goto send_packet;
   }

   PSCHED_GET_TIME(q->qidlestart);
   return NULL;
}
*/

/*
 * This function drops the packet
 */
/*
unsigned int split_red_drop(struct Qdisc* sch)
{
   struct sk_buff *skb;
   struct split_red_sched_data *q = qdisc_priv(sch);

   skb = __skb_dequeue_tail(&sch->q);
   if (skb) {
      unsigned int len = skb->len;
      sch->qstats.backlog -= len;
      sch->qstats.drops++;
      q->st.other++;
      kfree_skb(skb);
      packet_count--;
      sch->qstats.qlen--;
      return len;
   }
   PSCHED_GET_TIME(q->qidlestart);
   return 0;
}

void split_red_reset(struct Qdisc* sch)
{
   struct split_red_sched_data *q = qdisc_priv(sch);

   __skb_queue_purge(&sch->q);
   sch->qstats.backlog = 0;
   PSCHED_SET_PASTPERFECT(q->qidlestart);
   q->qave = 0;
   q->qcount = -1;
}
*/

/*
 * This function sets the parameters used by RED
 */
/*
```

```
int split_red_change(struct Qdisc *sch)
{
   struct split_red_sched_data *q = qdisc_priv(sch);

   packet_count = 0;

   sch_tree_lock(sch);
   q->flags = 0;
   q->Wlog = 9;
   q->Plog = 5;
   q->Rmask = q->Plog < 32 ? ((1<<q->Plog) - 1) : ~0UL;
// q->Scell_log = ctl->Scell_log;
// q->Scell_max = (255<<q->Scell_log);
   q->qth_min = 5000;
   q->qth_max = 15000;
   q->limit = 75000;
   q->qcount = -1;
   q->qave = 0;
   sch->qstats.backlog = 0;
   sch->bstats.packets = 0;
   sch->qstats.qlen = 0;

   if (skb_queue_len(&sch->q) == 0)
      PSCHED_SET_PASTPERFECT(q->qidlestart);

   sch_tree_unlock(sch);
   return 0;
}

int split_red_init(struct Qdisc* sch, struct rtattr *opt)
{
   return split_red_change(sch);
}

int split_red_copy_xstats(struct sk_buff *skb,
                          struct tc_red_xstats *st)
{
   RTA_PUT(skb, TCA_XSTATS, sizeof(*st), st);
   return 0;

rtattr_failure:
   return 1;
}

int split_red_dump(struct Qdisc *sch, struct sk_buff *skb)
```

```
{
   struct split_red_sched_data *q = qdisc_priv(sch);
   unsigned char    *b = skb->tail;
   struct rtattr *rta;
   struct tc_red_qopt opt;

   rta = (struct rtattr*)b;
   RTA_PUT(skb, TCA_OPTIONS, 0, NULL);
   opt.limit = q->limit;
   opt.qth_min = q->qth_min>>q->Wlog;
   opt.qth_max = q->qth_max>>q->Wlog;
   opt.Wlog = q->Wlog;
   opt.Plog = q->Plog;
   opt.Scell_log = q->Scell_log;
   opt.flags = q->flags;
   RTA_PUT(skb, TCA_RED_PARMS, sizeof(opt), &opt);
   rta->rta_len = skb->tail - b;

   if (split_red_copy_xstats(skb, &q->st))
      goto rtattr_failure;

   return skb->len;

rtattr_failure:
   skb_trim(skb, b - skb->data);
   return -1;
}

void split_red_destroy(struct Qdisc *sch)
{
}

static int __init split_red_module_init(void)
{
   return register_qdisc(&split_red_qdisc_ops);
}
static void __exit split_red_module_exit(void)
{
   unregister_qdisc(&split_red_qdisc_ops);
}

module_init(split_red_module_init)
module_exit(split_red_module_exit)
MODULE_LICENSE("GPL");

```

```
EXPORT_SYMBOL(split_red_enqueue);
EXPORT_SYMBOL(split_red_dequeue);
EXPORT_SYMBOL(split_red_requeue);
EXPORT_SYMBOL(split_red_drop);
EXPORT_SYMBOL(split_red_init);
EXPORT_SYMBOL(split_red_reset);
EXPORT_SYMBOL(split_red_destroy);
EXPORT_SYMBOL(split_red_change);
EXPORT_SYMBOL(split_red_dump);
EXPORT_SYMBOL(split_red_qdisc_ops);
*/
```

C.3 ip_in_intercept.c

This file contains the main packet processing code and is implemented as a kernel module.

```
/*
 *
 * Author:      Rahul Jain
 *
 * Filename:    ip_in_intercept.c
 *
 * Comment:
 *      04/13: Changed param struct tcphdr to struct
 *             sk_buff in process_tcp_ack
 *      11/15: Added code for proc file access. The
 *             proc file will contain the table for next
 *             and previous HB IP addr.
 */

#define __KERNEL__
#define MODULE

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/moduleparam.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/if_ether.h>
```

```
#include <linux/split_helper.h>
#include <linux/string.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#include <net/tcp.h>

/*
 * The netfilter hook variable that will be used to
 * register this hook
 */
static struct nf_hook_ops nf_ip_in;

/*
 * Global Variables
 */
#define DEST_IP_MASK 0xffffff
#define SRC_IP_MASK  0xffffff
#define DEST_IP_POOL 0xa0a000
#define SRC_IP_POOL  0x10a000
#define DELAY        30
#define MAX_SSTHRESH 0xffff
#define IPIP_FRAG_OP 0x8000

struct probe_info {
        struct split_flow_info *sfi;
        int flag;
        struct net_device *in_dev;
};

struct probe_info *pinfo;

/*
 * Command line arguements
 */
static __u32 neigh_hb_table[16] = {-1,-1,-1,-1,
                                   -1,-1,-1,-1,
                                   -1,-1,-1,-1,
                                   -1,-1,-1,-1};
static int tbl_cnt;

module_param_array(neigh_hb_table, int, &tbl_cnt, 0444);

MODULE_PARM_DESC(neigh_hb_table,
                 "Array containing next HB IP address");

```

```
/*
 * This array contains the network pools and hb addr
 * Format is :
 *      src_ip dst_ip nhb_ip phb_ip
 * Network layout to understand next and previous HB:
 *      S --- PHB --- ME --- NHB --- D
 */
__u32 nw_hb_table[NW_LIMIT][4];

static unsigned long curr_proc_buff_len = 0;
int NW_TABLE_POPULATED = 0;

struct split_flow_info *slh;
struct timer_list send_time;
int send_time_first = 1;
int rto_timer_expired = 1;

int prepare_fwd_data(unsigned long data, int flag);
void prepare_fwd_retrans_data(unsigned long data);
void remove_sfi_node(unsigned long data);
void prepare_tcp_probe(unsigned long data);
static void tcp_grow_window(struct split_flow_info *sfi,
                            struct tcp_state *tp,
                            struct sk_buff *skb);

/*********************************************************/
/*      Neighbour table processing Functions            */
/*********************************************************/
int populate_nw_table()
{
   int i, j;
   char **table_dup;
   char *table_entry;
   __u32 utemp;

   for(i = 0; i < 16; i++)
      printk(KERN_ALERT "neigh_hb_table: %x \n",
                        neigh_hb_table[i]);

   for(i = 0; i < NW_LIMIT; i++) {
      for(j = 0; j < 4; j++) {
         nw_hb_table[i][j] = neigh_hb_table[(4*i+j)];
      }
   }

   for(i = 0; i < NW_LIMIT; i++) {
      for(j = 0; j < 4; j++) {
         printk(KERN_ALERT "nw_hb_table: %x \n",
                                 nw_hb_table[i][j]);
      }
   }

   if(nw_hb_table != NULL) {
      return 1;
   }
   else
      return 0;
}

/*
 * This function returns the next HB addr. It requires
 * 1. Flow details
 * 2. Direction of flow
 */
__u32 get_nexthb_addr(struct flow_detail *fd, int flag)
{
   __u32 hb_addr = -1;
   __u32 src_addr, dest_addr;
   int i, tbl_index;

   if(flag == PREV_FLOW)
      tbl_index = 3;
   else if(flag == FWD_FLOW)
      tbl_index = 2;

   src_addr = fd->saddr << 12;
   dest_addr = fd->daddr << 12;

   for(i = 0; i < NW_LIMIT; i++) {
      if((src_addr & SRC_IP_MASK) == nw_hb_table[i][0] &&
         (dest_addr & DEST_IP_MASK) == nw_hb_table[i][1])
            hb_addr = nw_hb_table[i][tbl_index];
   }

   return hb_addr;
}

/*
 * This function is used to print the HW addr
 */
```

```
void uchar_to_hex(unsigned char addr[])
{
   int i;
   for(i = 0; i < ETH_ALEN; i++) {
      printk(KERN_ALERT "%x.", addr[i]);
   }
   printk(KERN_ALERT "\n");
   return;
}

/*****************************************************/
/*                Timer Related Functions            */
/*****************************************************/
/*
 * This function initializes the snd_timer.
 * add_delay is used when the user wants a delay of more
 * than 10msec
 */
void
init_send_timer(struct split_flow_info *sfi, int add_delay)
{
        init_timer(&send_time);
        send_time.expires = jiffies + DELAY + add_delay + 10;
        send_time.data = (unsigned long *)sfi;
        send_time.function = prepare_fwd_data;
        add_timer(&send_time);
        send_time_first = -1;
        return;
}

/*
 * This function initializes the rto_timer.
 */
void init_rto_timer(struct skbuff_list *queue_head,
                    struct tcp_state *tp, int mult_factor)
{
   init_timer(&(tp->rto_timer));
   tp->rto_timer.expires = jiffies + tp->rto * mult_factor;

   /* Setting the upper bound of 60 secs on the RTO */
   if(time_diff(tp->rto_timer.expires, jiffies) > TCP_RTO_MAX)
      tp->rto_timer.expires = jiffies + TCP_RTO_MAX;

   tp->rto_timer.data = (unsigned long *)queue_head;
   tp->rto_timer.function = prepare_fwd_retrans_data;
   add_timer(&(tp->rto_timer));

   return;
}

/*
 * This function initializes the time wait timer.
 */
void init_tw_timer(struct split_flow_info *sfi, int flag)
{
   struct tcp_state *tps, *tps_sibling;

   printk(KERN_ALERT "Inside init_tw_timer()....\n");

   if(flag == PREV_FLOW) {
      tps = sfi->lhs_tcp_state;
      tps_sibling = sfi->rhs_tcp_state;
   }
   else if(flag == FWD_FLOW) {
      tps = sfi->rhs_tcp_state;
      tps_sibling = sfi->lhs_tcp_state;
   }

   init_timer(&(tps->tw_timer));
   tps->tw_timer.expires = jiffies + (TCP_MSL<<1);
   tps->tw_timer.data = (unsigned long *)sfi;
   tps->tw_timer.function = remove_sfi_node;
   add_timer(&(tps->tw_timer));
   printk(KERN_ALERT "Exiting init_tw_timer()....\n");
   return;
}

/*
 * This function initializes the 0 window probe timer.
 */
void init_probe_timer(struct probe_info *pi,
                      struct tcp_state *tp,
                      int mult_factor)
{
   if(!timer_pending(&(tp->probe_timer)))
      del_timer_sync(&(tp->probe_timer));

   tp->probe_timer.expires = tp->rto*mult_factor+jiffies;

   if((tp->probe_timer.expires - jiffies) > TCP_RTO_MAX)
```

```
        tp->probe_timer.expires = jiffies + TCP_RTO_MAX;

    tp->probe_timer.data = (unsigned long *)pi;
    tp->probe_timer.function = prepare_tcp_probe;
    add_timer(&(tp->probe_timer));
    return;
}

/*****************************************************/
/*        General Split TCP Related Functions        */
/*****************************************************/
/*
 * This function returns 1 if the flow is supported for Split TCP
 * processing. It compares the source and destination addr. with
 * those in nw_hb_table. ipip_flag = 1 if the function is called
 * while removing the outer IP hdr else 0.
 */
int flow_supported(__u32 src_addr, __u32 dest_addr, int ipip_flag)
{
    __u32 src_netid, dest_netid;
    int src_net_present, dst_net_present, ret_value;
    int i, j;

    src_net_present = dst_net_present = ret_value = 0;

    src_netid = src_addr << 12;
    dest_netid = dest_addr << 12;

    if(nw_hb_table == NULL)
        return ret_value;

    /* Checking if flow is supported */
    for(i = 0; i < NW_LIMIT; i++) {
        if((src_netid & SRC_IP_MASK) == nw_hb_table[i][0])
            src_net_present = 1;
        if((dest_netid & DEST_IP_MASK) == nw_hb_table[i][1])
            dst_net_present = 1;

        if(ipip_flag)
            ret_value = (src_net_present || dst_net_present);
        else
            ret_value = (src_net_present && dst_net_present);

        if(ret_value == 1)
            break;
    }
    return(ret_value);
}

/*
 * This function removes the sfi node from the list.
 * It is called only after the TIME_WAIT period has elapsed. Hence
 * we can safely delete the node.
 */
void remove_sfi_node(unsigned long data)
{
    struct split_flow_info *sfi = (struct split_flow_info *)data;
    struct tcp_state *rhs_tps, *lhs_tps;

    rhs_tps = sfi->rhs_tcp_state;
    lhs_tps = sfi->lhs_tcp_state;

    if(timer_pending(&(rhs_tps->tw_timer)) ||
       timer_pending(&(lhs_tps->tw_timer)) )
       return;
    else {
       del_timer_sync(&(rhs_tps->tw_timer));
       del_timer_sync(&(lhs_tps->tw_timer));
       delete_sfi(sfi);
    }
    return;
}

/*
 * This function extracts the hardware address of the incoming packet
 * and stores it in the sfi node
 */
void get_hw_addr(struct sk_buff *skb, struct split_flow_info *sfi,
                 int flag)
{
    struct ethhdr *mac;
    mac = (struct ethhdr *)skb->mac.raw;

    if(flag == PREV_FLOW)
       memcpy(sfi->prev_flow_hw, mac , sizeof(struct ethhdr));
    else if(flag == FWD_FLOW)
       memcpy(sfi->fwd_flow_hw, mac, sizeof(struct ethhdr));

    return;
}
```

```

/*
 * This function extracts the IP header of the original data packet
 * from the IPIP packet
 */
struct iphdr* get_ip_header(struct sk_buff *skb)
{
   return((struct iphdr*)(skb->data + skb->nh.iph->ihl*4));
}

/*
 * This function extracts the TCP header of the original data packet
 * from the IPIP packet
 */
struct tcphdr* get_tcp_header(struct sk_buff *skb)
{
   struct iphdr *iph;
   iph = (struct iphdr*)(skb->data + skb->nh.iph->ihl*4);
   return((struct tcphdr*)
            (skb->data + skb->nh.iph->ihl*4 + iph->ihl*4));
}

/*
 * This function creates a new node for the incoming flow in the
 * linked list
 */
struct split_flow_info*
create_sfinode(struct split_flow_info *head, struct sk_buff *skb)
{
   struct split_flow_info *newsfi;

   /* Enqueue a new node in the linked list */
   newsfi = enqueue_sfi(slh);

   newsfi->prev_flow_hw = (struct ethhdr *)
                            kmalloc(sizeof(struct ethhdr), GFP_ATOMIC);
   newsfi->fwd_flow_hw = (struct ethhdr *)
                           kmalloc(sizeof(struct ethhdr), GFP_ATOMIC);

   /* Initialize the skbuff_list for this node */
   init_skbuff_list(newsfi);

   /* Initialize all the timers for the flow */
   init_timer(&(newsfi->lhs_tcp_state->tw_timer));
   init_timer(&(newsfi->rhs_tcp_state->tw_timer));
```

```
    init_timer(&(newsfi->lhs_tcp_state->rto_timer));
    init_timer(&(newsfi->rhs_tcp_state->rto_timer));
    init_timer(&(newsfi->lhs_tcp_state->probe_timer));
    init_timer(&(newsfi->rhs_tcp_state->probe_timer));

    pinfo = (struct probe_info*)
            kmalloc(sizeof(struct probe_info), GFP_ATOMIC);

    return newsfi;
}

/*
 * This function replicates qdisc_create_dflt() of the kernel
 * Update: 04/20/07 Temporarily not used. Relying on kernel RED
 * for marking support
 */
/*
struct Qdisc* assign_qdisc_ops(struct net_device *dev,
                               struct Qdisc_ops *ops,
                               struct Qdisc *sch)
{
//   int size = sizeof(*sch) + ops->priv_size;

//   sch = kmalloc((sizeof(struct Qdisc) + ops->priv_size),
//                  GFP_KERNEL);
//   if (!sch)
//      return NULL;
//   memset(sch, 0, size);


     skb_queue_head_init(&sch->q);
     sch->ops = ops;
     sch->enqueue = ops->enqueue;
     sch->dequeue = ops->dequeue;
     sch->dev = dev;
     sch->stats_lock = &dev->queue_lock;
     sch->refcnt.counter =  1;
     if (!ops->init || ops->init(sch, NULL) == 0)
        return sch;

     kfree(sch);
     return NULL;
}
*/

```

```
/*
 * This function will change the Qdisc for the given interface on
 * the fly. Used to change the Qdisc of an interface to RED when
 * ECN can be used.
 * Update: 04/20/07 Temporarily not used. Relying on kernel RED
 * for marking support
 */
/*
void change_qdisc(struct net_device *dev, struct Qdisc *sch)
{
   struct Qdisc *q;

   if(dev->qdisc_sleeping->ops != &split_red_qdisc_ops) {
      printk(KERN_ALERT "Changing Qdisc for %s \n", dev->name);

      spin_lock_bh(&dev->queue_lock);
      q = assign_qdisc_ops(dev, &split_red_qdisc_ops, sch);
      spin_unlock_bh(&dev->queue_lock);

      if(q == NULL) {
         printk(KERN_ALERT "Qdisc change failed in SplitTCP \n");
         return;
      }
      printk(KERN_ALERT "Qdisc registeration successful \n");

      INIT_LIST_HEAD(&q->list);
      list_add_tail(&q->list, &dev->qdisc_list);
      dev->qdisc_sleeping = q;

      spin_lock_bh(&dev->queue_lock);
      dev->qdisc = dev->qdisc_sleeping;
      dev->trans_start = jiffies;
      __netdev_watchdog_up(dev);
      spin_unlock_bh(&dev->queue_lock);
   }
}
*/

/*******************************************************/
/*                Queue Management Functions           */
/*******************************************************/
/*
 * This function is responsible for finding new holes in the queue.
 * If a new hole is found, the following tcp_state and skbuff_list
 * variable are updated
 * 1. rcv_next
 * 2. pkt_bfr_hole
 * The function returns 1 if a new hole is found, else returns -1
 * Change (02/23/2006).
 *      - Introduced 4 variables to store the correct tcp and ip
 *        headers in case of IPIP
 *      - Changed all ref of type
 *              {curr/next}_pkt->nh.iph-> to {curr/next}_pkt_iph->
 *      - Similar for tcp.
 */
int find_new_hole_update(struct skbuff_list *queue_head,
                          struct tcp_state *tps)
{
   __u32 curr_seq, next_seq;
   struct skbuff_list *curr_skbl = queue_head->pkt_bfr_hole;
   struct skbuff_list *next_skbl;
   struct sk_buff *curr_pkt, *next_pkt;
   struct iphdr *curr_pkt_iph, *next_pkt_iph;
   struct tcphdr *curr_pkt_th, *next_pkt_th;
   int data_bytes;

   if(curr_skbl == NULL)
      return -1;

   if(curr_skbl == queue_head) {
      printk(KERN_ALERT
            "pkt_bfr_hole is pointing to the head of the queue\n");
   }

   next_skbl = curr_skbl->next;

   while(next_skbl != queue_head) {
      curr_pkt = curr_skbl->sb_pkt;
      next_pkt = next_skbl->sb_pkt;

      if(curr_pkt != NULL && next_pkt != NULL) {
         /* Get proper TCP and IP header */
         if(curr_pkt->nh.iph->protocol == IPPROTO_IPIP) {
            curr_pkt_iph = get_ip_header(curr_pkt);
            curr_pkt_th = get_tcp_header(curr_pkt);
         }
         else {
            curr_pkt_iph = curr_pkt->nh.iph;
            curr_pkt_th = curr_pkt->h.th;
         }
```

```

            if(next_pkt->nh.iph->protocol == IPPROTO_IPIP) {
               next_pkt_iph = get_ip_header(next_pkt);
               next_pkt_th = get_tcp_header(next_pkt);
            }
            else {
               next_pkt_iph = next_pkt->nh.iph;
               next_pkt_th = next_pkt->h.th;
            }

            curr_seq = ntohl(curr_pkt_th->seq);
            next_seq = ntohl(next_pkt_th->seq);

            data_bytes = ntohs(curr_pkt_iph->tot_len) -
                         (curr_pkt_iph->ihl*4) - (curr_pkt_th->doff*4);

            //TODO: checking seq properly
            if(next_seq > (curr_seq + data_bytes + (curr_pkt_th->syn ||
               curr_pkt_th->fin))) {
               queue_head->pkt_bfr_hole = curr_skbl;
               tps->rcv_next = curr_seq + ntohs(curr_pkt_iph->tot_len) -
                             (curr_pkt_iph->ihl*4)-(curr_pkt_th->doff*4);
               return 1;
            }
         }
         curr_skbl = next_skbl;
         next_skbl = curr_skbl->next;
      }

      queue_head->pkt_bfr_hole = NULL;

      return -1;
   }

   /*
    * This function is called when we recieve a packet that fills a hole
    * in the queue. If the packet does not fit anywhere, it is silently
    * dropped.
    * Change (02/23/2006).
    *       - Introduced 4 variables to store the correct tcp and ip
    *         headers in case of IPIP
    *       - Changed all ref of type
    *               {curr/next}_pkt->nh.iph-> to {curr/next}_pkt_iph->
    *       - Similar for tcp.
    */
```

```
void find_hole_and_enqueue(struct skbuff_list *queue_head,
                           struct sk_buff *skb)
{
   __u32 curr_seq, next_seq, skb_seq;
   struct skbuff_list *curr_skbl = queue_head->pkt_bfr_hole;
   struct skbuff_list *skbl_node = NULL;
   struct skbuff_list *next_skbl = NULL;
   struct sk_buff *curr_pkt = NULL;
   struct sk_buff *next_pkt = NULL;
   struct iphdr *curr_pkt_iph, *next_pkt_iph;
   struct tcphdr *curr_pkt_th, *next_pkt_th;
   int data_bytes;

   if(curr_skbl == NULL) {
      return;
   }

   skb_seq = ntohl(skb->h.th->seq);
   next_skbl = curr_skbl->next;

   while(curr_skbl != queue_head) {
      curr_pkt = curr_skbl->sb_pkt;

      /* Get proper tcp and ip heaer */
      if(curr_pkt->nh.iph->protocol == IPPROTO_IPIP) {
         curr_pkt_iph = get_ip_header(curr_pkt);
         curr_pkt_th = get_tcp_header(curr_pkt);
      }
      else {
         curr_pkt_iph = curr_pkt->nh.iph;
         curr_pkt_th = curr_pkt->h.th;
      }

      curr_seq = ntohl(curr_pkt_th->seq);
      data_bytes = ntohs(curr_pkt_iph->tot_len) -
                   (curr_pkt_iph->ihl*4)-(curr_pkt_th->doff*4);

      //TODO: checking seq properly
      if((curr_seq + data_bytes) <= skb_seq) {
         if(next_skbl != queue_head) {
            next_pkt = next_skbl->sb_pkt;

            /* Get proper tcp and ip heaer */
            if(next_pkt->nh.iph->protocol == IPPROTO_IPIP) {
               next_pkt_iph = get_ip_header(next_pkt);
```

```
                next_pkt_th = get_tcp_header(next_pkt);
             }
             else {
                next_pkt_iph = next_pkt->nh.iph;
                next_pkt_th = next_pkt->h.th;
             }

             next_seq = ntohl(next_pkt_th->seq);
             if(skb_seq < next_seq) {
                skbl_node = (struct skbuff_list *)
                            kmalloc(sizeof(struct skbuff_list),
                                    GFP_ATOMIC);
                insert_skbuff_list(curr_skbl, skbl_node, queue_head);
                skbl_node->sb_pkt = skb_copy(skb, GFP_ATOMIC);
                return;
             }
          }
          else {
             skbl_node = (struct skbuff_list *)
                         kmalloc(sizeof(struct skbuff_list),
                                 GFP_ATOMIC);
             insert_skbuff_list(curr_skbl, skbl_node, queue_head);
             skbl_node->sb_pkt = skb_copy(skb, GFP_ATOMIC);
             return;
          }
       }
       curr_skbl = next_skbl;
       next_skbl = curr_skbl->next;
    }

    return;
}

/*
 * This function will enqueue a sk_buff into the appropriate queue
 */
int
enqueue_packet(struct split_flow_info *sfi, struct sk_buff *skb,
               struct skbuff_list *queue_head, struct tcp_state *tps)
{
    struct skbuff_list *skbl_node;
    struct sk_buff *prev_pkt = queue_head->prev->sb_pkt;
    struct iphdr *prev_pkt_iph;
    struct tcphdr *prev_pkt_tcph;
    int data_bytes;
    int hole_present, ret_value = -1;
    __u32 prev_pkt_seq;
    __u32 curr_pkt_seq;

    curr_pkt_seq = ntohl(skb->h.th->seq);
    if(prev_pkt != NULL) {
       if(prev_pkt->nh.iph->protocol == IPPROTO_IPIP) {
          prev_pkt_iph = get_ip_header(prev_pkt);
          prev_pkt_tcph = get_tcp_header(prev_pkt);
       }
       else {
          prev_pkt_iph = prev_pkt->nh.iph;
          prev_pkt_tcph = prev_pkt->h.th;
       }
       prev_pkt_seq = ntohl(prev_pkt_tcph->seq);
       data_bytes = ntohs(prev_pkt_iph->tot_len)-(prev_pkt_iph->ihl*4)
                                                - (prev_pkt_tcph->doff*4);
    }

    hole_present = queue_head->hole_in_queue;

    /* The first if statement will not be needed once the SYN 3-way
     * handshake is done end to end
     */
    /* Copy in queue if packet contains some data or is a FIN */
    if((tps->end_seq - curr_pkt_seq) > 0 ||
        skb->h.th->fin || skb->h.th->syn) {
       /* If queue is empty, simple enqueue the packet */
       if(get_queue_pkt_count(queue_head) == 0 ) {
          if(skb->h.th->syn || curr_pkt_seq == tps->rcv_next) {
             ret_value = 1;
             goto enqueue_pkt;
          }
          else if(curr_pkt_seq > tps->rcv_next)
             goto pkt_crt_hole;
       }

       /* Case 1: There are no holes in the queue */
       if(!(hole_present)) {
          /* Packet is in order */
          if((prev_pkt_seq + prev_pkt_tcph->syn + data_bytes) ==
                                                     curr_pkt_seq) {
             ret_value = 1;
             goto enqueue_pkt;
          }
```

```
         /* Packet going to create a hole */
         else if(curr_pkt_seq >
                 (prev_pkt_seq + prev_pkt_tcph->syn + data_bytes)) {
pkt_crt_hole:
            hole_present = 1;
            ret_value = -1;
            queue_head->pkt_bfr_hole = queue_head->prev;
            tps->max_rcv_byte = tps->end_seq;
            goto enqueue_pkt;
         }
      }
      /* Case 2: There are hole(s) in the queue */
      else if(hole_present) {
         /* We got the lost packet */
         if(curr_pkt_seq == tps->rcv_next) {
            /* This is the missing packet, enqueue and update */
            skbl_node = (struct skbuff_list *)
                        kmalloc(sizeof(struct skbuff_list), GFP_ATOMIC);
            skbl_node->next = skbl_node->prev = skbl_node;

            write_lock(&queue_head->lock);
            insert_skbuff_list(queue_head->pkt_bfr_hole, skbl_node,
                               queue_head);
            write_unlock(&queue_head->lock);

            skbl_node->sb_pkt = skb_copy(skb, GFP_ATOMIC);

            /* Check if more holes present */
            if(find_new_hole_update(queue_head, tps) == 1) {
//             printk(KERN_ALERT "Additional holes \n");
               hole_present = 1;
               ret_value = -1;
               /* rcv_next updated by find_new_hole_update() */
            }
            else {
               hole_present = 0;
               ret_value = -1;
               tps->rcv_next = tps->max_rcv_byte;
               /* rcv_next will be updated by tcp_process_ack() */
            }
            goto update_and_ret;
         }
         /* Packet in sequence */
         else if(tps->end_seq >= tps->max_rcv_byte) {
            tps->max_rcv_byte = tps->end_seq;
            ret_value = -1;
            goto enqueue_pkt;
         }
         /* Lost packet filling another hole */
         else {
            find_hole_and_enqueue(queue_head, skb);
            ret_value = -1;
            goto update_and_ret;
         }
      }

enqueue_pkt:
      /* Charge the buffer and grow the window */
      if(!skb->h.th->syn) {
         data_bytes = ntohs(skb->nh.iph->tot_len)-skb->nh.iph->ihl*4
                                                - skb->h.th->doff*4;

         tcp_charge_buffer(sfi, data_bytes, tps);

         if(data_bytes > 128)
            tcp_grow_window(sfi, tps, skb);
      }

      skbl_node = (struct skbuff_list *)
                    kmalloc(sizeof(struct skbuff_list), GFP_ATOMIC);

      write_lock(&queue_head->lock);
      enqueue_skbuff_list(sfi, skbl_node, queue_head);
      write_unlock(&queue_head->lock);

      skbl_node->sb_pkt = skb_copy(skb, GFP_ATOMIC);
   }

update_and_ret:
   queue_head->hole_in_queue = hole_present;
   return(ret_value);
}

/*
 * This function adds the MAC header and puts it in the device queue
 */
void add_eth_hdr(struct split_flow_info *sfi,
                  struct sk_buff *skb, int flag)
{
   struct ethhdr *eth_out = (struct ethhdr *)skb_push(skb, ETH_HLEN);
```

```
    struct ethhdr *eth_in = NULL;
    struct ethhdr *dump;

    if(flag == PREV_FLOW) {
       eth_in = sfi->prev_flow_hw;
    }
    else if(flag == FWD_FLOW) {
       eth_in = sfi->fwd_flow_hw;
    }

    /* Populate the MAC header */
    eth_out->h_proto = eth_in->h_proto;
    memcpy(eth_out->h_dest, eth_in->h_source, ETH_ALEN);
    memcpy(eth_out->h_source, eth_in->h_dest, ETH_ALEN);

    /* Put it on the wire now*/
    dev_queue_xmit(skb);
    return;
}

/*
 * This function adds the IP header.
 */
void add_ip_and_send(struct split_flow_info *sfi,
                     struct sk_buff *skb, int flag)
{
    struct iphdr *iph;
    struct flow_detail *out_flow = NULL;
    struct tcp_state *tp = NULL;

    if(flag == PREV_FLOW) {
       tp = sfi->lhs_tcp_state;
       out_flow = sfi->i2r_flow;
    }
    else if(flag == FWD_FLOW) {
       tp = sfi->rhs_tcp_state;
       out_flow = sfi->r2i_flow;
    }

    /* Adding IP hdr at the start */
    iph = (struct iphdr *)skb_push(skb, sizeof(struct iphdr));

    /* Filling the fields with values */
    iph->version = 4;
    iph->ihl = 5;
```

```
   if(tp->ecn_capable)
      iph->tos = 0x02;
   else
      iph->tos = 0;
   iph->tot_len = htons(skb->len);
   iph->id = ++tp->ip_id;
   iph->frag_off = htons(IP_DF);
   iph->ttl = IPDEFTTL;
   iph->protocol = IPPROTO_TCP;
   iph->saddr = out_flow->daddr;
   iph->daddr = out_flow->saddr;

   skb->nh.iph = iph;
   ip_send_check(iph);

   /* Done with IP part. Build MAC header now */
   add_eth_hdr(sfi, skb, flag);

   return;
}

/*****************************************************/
/*              RTO Calculation Functions            */
/*****************************************************/
/* This function calculates the values for srtt and rttvar.
 * The rto variable is _not_ set in this function.
 * The code was copied from the kernel function tcp_rtt_estimator()
 * The next 3 function deal with the RTT measurement. They should
 * always be called in the order they are defined in.
 * All these fns are copied from the kernel. The reason I am not
 * calling the corresponding kernel function is because I am making
 * use of my own structure for tcp_state.
 */
void tcp_rtt_estimate(struct tcp_state *tp)
{
   long m;

   m = tp->ack_seq_tstamp - tp->rtt_seq_tstamp;

   /* The following code was copied from the kernel function
    * tcp_rtt_estimator() */
   if(m == 0)
      m = 1;
   if (tp->srtt != 0) {
      m -= (tp->srtt >> 3);   /* m is now error in rtt est */
```

```
        tp->srtt += m;          /* rtt = 7/8 rtt + 1/8 new */
        if (m < 0) {
           m = -m;         /* m is now abs(error) */
           m -= (tp->mdev >> 2);   /* similar update on mdev */
           if (m > 0)
              m >>= 3;
        } else {
           m -= (tp->mdev >> 2);   /* similar update on mdev */
        }
        tp->mdev += m;          /* mdev = 3/4 mdev + 1/4 new */
        if (tp->mdev > tp->mdev_max) {
           tp->mdev_max = tp->mdev;
           if (tp->mdev_max > tp->rttvar)
              tp->rttvar = tp->mdev_max;
        }
        if (after(tp->snd_una, tp->rtt_seq)) {
           if (tp->mdev_max < tp->rttvar)
              tp->rttvar -= (tp->rttvar-tp->mdev_max)>>2;
           tp->rtt_seq = tp->snd_next;
           tp->mdev_max = TCP_RTO_MIN;
        }
    }else {
      /* no previous measure. */
      tp->srtt = m<<3;        /* take the measured time to be rtt */
      tp->mdev = m<<1;        /* make sure rto = 3*rtt */
      tp->mdev_max = tp->rttvar = max(tp->mdev, TCP_RTO_MIN);
      tp->rtt_seq = tp->snd_next;
    }

    return;
}

/* This function sets the value of the rto variable
 * Copied from the kernel function with the same name
 */
void tcp_set_rto(struct tcp_state *tp)
{
    tp->rto = (tp->srtt >> 3) + tp->rttvar;
}

/* This function puts an upper bound to the value of rto
 * Copied from the kernel function with the same name
 */
void tcp_bound_rto(struct tcp_state *tp)
{
   if (tp->rto > TCP_RTO_MAX)
      tp->rto = TCP_RTO_MAX;
}

/*****************************************************/
/*       Advertised Window Calculation Functions      */
/*****************************************************/
static int
__tcp_grow_window(struct split_flow_info *sfi,
                  struct tcp_state *tp, struct sk_buff *skb)
{
   /* Optimize this! */
   int truesize = skb->truesize/2;
   int window = (sfi->buff_clamp>>1)/2;

   while (tp->rcv_ssthresh <= window) {
      if (truesize <= skb->len)
         return 2*tp->snd_mss;

      truesize >>= 1;
      window >>= 1;
   }

   return 0;
}

static __inline__ void
tcp_grow_window(struct split_flow_info *sfi, struct tcp_state *tp,
                struct sk_buff *skb)
{
   /* Check #1 */
   /* Remove the space we committed in our last adv wnd */
   int flow_space = (sfi->buff_clamp>>1) - (int)tp->local_rcv_wnd;

   if (tp->rcv_ssthresh < tp->wnd_clamp &&
       (int)tp->rcv_ssthresh < flow_space ) {
      int incr;

      /* Check #2. Increase window, if skb
       * with such overhead
       * will fit to rcvbuf
       * in future.
       */
      if(skb->truesize <= skb->len)
         incr = 2*tp->rcv_mss;
```

```
        else
            incr = __tcp_grow_window(sfi, tp, skb);

        if (incr) {
            tp->rcv_ssthresh = min(tp->rcv_ssthresh + incr,
                                   tp->wnd_clamp);
        }
    }
}

/*
 * This function returns the amount by which we can increase the
 * advertised window.
 * Copied from the kernel function with the same name. Some changes.
 */
u32 __tcp_select_window_(struct split_flow_info *sfi, int flag)
{
    struct tcp_state *tps;
    struct skbuff_list *pri_queue, *sec_queue;

    if(flag == PREV_FLOW) {
        tps = sfi->lhs_tcp_state;
        /* pri_queue points to the rcv_queue of the flow */
        pri_queue = sfi->i2r_queue;
        sec_queue = sfi->r2i_queue;
    }
    else if(flag == FWD_FLOW) {
        tps = sfi->rhs_tcp_state;
        /* pri_queue points to the rcv_queue of the flow */
        pri_queue = sfi->r2i_queue;
        sec_queue = sfi->i2r_queue;
    }

    /* MSS for the peer's data.  Previous verions used mss_clamp
     * here.  I don't know if the value based on our guesses
     * of peer's MSS is better for the performance.  It's more correct
     * but may be worse for the performance because of rcv_mss
     * fluctuations.  --SAW  1998/11/1
     */
    int mss = tps->snd_mss;
    __u32 full_space = sfi->buff_clamp >> 1;
    int window;
    int priq_pkt_cnt = get_queue_pkt_count(pri_queue);
    int secq_pkt_cnt = get_queue_pkt_count(sec_queue);
    int free_space = full_space - priq_pkt_cnt * tps->rcv_mss;
```

```

   if (mss > full_space)
      mss = full_space;

   if (free_space < full_space/2) {
      if (free_space < mss)
         return 0;
   }

   if (free_space > tps->rcv_ssthresh)
      free_space = tps->rcv_ssthresh;

   /* Get the largest window that is a nice multiple of mss.
    * window clamp already applied above.
    * If our current window offering is within 1 mss of the
    * free space we just keep it. This prevents the divide
    * and multiply from happening most of the time.
    * We also don't do any window rounding when the free space
    * is too small.
    */
    window = tps->local_rcv_wnd;
    if (window <= free_space - mss || window > free_space)
       window = (free_space/mss)*mss;

    return window;
}

u32 __tcp_receive_window_(struct tcp_state *tps)
{
   s32 win = tps->rcv_wup + tps->local_rcv_wnd - tps->rcv_next;

   if (win < 0)
      win = 0;
   return (u32) win;
}

/*
 * This function selects a new advertised window that can be directly
 * fed into th->window
 * Copied from the kernel function with the same name. Some changes.
 */
u16 tcp_select_window_(struct split_flow_info *sfi,
                       struct tcp_state *tps, int flag)
{
   u32 cur_win = __tcp_receive_window_(tps);
```

```
    u32 new_win = __tcp_select_window_(sfi, flag);
    int hole_in_q;

    if(flag == PREV_FLOW)
       hole_in_q = sfi->i2r_queue->hole_in_queue;
    else if(flag == FWD_FLOW)
       hole_in_q = sfi->r2i_queue->hole_in_queue;

    /* Never shrink the offered window */
    if(new_win < cur_win) {
       /* Danger Will Robinson!
        * Don't update rcv_wup/rcv_wnd here or else
        * we will not be able to advertise a zero
        * window in time.  --DaveM
        *
        * Relax Will Robinson.
        */
       new_win = cur_win;
    }

    /* Advertise a window of 1 MSS if there is a hole and we are
     * about to advertise a zero window
     */
    if(htons(new_win) <= htons(tps->snd_mss) && hole_in_q) {
       new_win = tps->snd_mss;
    }
    tps->local_rcv_wnd = ntohs(htons(new_win));
    tps->rcv_wup = tps->rcv_next;

    return new_win;
}

/*******************************************************/
/*  Initialization & wnd and buffer Update Functions   */
/*******************************************************/
/*
 * This function initializes the mss and window parameters for a flow
 * pkt_wnd is set manually to 2.
 * Change (05/23/2006): Moved wnd_clamp to ip_in_process and
 * synack function
 */
void tcp_init_mss_wnd(struct tcp_state *tp, __u16 mss, __u16 window)
{
    tp->wnd_curr = ntohs(window);
    tp->wnd_curr_pkt = tp->wnd_curr / tp->mss_clamp;
```

```
   tp->rcv_wnd = ntohs(window);
   tp->rcv_wnd_pkt = tp->rcv_wnd / tp->mss_clamp;

   /* Setting rcv_ssthresh to 4*mss */
   tp->rcv_ssthresh = tp->mss_clamp<<2;

   tp->ssthresh = MAX_SSTHRESH / tp->mss_clamp;
   tp->cwnd = 2;

   tp->pkt_wnd = min(tp->cwnd, tp->rcv_wnd_pkt);
   tp->dup_ack_cnt = 0;

   return;
}

/*
 * This function calculates the window scaling factor that we will
 * advertise
 */
void calc_rcv_wscale(int __space, __u32 mss, __u32 window_clamp,
                     int wscale_ok, __u8 *wscale)
{
   unsigned int space = (__space < 0 ? 0 : __space);

   if(window_clamp == 0)
      window_clamp = (65535 << 14);

   space = min(window_clamp, space);

   (*wscale) = 0;
   if(wscale_ok) {
      while(space > 65535 && (*wscale) < 14) {
         space >>= 1;
         (*wscale)++;
      }
   }
}

/*
 * Write general description
 * This function returns 1 if the new packet can be enqueued. Else -1
 */
int tcp_wnd_may_update(struct split_flow_info *sfi, int flag)
{
   struct tcp_state *tp = NULL;

   if(flag == PREV_FLOW) {
      tp = sfi->lhs_tcp_state;
   }
   else if(flag == FWD_FLOW) {
      tp = sfi->rhs_tcp_state;
   }

   /* Check if window allows us to accept a new packet */
   if(((tp->snd_next - tp->snd_una) <= tp->rcv_wnd) &&
      /* Check if there is space in the buffer */
      ((tp->end_seq - tp->rcv_next) + sfi->buff_curr <= sfi->buff_clamp) &&
      /* Check if good ACK */
       (tp->ack_seq >= tp->snd_una))
         return 1;

   return -1;
}

/*
 * This function frees the buffer being used by a TCP flow off the
 * acked data
 */
void tcp_free_buffer(struct split_flow_info *sfi, int data_bytes,
                     struct tcp_state *tps)
{
   sfi->buff_curr -= data_bytes;
}

/*
 * This function charges the buffer being used by a TCP flow for the
 * new data
 */
void tcp_charge_buffer(struct split_flow_info *sfi, int data_bytes,
                       struct tcp_state *tps)
{
   sfi->buff_curr += data_bytes;
   tps->wnd_curr -= (tps->end_seq - tps->rcv_next);
}

/*
 * This function updates the congestion window
 * For FWD_FLOW, it updates pkt_wnd and removes data bytes from
 * buff_curr.
 * For PREV_FLOW, it increases buff_curr and decreases wnd_curr by
 * data bytes
 */
void tcp_update_cwnd(struct split_flow_info *sfi, int flag,
                     int pkt_dequeue)
{
   struct tcp_state *tp = NULL;

   if(flag == FWD_FLOW) {
      tp = sfi->rhs_tcp_state;
   }
   else if(flag == PREV_FLOW) {
      tp = sfi->lhs_tcp_state;
   }

   if(tp->in_fast_recovery) {
      tp->cwnd +=1;
   }
   else if((tp->cwnd <= tp->ssthresh) && (pkt_dequeue == 1)) {
      tp->cwnd += 1;
   }
   else if(tp->cwnd > tp->ssthresh) {
      if (tp->cwnd_cnt >= tp->cwnd) {
         tp->cwnd++;
         tp->cwnd_cnt=0;
      } else
         tp->cwnd_cnt++;
   }

   return;
}

/*******************************************************/
/*        Various Packet Generation Functions          */
/*******************************************************/
/*
 * This function will prepare a TCP probe packet for 0 wnd.
 * 1. Allocate new skb
 * 2. Reserve space in skb for headers
 * 3. Populate tcph fields and then send fwd to IP
 */
void prepare_tcp_probe(unsigned long data)
{
   struct probe_info *pi = (struct probe_info *)data;
   struct split_flow_info *sfi = pi->sfi;
   struct sk_buff *new_skb;
```

```
    struct tcphdr *th;
    struct tcp_state *tp = NULL;
    struct flow_detail *out_flow = NULL;
    int tcp_header_size;

    if(pi->flag == PREV_FLOW) {
       tp = sfi->lhs_tcp_state;
       out_flow = sfi->i2r_flow;
    }
    else if(pi->flag == FWD_FLOW) {
       tp = sfi->rhs_tcp_state;
       out_flow = sfi->r2i_flow;
    }

    /* Allocate a new sk_buff */
    new_skb = alloc_skb(MAX_TCP_HEADER, GFP_ATOMIC);

    if(new_skb == NULL)
       goto skb_alloc_fail;

    /* Reserve space for headers */
    skb_reserve(new_skb, MAX_TCP_HEADER);
    tcp_header_size = (sizeof(struct tcphdr) );
    new_skb->h.th = th = (struct tcphdr *)
                         skb_push(new_skb, tcp_header_size);

    /* Filling the values in the fields */
    memset(th, 0, sizeof(struct tcphdr));
    th->ack = 1;
    th->source = out_flow->dport;
    th->dest = out_flow->sport;

    th->seq = htonl(tp->snd_next) - 1;
    th->ack_seq = htonl(tp->rcv_next);
    th->window = htons(tcp_select_window_(sfi, tp, pi->flag));
    th->doff = (tcp_header_size >> 2);

    /* Calculate TCP csum */
    new_skb->csum = 0;
    th->check = tcp_v4_check(th, new_skb->len,
                             out_flow->daddr, out_flow->saddr,
                             csum_partial((char *)th, new_skb->len,
                                          new_skb->csum));

    new_skb->dev = pi->in_dev;
```

```

   /* Done with TCP part. Build IP header now */
   ++tp->probes_out;
   add_ip_and_send(sfi, new_skb, pi->flag);
   return;

skb_alloc_fail:
   printk(KERN_ALERT "TCP probe skb alloc problem \n");
   return;
}

void prepare_tcp_reset(struct sk_buff *skb, struct net_device *in,
                       int ip_id)
{
   struct tcphdr *th;
   struct iphdr *iph;
   struct ethhdr *org_mac, *pkt_mac;
   struct sk_buff *new_skb;

   org_mac = (struct ethhdr *)skb->mac.raw;
   pkt_mac = (struct ethhdr *)
             kmalloc(sizeof(struct ethhdr), GFP_ATOMIC);
   iph = skb->nh.iph;
   th = (struct tcphdr *)(skb->data + iph->ihl*4);

   /* Allocated sk_buff for new packet */
   new_skb = alloc_skb(MAX_TCP_HEADER, GFP_ATOMIC);

   if(new_skb == NULL)
      goto skb_alloc_fail;

   /* Reserve space for headers */
   skb_reserve(new_skb, MAX_TCP_HEADER);

   /* Push and populate TCP hdr */
   new_skb->h.th = (struct tcphdr *)
                   skb_push(new_skb, sizeof(struct tcphdr));
   memset(new_skb->h.th, 0, sizeof(struct tcphdr));
   new_skb->h.th->doff = sizeof(struct tcphdr)/4;
   new_skb->h.th->source = th->dest;
   new_skb->h.th->dest = th->source;
   new_skb->h.th->rst = 1;

   if(th->ack)
      new_skb->h.th->seq = th->ack_seq;
```

```
    else {
       new_skb->h.th->ack = 1;
       new_skb->h.th->ack_seq = htonl(ntohl(th->seq) + th->syn +
                                      th->fin + skb->data -
                                      (skb->nh.iph->ihl<<2) -
                                      (skb->h.th->doff<<2));
    }

    new_skb->csum = 0;
    new_skb->h.th->check =
                     tcp_v4_check(new_skb->h.th, new_skb->len,
                                  iph->daddr, iph->saddr,
                                  csum_partial((char *)new_skb->h.th,
                                               new_skb->len,
                                               new_skb->csum));
    new_skb->dev = in;

    /* Push and populate the IP hdr */
    new_skb->nh.iph = (struct iphdr *)
                     skb_push(new_skb, sizeof(struct iphdr));
    memset(new_skb->nh.iph, 0, sizeof(struct iphdr));
    new_skb->nh.iph->version = 4;
    new_skb->nh.iph->ihl = 5;

    if(ip_id == 0)
       new_skb->nh.iph->id = th->seq ^ jiffies;
    else
       new_skb->nh.iph->id = ip_id;

    new_skb->nh.iph->tot_len = htons(new_skb->len);
    new_skb->nh.iph->ttl = IPDEFTTL;
    new_skb->nh.iph->protocol = IPPROTO_TCP;
    new_skb->nh.iph->saddr = iph->daddr;
    new_skb->nh.iph->daddr = iph->saddr;

    ip_send_check(new_skb->nh.iph);

    /* Push and populate the MAC hdr */
    new_skb->mac.raw = (struct ethhdr *)
                      skb_push(new_skb, sizeof(struct ethhdr));
    pkt_mac->h_proto = org_mac->h_proto;
    memcpy(pkt_mac->h_source, org_mac->h_dest, ETH_ALEN);
    memcpy(pkt_mac->h_dest, org_mac->h_source, ETH_ALEN);
    memcpy(new_skb->mac.raw, pkt_mac, sizeof(struct ethhdr));

    /* Send it out */
    dev_queue_xmit(new_skb);
    return;

skb_alloc_fail:
    printk(KERN_ALERT "RST skb alloc problem \n");
    return;
}

/*
 * This function will prepare a TCP ACK.
 * 1. Allocate new skb
 * 2. Reserve space in skb for headers
 * 3. Populate tcph fields and then send fwd to IP
 */
void prepare_tcp_ack(struct split_flow_info *sfi,
                     struct tcphdr *in_th, struct net_device *in,
                     int flag)
{
    struct sk_buff *new_skb;
    struct tcphdr *th;
    struct tcp_state *tp = NULL;
    struct flow_detail *out_flow = NULL;
    int tcp_header_size;

    if(flag == PREV_FLOW) {
        tp = sfi->lhs_tcp_state;
        out_flow = sfi->i2r_flow;
    }
    else if(flag == FWD_FLOW) {
        tp = sfi->rhs_tcp_state;
        out_flow = sfi->r2i_flow;
    }

    /* Allocate a new sk_buff */
    new_skb = alloc_skb(MAX_TCP_HEADER, GFP_ATOMIC);

    if(new_skb == NULL)
        goto skb_alloc_fail;

    /* Reserve space for headers */
    skb_reserve(new_skb, MAX_TCP_HEADER);
    tcp_header_size = (sizeof(struct tcphdr) );

    new_skb->h.th = th = (struct tcphdr *)
```

```
                              skb_push(new_skb, tcp_header_size);

   /* Filling the values in the fields */
   memset(th, 0, sizeof(struct tcphdr));
   th->ack = 1;
   th->source = out_flow->dport;
   th->dest = out_flow->sport;

   th->seq = htonl(tp->snd_next);
   th->ack_seq = htonl(tp->rcv_next);
   tp->snd_next = ntohl(th->seq);

   /* ECN related TCP processing */
   if(tp->ecn_capable) {
      if(tp->do_cwr)
         th->cwr = 1;
      if(tp->demand_cwr)
         th->ece = 1;
   }

   /* Need a way to compute window. This is just a patch up job */
   th->window = htons(tcp_select_window_(sfi, tp, flag));
   th->doff = (tcp_header_size >> 2);

   /* Calculate TCP csum */
   new_skb->csum = 0;
   th->check = tcp_v4_check(th, new_skb->len,
                            out_flow->daddr, out_flow->saddr,
                            csum_partial((char *)th, new_skb->len,
                                         new_skb->csum));

   new_skb->dev = in;

   /* For debugging */
   new_skb->cb[45] = 'L';

   /* Done with TCP part. Build IP header now */
   add_ip_and_send(sfi, new_skb, flag);
   return;

skb_alloc_fail:
   printk(KERN_ALERT "ACK skb alloc problem \n");
   return;
}

```

```
/*
 * This function will prepare a TCP FIN-ACK.
 * 1. Allocate new skb
 * 2. Reserve space in skb for headers
 * 3. Populate tcph fields and then send fwd to IP
 */
void prepare_tcp_finack(struct split_flow_info *sfi,
                        struct tcphdr *in_th, struct net_device *in,
                        int corr_val, int flag)
{
   struct sk_buff *new_skb;
   struct tcphdr *th;
   struct tcp_state *tp = sfi->lhs_tcp_state;
   struct flow_detail *out_flow = NULL;
   int tcp_header_size;

   if(flag == PREV_FLOW) {
      tp = sfi->lhs_tcp_state;
      out_flow = sfi->i2r_flow;
   }
   else if(flag == FWD_FLOW) {
      tp = sfi->rhs_tcp_state;
      out_flow = sfi->r2i_flow;
   }

   /* Allocate a new sk_buff */
   new_skb = alloc_skb(MAX_TCP_HEADER, GFP_ATOMIC);

   if(new_skb == NULL)
      goto skb_alloc_fail;

   /* Reserve space for headers */
   skb_reserve(new_skb, MAX_TCP_HEADER);
   tcp_header_size = (sizeof(struct tcphdr) );

   new_skb->h.th = th = (struct tcphdr *)
                        skb_push(new_skb, tcp_header_size);

   /* Filling the values in the fields */
   memset(th, 0, sizeof(struct tcphdr));
   th->fin = 1;
   th->ack = 1;
   th->source = out_flow->dport;
   th->dest = out_flow->sport;

```

```
    th->seq = htonl(tp->snd_next + corr_val);
    th->ack_seq = htonl(tp->rcv_next + 1 + corr_val);
    tp->rcv_next = ntohl(th->ack_seq);

    /* snd_next incremented. Next packet will be the last packet of
     * FIN. This will contain ACK = SEQ + 1. */
    tp->snd_next += (1 + corr_val);
    tp->snd_una = tp->snd_next;

    th->window = in_th->window;
    th->doff = (tcp_header_size >> 2);

    /* Have to calculate TCP csum */
    new_skb->csum = 0;
    th->check = tcp_v4_check(th, new_skb->len,
                             out_flow->daddr, out_flow->saddr,
                             csum_partial((char *)th, new_skb->len,
                                          new_skb->csum));

    new_skb->dev = in;

    /* Done with TCP part. Build IP header now */
    add_ip_and_send(sfi, new_skb, flag);
    return;

skb_alloc_fail:
    printk(KERN_ALERT "FIN-ACK skb alloc problem \n");
    return;
}

void prepare_tcp_synack(struct split_flow_info *sfi, int flag)
{
    struct flow_detail *in_flow, *out_flow;
    struct tcp_state *tps = NULL;
    struct tcp_state *prev_tps = NULL;
    struct skbuff_list *skb_node = NULL;
    struct sk_buff *org_skb;
    struct sk_buff *new_skb;
    struct sk_buff *synack_skb;
    struct tcphdr *th;
    struct iphdr *iph;
    int err, opt_length, tcp_hdr_size;
    __u32 *ptr;
    __u8 wscale = 0;
    unsigned int check_len;

    if(flag == PREV_FLOW) {
       tps = sfi->rhs_tcp_state;
       prev_tps = sfi->lhs_tcp_state;
       skb_node = head_peek_skb_list(sfi->i2r_queue);
       in_flow = sfi->i2r_flow;
       out_flow = sfi->r2i_flow;
    }
    else if(flag == FWD_FLOW) {
       tps = sfi->lhs_tcp_state;
       prev_tps = sfi->rhs_tcp_state;
       skb_node = head_peek_skb_list(sfi->r2i_queue);
       in_flow = sfi->r2i_flow;
       out_flow = sfi->i2r_flow;
    }

    /* Make copy of pkt that will actually be sent out */
    if(skb_node == NULL)
       goto err_skb_list;

    org_skb = skb_node->sb_pkt;
    skb_linearize(org_skb, GFP_ATOMIC);

    /* Check if nh is out of range */
    if(org_skb->nh.raw < org_skb->head ||
       org_skb->nh.raw > org_skb->tail)
       goto out_of_range;

    if(prev_tps->wscale_ok != 1) {
       synack_skb = alloc_skb(MAX_TCP_HEADER, GFP_ATOMIC);
       skb_reserve(synack_skb, MAX_TCP_HEADER);

       tcp_hdr_size = (sizeof(struct tcphdr) + TCPOLEN_MSS +
                       TCPOLEN_WSCALE_ALIGNED);
       if(tcp_hdr_size < (org_skb->h.th->doff<<2))
          tcp_hdr_size = org_skb->h.th->doff<<2;

       synack_skb->h.th =
       th = (struct tcphdr *)skb_push(synack_skb, tcp_hdr_size);
       memcpy(th, org_skb->h.th, (ntohs(org_skb->nh.iph->tot_len) -
                                  (org_skb->nh.iph->ihl<<2)));

       synack_skb->nh.iph =
       iph = (struct iphdr *)skb_push(synack_skb, sizeof(struct iphdr));
       memcpy(iph, org_skb->nh.iph, (org_skb->nh.iph->ihl<<2));
```

```

      synack_skb->mac.raw = (struct ethhdr *)
                            skb_push(synack_skb, ETH_HLEN);
      memcpy(synack_skb->mac.raw, org_skb->mac.raw, ETH_HLEN);
   }
   else {
      /* Get headers */
      iph = org_skb->nh.iph;
      th = org_skb->h.th;
      tcp_hdr_size = th->doff<<2;
   }

   /* Store the forwarding flow details */
   out_flow->saddr = iph->daddr;
   out_flow->daddr = iph->saddr;
   out_flow->sport = th->dest;
   out_flow->dport = th->source;

   /* Changing relevant TCP hdr fields */
   if(th == NULL) {
      goto err_tcphdr;
   }

   /* Get the route */
   if(org_skb->dst == NULL)
      if( (err = ip_route_input(org_skb, iph->daddr, iph->saddr,
                                iph->tos, org_skb->dev)) )
         goto bad_route;

   if(prev_tps->wscale_ok != 1) {
      synack_skb->dst = org_skb->dst;
      synack_skb->dev = org_skb->dev;
      if(org_skb->dst) {
         if(atomic_read(&org_skb->dst->__refcnt) < 1)
            atomic_set(&org_skb->dst->__refcnt, 1);
      }
         kfree_skb(org_skb);
         org_skb = synack_skb;
   }
   else
      synack_skb = org_skb;

   tps->rcv_next = ntohl(th->ack_seq);
   tps->rcv_wup = tps->rcv_next;
   tps->snd_next = ntohl(th->seq) + 1;
```

```
    tps->snd_una = tps->snd_next;
    tps->snt_isn = ntohl(th->seq);

    /* Adding ECN capability in TCP header */
    if(tps->ecn_capable && !prev_tps->ecn_capable) {
       th->ece = 1;
    }
    else if (!tps->ecn_capable && prev_tps->ecn_capable) {
       th->ece = 0;
       th->cwr = 0;
    }

    th->window = htons(dst_metric(org_skb->dst, RTAX_ADVMSS)<< 2);

    tps->wnd_clamp = 65535U;
    tps->local_rcv_wnd = ntohs(th->window);

    tps->snd_mss = min(dst_metric(org_skb->dst, RTAX_ADVMSS),
                         tps->mss_clamp);

    /* Correcting MSS for IPIP */
    if(prev_tps->forwarding_option == NO_IP_OVER_IP)
       tps->snd_mss -= 20;

    tps->mss_clamp = tps->snd_mss;

    /* Update TCP options */
    opt_length = tcp_hdr_size - sizeof(struct tcphdr)-1;
    ptr = th + 1;
    (*ptr++) = htonl((TCPOPT_MSS << 24) | (TCPOLEN_MSS << 16) |
                       tps->snd_mss);

    /* Window Scaling considerations */
    if(tps->wscale_ok == 1) {
       tps->wnd_clamp = 1048576U;
       calc_rcv_wscale(sfi->buff_clamp, tps->snd_mss, tps->wnd_clamp,
                         tps->wscale_ok, &wscale);
       tps->rcv_wscale = wscale;
       (*ptr++) = htonl((TCPOPT_NOP << 24) | (TCPOPT_WINDOW << 16) |
                          (TCPOLEN_WINDOW << 8) | wscale);
    }

    tps->wnd_clamp = min((65535U << tps->rcv_wscale), tps->wnd_clamp);
    prev_tps->wnd_clamp = min((65535U << prev_tps->rcv_wscale),
                                 prev_tps->wnd_clamp);

    /* NUlify all other TCP options */
    while(opt_length > 0) {
       *ptr++ = __constant_htonl((TCPOPT_NOP << 24)|(TCPOPT_NOP << 16)|
                                  (TCPOPT_NOP << 8) | TCPOPT_NOP);
       --opt_length;
    }

    /* Updating TCP & IP header length for wscale option */
    if(prev_tps->wscale_ok != 1) {
       if(tcp_hdr_size > (th->doff<<2)) {
          th->doff = tcp_hdr_size>>2;
          iph->tot_len = htons(ntohs(iph->tot_len) +
                                 TCPOLEN_WSCALE_ALIGNED);
       }
    }

    org_skb->csum = 0;
    check_len = ntohs(iph->tot_len) - (iph->ihl*4);

    th->check = 0;
    th->check = tcp_v4_check(th, check_len,
                          iph->saddr, iph->daddr,
                          csum_partial((char *)th, check_len,
                                       org_skb->csum));

 err_tcphdr:
    /* Changing relevant IP hdr fields */
    tps->ip_id = th->seq ^ jiffies;
    iph->id = tps->ip_id;
    iph->ttl = 16;

    /* Change the fragmentation option to indicate this
     * HB's presence */
    iph->frag_off = org_skb->nh.iph->frag_off | htons(IPIP_FRAG_OP);

    /* Adding ECN capability in IP header */
    if(tps->ecn_capable && !prev_tps->ecn_capable)
       iph->tos = 0x01;
    else if (!tps->ecn_capable && prev_tps->ecn_capable)
       iph->tos = 0x00;

    ip_send_check(iph);

    skb_node->pkt_state = SENT;
    if(tps->wscale_ok)
       skb_node->sb_pkt = synack_skb;

    /* Making copy iof sk_buff that will be sent out */
    new_skb = skb_cloned(org_skb) ? pskb_copy(org_skb, GFP_ATOMIC):
                                     skb_clone(org_skb, GFP_ATOMIC);

    skb_linearize(new_skb, GFP_ATOMIC);
    if(new_skb == NULL)
       goto err_skb_alloc;

    tps->state = TCP_SYNA_SENT;

    ip_output(new_skb);
    return;

 bad_route:
    printk(KERN_ALERT
           "Could not get route. Dropping SYN-ACK. err = %d \n", err);
    kfree_skb(new_skb);
    return;
 err_skb_list:
    printk(KERN_ALERT "Did not get packet from forward queue \n");
 err_skb_alloc:
    printk(KERN_ALERT "skb_copy failed for SYN-ACK \n");
    return;
 out_of_range:
    printk(KERN_ALERT "nh is out of range \n");
    return;
 }
 /*
  * This function prepares a TCP SYN packet for the forward connection
  * 1. Make a copy of the packet from the fwd-q
  * 2. Make changes in tcphdr - new seq, check, ack
  * 3. Update rhs_tcp_state variables
  * 4. Make changes in iphdr
  * 5. Get route by calling ip_route_input()
  * 6. Give packet to kernel for putting on wire by calling
  *    ip_forward_finish()
  * 06/06/2006: Replaced org_skb with syn_skb after creating syn_skb
  */
 void prepare_tcp_syn(struct split_flow_info *sfi, int flag)
 {
    struct flow_detail *in_flow, *out_flow;
    struct tcp_state *tps = NULL;
    struct tcp_state *prev_tps = NULL;
    struct skbuff_list *skb_node, *tx_queue = NULL;
    struct sk_buff *org_skb;
    struct sk_buff *new_skb;
    struct sk_buff *syn_skb;
    struct tcphdr *th;
    struct iphdr *iph;
    int err, opt_length, tcp_hdr_size;
    unsigned int check_len;
    __u32 *ptr;
    __u8 wscale = 0;

    if(flag == PREV_FLOW) {
       tps = sfi->rhs_tcp_state;
       prev_tps = sfi->lhs_tcp_state;
       tx_queue = sfi->i2r_queue;
       in_flow = sfi->i2r_flow;
       out_flow = sfi->r2i_flow;
    }
    else if(flag == FWD_FLOW) {
       tps = sfi->lhs_tcp_state;
       prev_tps = sfi->rhs_tcp_state;
       tx_queue = sfi->r2i_queue;
       in_flow = sfi->r2i_flow;
       out_flow = sfi->i2r_flow;
    }
    skb_node = head_peek_skb_list(tx_queue);

    /* Make copy of pkt that will actually be sent out */
    if(skb_node == NULL)
       goto err_skb_list;

    org_skb = skb_node->sb_pkt;
    skb_linearize(org_skb, GFP_ATOMIC);

    /* Check if nh is out of range */
    if(org_skb->nh.raw < org_skb->head ||
       org_skb->nh.raw > org_skb->tail)
       goto out_of_range;

    if(prev_tps->wscale_ok != 1) {
       syn_skb = alloc_skb(MAX_TCP_HEADER, GFP_ATOMIC);
       skb_reserve(syn_skb, MAX_TCP_HEADER);

       tcp_hdr_size = (sizeof(struct tcphdr) + TCPOLEN_MSS +
                           TCPOLEN_WSCALE_ALIGNED);
      if(tcp_hdr_size < (org_skb->h.th->doff<<2))
         tcp_hdr_size = org_skb->h.th->doff<<2;

      syn_skb->h.th =
      th = (struct tcphdr *)skb_push(syn_skb, tcp_hdr_size);
      memcpy(th, org_skb->h.th, (ntohs(org_skb->nh.iph->tot_len) -
                                 (org_skb->nh.iph->ihl<<2)));

      syn_skb->nh.iph =
      iph = (struct iphdr *)skb_push(syn_skb, sizeof(struct iphdr));
      memcpy(iph, org_skb->nh.iph, (org_skb->nh.iph->ihl<<2));

      syn_skb->mac.raw = (struct ethhdr *)
                             skb_push(syn_skb, ETH_HLEN);
      memcpy(syn_skb->mac.raw, org_skb->mac.raw, ETH_HLEN);
   }
   else {
      /* Get headers */
      iph = org_skb->nh.iph;
      th = org_skb->h.th;
      tcp_hdr_size = th->doff<<2;
   }

   /* Store the forwarding flow details */
   out_flow->saddr = iph->daddr;
   out_flow->daddr = iph->saddr;
   out_flow->sport = th->dest;
   out_flow->dport = th->source;

   /* Changing relevant TCP hdr fields */
   if(th == NULL) {
      goto err_tcphdr;
   }

   /* Get the route */
   if(org_skb->dst == NULL)
      if((err = ip_route_input(org_skb, iph->daddr, iph->saddr,
                               iph->tos, org_skb->dev)))
         goto bad_route;

   if(prev_tps->wscale_ok != 1) {
      syn_skb->dst = org_skb->dst;
      syn_skb->dev = org_skb->dev;
      if(org_skb->dst) {
            if(atomic_read(&org_skb->dst->__refcnt) < 1)
               atomic_set(&org_skb->dst->__refcnt, 1);
            }
         kfree_skb(org_skb);
      }
      else
         syn_skb = org_skb;

      tps->recover = ntohl(th->seq);
      th->ack_seq = 0;
      tps->rcv_next = 0;
      tps->rcv_wup = tps->rcv_next;
      tps->snd_next = ntohl(th->seq) + 1;
      tps->snd_una = tps->snd_next;
      tps->snt_isn = ntohl(th->seq);

      /* Adding ECN functionality in TCP header */
      if(!prev_tps->ecn_capable) {
         th->ece = 1;
         th->cwr = 1;
      }

      th->window = htons(dst_metric(syn_skb->dst, RTAX_ADVMSS)<< 2);
      tps->local_rcv_wnd = ntohs(th->window);

      tps->snd_mss = min(dst_metric(syn_skb->dst, RTAX_ADVMSS),
                         tps->mss_clamp);

      /* Correcting MSS for IPIP */
      if(prev_tps->forwarding_option == NO_IP_OVER_IP)
         tps->snd_mss -= 20;

      tps->mss_clamp = tps->snd_mss;

      calc_rcv_wscale(sfi->buff_clamp, tps->snd_mss, 1048576U, 1,
                      &wscale);
      tps->rcv_wscale = wscale;

      /* Update TCP options */
      opt_length = tcp_hdr_size - sizeof(struct tcphdr)-1;
      ptr = th + 1;
      (*ptr++) = htonl((TCPOPT_MSS << 24) | (TCPOLEN_MSS << 16) |
                        tps->snd_mss);
      (*ptr++) = htonl((TCPOPT_NOP << 24) | (TCPOPT_WINDOW << 16) |
                       (TCPOLEN_WINDOW << 8) | wscale);
```

```
    while(opt_length > 0) {
       *ptr++ = __constant_htonl((TCPOPT_NOP << 24)|(TCPOPT_NOP << 16)|
                                 (TCPOPT_NOP << 8) | TCPOPT_NOP);
       --opt_length;
    }

    /* Updating TCP & IP header length for wscale option */
    if(prev_tps->wscale_ok != 1) {
       if(tcp_hdr_size > (th->doff<<2)) {
          th->doff = tcp_hdr_size>>2;
          iph->tot_len = htons(ntohs(iph->tot_len) +
                               TCPOLEN_WSCALE_ALIGNED);
       }
    }

    syn_skb->csum = 0;
    check_len = ntohs(iph->tot_len) - (iph->ihl*4);

    th->check = 0;
    th->check = tcp_v4_check(th, check_len, iph->saddr, iph->daddr,
                             csum_partial((char *)th, check_len,
                                          syn_skb->csum));

 err_tcphdr:
    /* Changing relevant IP hdr fields */
    tps->ip_id = th->seq ^ jiffies;
    iph->id = tps->ip_id;
    iph->ttl = IPDEFTTL;
    iph->protocol = syn_skb->nh.iph->protocol;

    /* Change the fragment option to indicate this HB's presence */
    iph->frag_off = syn_skb->nh.iph->frag_off | htons(IPIP_FRAG_OP);

    /* Adding ECN capability in IP header */
    if(!prev_tps->ecn_capable) {
       iph->tos = 0x01;
    }

    ip_send_check(iph);

    skb_node->pkt_state = SENT;
    skb_node->sb_pkt = syn_skb;

    /* Making copy iof sk_buff that will be sent out */
    new_skb = skb_cloned(org_skb) ? pskb_copy(syn_skb, GFP_ATOMIC)
```

```
                                        : skb_clone(syn_skb, GFP_ATOMIC);

    skb_linearize(new_skb, GFP_ATOMIC);
    if(new_skb == NULL)
       goto err_skb_alloc;

    tps->state = TCP_SYN_SENT;

    ip_output(new_skb);

    /* Note time and Start the rto timer */
    skb_node->snd_tstamp = jiffies;
    tps->rtt_seq_tstamp = jiffies;

    init_rto_timer(tx_queue, tps, 1);
    return;

bad_route:
    printk(KERN_ALERT
            "Could not get route. Dropping forward SYN. err = %d \n", err);
    kfree_skb(new_skb);
    return;
err_skb_list:
    printk(KERN_ALERT "Did not get packet from forward queue \n");
err_skb_alloc:
    printk(KERN_ALERT "skb_copy failed for forward SYN \n");
    return;
out_of_range:
    printk(KERN_ALERT "nh is out of range \n");
    return;
}

/*
 * This function is used to retransmit a data packet in the FWD_FLOW
 * No header fields are modified as we just need to retransmit the
 * data.
 */
void prepare_fwd_retrans_data(unsigned long data)
{
    struct skbuff_list *queue_head = (struct skbuff_list *)data;
    struct tcp_state *tps = queue_head->tps_ptr;
    struct skbuff_list *skb_node;
    struct sk_buff *skb, *org_skb;
    struct iphdr *iph;

    if(get_queue_pkt_count(queue_head) <= 0)
       goto sb_pkt_err;

    skb_node = head_peek_skb_list(queue_head);

    if(skb_node->pkt_state == NOT_SENT) {
       goto unsent_pkt_err;
    }

    org_skb = skb_node->sb_pkt;
    skb_linearize(org_skb, GFP_ATOMIC);

    /* Making copy of sk_buff that will be sent out */
    skb = skb_cloned(org_skb) ? pskb_copy(org_skb, GFP_ATOMIC)
                              : skb_clone(org_skb, GFP_ATOMIC);

    if(skb == NULL)
       goto err_skb_alloc;

    iph = skb->nh.iph;

    /* Get the route */
    if(skb->dst == NULL)
       if(ip_route_input(skb, iph->daddr, iph->saddr, iph->tos,
          skb->dev))
          goto bad_route;

    /* Giving to kernel to actually send it */
    ip_output(skb);

    /* Note the time when the packet was sent */
    skb_node->snd_tstamp = jiffies;
    skb_node->pkt_state = RETRANSMIT;

    /* Setting ssthresh = max(Flight Size / 2, 2 * MSS) */
    del_timer_sync(&(tps->rto_timer));
    if(rto_timer_expired == 0) {
       tps->ssthresh = max(tps->pkts_in_flight/2, 2);
       tps->cwnd = tps->ssthresh + 3;
       tps->recover = tps->snd_next;
       init_rto_timer(queue_head, tps, 1);
    }
    else if(rto_timer_expired != 0 && rto_timer_expired != 2){
       tps->ssthresh = max(tps->pkts_in_flight/2, 2);
       tps->cwnd = 1;
```

```
        init_rto_timer(queue_head, tps, 2);
    }
    else if(rto_timer_expired == 2)
        init_rto_timer(queue_head, tps, 1);

    rto_timer_expired = 1;

    return;

bad_route:
    printk(KERN_ALERT
            "Could not get route. Dropping retransmit packet \n");
    if(skb->dst) {
        if(atomic_read(&skb->dst->__refcnt) < 1)
            atomic_set(&skb->dst->__refcnt, 1);
    }
    kfree_skb(skb);
    return;
sb_pkt_err:
    printk(KERN_ALERT
            "Trying to retransmit when there are no packets \n");
    return;
unsent_pkt_err:
    printk(KERN_ALERT
      "Trying to retransmit a packet that has not been fwded yet \n");
    return;
err_skb_alloc:
    printk(KERN_ALERT "skb_copy failed for retransmission \n");
    return;
}

/*
 * This function prepares the data packets for the fwd-flow. TCP
 * handshake has been completed
 * TO DO: Check if locks need to be added for the fwd-q.
 */
int prepare_fwd_data(unsigned long data, int flag)
{
    struct split_flow_info *sfi = (struct split_flow_info *)data;
    unsigned long *queue_head_data;
    struct sk_buff *new_skb, *org_skb, *skb, *ipip_skb;
    struct skbuff_list *skb_node, *curr_skbl;
    struct skbuff_list *tx_queue = NULL;
    struct tcp_state *tps = NULL;
    struct tcp_state *src_tps = NULL;
    struct flow_detail *in_flow = NULL;
    struct tcphdr *th;
    struct iphdr *iph, *ipip_iph;
    __u32 data_bytes, check_len, seq;
    int first_pkt = 1, restart_rto_timer = 0,
        pkts_dequeued = 0, ack_retransmit = 0;

    if(flag == PREV_FLOW) {
       tps = sfi->rhs_tcp_state;
       src_tps = sfi->lhs_tcp_state;
       tx_queue = sfi->i2r_queue;
       in_flow = sfi->i2r_flow;
    }
    else if(flag == FWD_FLOW) {
       tps = sfi->lhs_tcp_state;
       src_tps = sfi->rhs_tcp_state;
       tx_queue = sfi->r2i_queue;
       in_flow = sfi->r2i_flow;
    }

    if(get_queue_pkt_count(tx_queue) <= 0) {
       if(timer_pending(&(tps->rto_timer)))
          del_timer_sync(&(tps->rto_timer));
       goto err_fwdq_empty;
    }

    skb_node = head_peek_skb_list(tx_queue);

    while(tx_queue->pkt_count > 0) {
       curr_skbl = head_peek_skb_list(tx_queue);
       skb = curr_skbl->sb_pkt;

       if(skb->nh.iph->protocol == IPPROTO_IPIP) {
          iph = get_ip_header(skb);
          th = get_tcp_header(skb);
       }
       else {
          iph = skb->nh.iph;
          th = skb->h.th;
       }

       data_bytes = ntohs(iph->tot_len) - (iph->ihl*4)
                                        - (th->doff*4)
                                        + th->fin;
       seq = ntohl(th->seq) + data_bytes;
```

```

      if((curr_skbl->pkt_state == SENT ||
          curr_skbl->pkt_state == RETRANSMIT) &&
          (seq <= tps->ack_seq)) {
         /* Update pkt_bfr_hole */
         if((tx_queue->hole_in_queue) &&
            (curr_skbl == tx_queue->pkt_bfr_hole))
            tx_queue->pkt_bfr_hole = tx_queue;
         /* Free the memory */
         write_lock(&tx_queue->lock);
         free_head_skbuff_list(sfi, tx_queue);
         write_unlock(&tx_queue->lock);

         if(curr_skbl->pkt_state == RETRANSMIT)
            ack_retransmit = 1;

         /* Update packets in flight if not first good ack
          * after dup acks */
         tps->pkts_in_flight -= 1;
         pkts_dequeued += 1;

         /* Free data bytes off the buffer */
         tcp_free_buffer(sfi, data_bytes, tps);

         /* Check if we can increase the cnwd */
         tcp_update_cwnd(sfi, (flag^1), 1);

         /* RTT calculations */
         if(tps->ack_seq == seq) {
            if(!ack_retransmit) {
               tps->rtt_seq_tstamp = curr_skbl->snd_tstamp;
               tcp_rtt_estimate(tps);
               tcp_set_rto(tps);
               tcp_bound_rto(tps);
            }
            del_timer_sync(&(tps->rto_timer));
            restart_rto_timer += 1;
         }
      }
      else {
         /* This is the first packet not acked by the ACK  */
         /* Hence its our new snd_una only if it was sent before */
         if(curr_skbl->pkt_state != NOT_SENT) {
            tps->snd_una = seq - data_bytes;
            first_pkt = 0;
            }
            break;
         }

         /* If fwd-q is empty and PREV_FLOW is closed,
          * prepare FIN for FWD_FLOW */
         if(tx_queue->pkt_count == 0) {
            /* Delete rto timer if its running */
            if(timer_pending(&(tps->rto_timer)))
               del_timer_sync(&(tps->rto_timer));
            goto err_fwdq_empty;
         }
      }

      skb_node = curr_skbl;

      /* Update how many packets we can send now                    *
       * If the packet dequeue logic was used and snd_una was updated,*
       * then there are packets in flight. Hence we check that       *
       * packets in flight < current rcv_wnd                         */
      if(tcp_wnd_may_update(sfi, (flag^1)) && !first_pkt)
         tps->pkt_wnd = min(tps->cwnd, tps->rcv_wnd_pkt);

      /* snd_una was not updated by dequeue logic because 'first_pkt' *
       * is 1. This means that there are no packets in flight. Hence  *
       * we can update the wnd                                        */
      else if(first_pkt)
         tps->pkt_wnd = min(tps->cwnd, tps->rcv_wnd_pkt);

      if((tps->pkt_wnd * tps->mss_clamp ) > tps->rcv_wnd)
         goto wnd_overflow;

      /* Update packets in flight if not first good ack after dup acks*/
      /* Reset first_good_ack if the ACK ack's all outstanding packet */
      if(tps->first_good_ack && (tps->ack_seq >= tps->recover)) {
         tps->first_good_ack = 0;
      }

      if(tps->pkts_in_flight >= tps->pkt_wnd)
         tps->pkt_wnd = 0;
      else
         tps->pkt_wnd -= tps->pkts_in_flight;

      if(tps->pkt_wnd <= 0) {
         /* Do fast retransmit even if window is 0 */
```

```
        if(tps->dup_ack_cnt == 3 && (!tps->in_fast_recovery))
            goto prepare_pkt;
        /* Retransmit if partial ACK in Fast recovery */
        else if(tps->in_fast_recovery && tps->first_good_ack)
            goto prepare_pkt;
        /* Restart timer if there are unacked packets AND  *
         * timer has been deleted                          */
        else if(skb_node->pkt_state != NOT_SENT &&
                (!(timer_pending(&(tps->rto_timer))))) {
            restart_rto_timer = 1;
            goto restart_timer;
        }
        else
            goto err_zero_wnd;
    }

prepare_pkt:
    skb_node = head_peek_skb_list(tx_queue);

    if(skb_node->pkt_state != NOT_SENT) {
        /* Enter Fast Retransmit if allowed */
        if(tps->dup_ack_cnt == 3 && (!tps->in_fast_recovery) &&
            (tps->ack_seq >= tps->recover)) {
            queue_head_data = (unsigned long *)tx_queue;
            rto_timer_expired = 0;
            tps->in_fast_recovery = 1;
            prepare_fwd_retrans_data(queue_head_data);
            return 1;
        }
        /* Dealing with partial ACK's */
        else if(tps->in_fast_recovery && tps->first_good_ack &&
                (tps->ack_seq < tps->recover)) {
            rto_timer_expired = 2;
            tps->first_good_ack = 0;
            queue_head_data = (unsigned long *)tx_queue;
            prepare_fwd_retrans_data(queue_head_data);
            tps->cwnd -= pkts_dequeued;
            if(pkts_dequeued > 0)
                tps->cwnd += 1;
            return 1;
        }
        /* Incoming packet had ECE = 1. Treating it as a lost
         * packet signal */
        else if(tps->do_cwr && !tps->prev_do_cwr) {
            queue_head_data = (unsigned long *)tx_queue;
            rto_timer_expired = 0;
            prepare_fwd_retrans_data(queue_head_data);
            return 1;
         }
         else {
            /* Find first packet that is not dirty */
            while(skb_node != tx_queue) {
               if(skb_node->pkt_state == NOT_SENT)
                  /* We found the first pkt not sent till now */
                  break;
               skb_node = skb_node->next;
            }

            /* Check whether we came out of the loop because all
             * packets were dirty. If yes, and if the PREV_FLOW has
             * closed => no more packets to send, return
             * else fwd_q is empty, init timer
             */
            if(skb_node == tx_queue) {
               tps->pkt_wnd = 0;
               goto restart_timer;
            }
         }
      }

      org_skb = skb_node->sb_pkt;

      /* Check if nh is out of range */
      if(org_skb->nh.raw < org_skb->head ||
         org_skb->nh.raw > org_skb->tail)
         goto out_of_range;

      iph = org_skb->nh.iph;
      th = org_skb->h.th;

      /* Check if this packet is after a hole */
      if(ntohl(th->seq) > src_tps->rcv_next) {
         prepare_tcp_ack(sfi, th, org_skb->dev, flag);
         goto err_hole_pkt;
      }

      /* Changing relevant TCP header fields */
      data_bytes = ntohs(iph->tot_len) - (iph->ihl*4) - (th->doff*4);
      th->ack_seq = htonl(tps->rcv_next);

```

```
    if(first_pkt) {
       tps->snd_una = tps->snd_next;
       first_pkt = 0;

       /* Restart the rto timer */
       if(timer_pending(&(tps->rto_timer)))
          del_timer_sync(&(tps->rto_timer));
       restart_rto_timer = 1;
    }
    tps->snd_next += data_bytes + th->fin;

    /* ECN related TCP processing */
    if(tps->ecn_capable && skb_node->pkt_state == NOT_SENT) {
       if(tps->do_cwr) {
          th->cwr = 1;
          printk(KERN_ALERT "CWR set in outgoing packet \n");
       }
       if(tps->demand_cwr) {
          th->ece = 1;
          printk(KERN_ALERT "ECE set in outgoing packet \n");
       }
    }

    th->window = htons(tcp_select_window_(sfi, tps, (flag^1)));
    org_skb->csum = 0;
    check_len = ntohs(iph->tot_len) - (iph->ihl*4);
    th->check = 0;
    th->check = tcp_v4_check(th, check_len, iph->saddr, iph->daddr,
                             csum_partial((char *)th, check_len,
                                          org_skb->csum));


    /* Changing relevant IP header fields */
    iph->id = ++tps->ip_id;
    iph->ttl = IPDEFTTL;
    iph->protocol = org_skb->nh.iph->protocol;
    iph->frag_off = org_skb->nh.iph->frag_off;
    /* ECN related IP processing */
    if(tps->ecn_capable)
       iph->tos = 0x02;
    ip_send_check(iph);

    skb_node->pkt_state = SENT;

    if(tps->forwarding_option == IP_OVER_IP) {
        /* Allocate memory
         * Copy original packet
         * Give values to the new IP header
         */
        int new_headroom = skb_headroom(org_skb) + sizeof(struct iphdr);
        org_skb = skb_realloc_headroom(org_skb, new_headroom);

        /* Adding IP hdr at the start */
        org_skb->h.raw = org_skb->nh.raw;
        org_skb->nh.raw = skb_push(org_skb, sizeof(struct iphdr));

        ipip_iph                =       org_skb->nh.iph;
        ipip_iph->version       =       4;
        ipip_iph->ihl           =       sizeof(struct iphdr)>>2;
        ipip_iph->tos           =       iph->tos;
        ipip_iph->tot_len       =       htons(org_skb->len);
        ipip_iph->id            =       iph->id;
        ipip_iph->frag_off      =       iph->frag_off;
        ipip_iph->ttl           =       IPDEFTTL;
        ipip_iph->protocol      =       IPPROTO_IPIP;
        ipip_iph->daddr         =       tps->next_hb_addr;
        ipip_iph->saddr         =       iph->saddr;

        ip_send_check(ipip_iph);
        skb_node->sb_pkt = org_skb;
    }

    /* Making copy of sk_buff that will be sent out */
    new_skb = skb_cloned(org_skb) ? pskb_copy(org_skb, GFP_ATOMIC)
                                  : skb_clone(org_skb, GFP_ATOMIC);

    new_skb->data_len = org_skb->data_len;

    skb_linearize(new_skb, GFP_ATOMIC);

    if(new_skb == NULL)
       goto err_skb_alloc;

    /* Get the route */
    if(new_skb->dst == NULL)
       if(ip_route_input(new_skb, iph->daddr, iph->saddr, iph->tos,
          new_skb->dev))
          goto bad_route;

    /* Giving to kernel to actually send it */
    ip_output(new_skb);

    /* Note the time when the packet was sent */
    skb_node->snd_tstamp = jiffies;

    /* Check if forwarding FIN packet */
    if(th->fin && tps->state != TCP_FIN_RCVD) {
       tps->state = TCP_FIN_SENT;
    }

    /* Update packet window and packets in flight */
    tps->pkts_in_flight += 1;
    tps->pkt_wnd -= 1;

restart_timer:
    /* Restart/start the rto timer now */
    if(restart_rto_timer == 1) {
       if(timer_pending(&(tps->rto_timer)))
          del_timer_sync(&(tps->rto_timer));
    init_rto_timer(tx_queue, tps, 1);
    restart_rto_timer += 1;
    }

    if(tps->pkt_wnd > 0) {
       goto prepare_pkt;
    }

    return 1;

err_fwdq_empty:
    printk(KERN_ALERT "No more packets in fwd-q \n");
    return 0;
bad_route:
    printk(KERN_ALERT
            "Could not get route. Dropping forward packet \n");
    if(skb->dst) {
       if(atomic_read(&skb->dst->__refcnt) < 1)
          atomic_set(&skb->dst->__refcnt, 1);
    }
    kfree_skb(new_skb);
    return 0;
err_skb_alloc:
    printk(KERN_ALERT "skb_copy failed for forward packet \n");
    return 0;
out_of_range:
   printk(KERN_ALERT "nh is out of range \n");
   return 0;
wnd_overflow:
   printk(KERN_ALERT "fwd wnd exhausted. Not sending packet \n");
   return 0;
err_zero_wnd:
   printk(KERN_ALERT "fwd wnd is 0. Not sending packet \n");
   return -1;
err_hole_pkt:
   printk(KERN_ALERT "Error: Tried to send packet after hole.\n");
   return 0;
}

/*
 * This function processes the ECN flags of the incoming packet
 * We set demand_cwr and do_cwr here
 */
void process_ecn_flags(struct tcp_state *tps)
{
   int ecn_state;

   tps->prev_do_cwr = tps->do_cwr;
   ecn_state = tps->ecn_flags;

   switch(ecn_state) {
      case 2:
      case 3:
         tps->demand_cwr = 0;
         tps->do_cwr = 1;
         break;
      case 4:
      case 5:
         tps->demand_cwr = 1;
         tps->do_cwr = 0;
         break;
      case 6:
      case 7:
         tps->demand_cwr = 1;
         tps->do_cwr = 1;
         break;
      default:
         tps->demand_cwr = 0;
         tps->do_cwr = 0;
         break;
   }
```

```
}

/*
 * This function processes an incoming ACK. The ACK can be for fwd
 * or prev flow.
 * Updates the proper tcp_state variable
 * Imp: tp->rcv_next is set here
 */
int process_tcp_ack(struct split_flow_info *sfi, struct sk_buff *skb,
                    int flag)
{
   struct tcp_state *tp = NULL;
   struct tcp_state *fwd_tp = NULL;
   struct tcphdr *tcph = skb->h.th;
   struct iphdr *iph = skb->nh.iph;
   struct skbuff_list *rcv_queue = NULL;
   struct skbuff_list *tx_queue = NULL;
   __u32 ack = ntohl(tcph->ack_seq);
   __u32 seq = ntohl(tcph->seq);
   __u32 r_seq;
   __u32 un_ack;
   int can_update, no_hole;
   unsigned long data = (unsigned long *)sfi;
   int two_way_xfer;

   if(flag == PREV_FLOW) {
      tp = sfi->lhs_tcp_state;
      fwd_tp = sfi->rhs_tcp_state;
      rcv_queue = sfi->i2r_queue;
      tx_queue = sfi->r2i_queue;
   }
   else if(flag == FWD_FLOW) {
      tp = sfi->rhs_tcp_state;
      fwd_tp = sfi->lhs_tcp_state;
      rcv_queue = sfi->r2i_queue;
      tx_queue = sfi->i2r_queue;
   }

   two_way_xfer = tp->data_pkt_seen & fwd_tp->data_pkt_seen;

   r_seq = tp->rcv_next;
   un_ack = tp->snd_una;

   if(ack < un_ack) {
      if(tp->state == TCP_CLOSING)
```

```
            /* Retransmitting the FIN-ACK */
            prepare_tcp_finack(sfi, tcph, skb->dev, -1, flag);
         goto uninteresting_ack;
      }

      /* Update rcv_wnd using the adv_wnd of the ACK and wscale factor */
      tp->rcv_wnd = ntohs(tcph->window);
      tp->rcv_wnd <<= tp->snd_wscale;
      tp->rcv_wnd_pkt = tp->rcv_wnd / tp->mss_clamp;

      /* Getting all ECN information from the packet   */
      /* We record ECN info off every packet because - */
      /* 1. Router marks packet even if its dup     */
      /* 2. No dup packet will have TCP ECN markers         */
      if(tp->ecn_capable && !tcph->syn) {
         tp->ce = ((iph->tos & 0x03) == 0x03) ? 1 : 0;
         tp->ece = tcph->ece;
         tp->cwr = tcph->cwr;

         tp->ecn_flags = ((tp->ce << 2) | (tp->ece << 1) | (tp->cwr));

         process_ecn_flags(tp);
      }

      /* Send a ACK if we have already recieved this packet */
      if(seq < r_seq) {
         /* Fwd data if you can */
         if((seq + 1) == r_seq &&
            rcv_queue->pkt_bfr_hole != rcv_queue &&
            get_queue_pkt_count(rcv_queue) > 0 &&
            fwd_tp->rcv_wnd > 0) {
            prepare_fwd_data(data, flag);
         }
         /* Send ACK to give current status to the host */
         prepare_tcp_ack(sfi, tcph, skb->dev, flag);
         goto bad_r_seq;
      }
      /* New packet. Enqueue it and update window if allowed */
      else if((can_update = tcp_wnd_may_update(sfi, flag))) {
         no_hole = enqueue_packet(sfi, skb, rcv_queue, tp);
      }

      if(ack >= un_ack) {
         if((ack == tp->prev_ack_seq) && ((tp->end_seq - seq) <= 0)) {
            tp->dup_ack_cnt += 1;
            tcp_update_cwnd(sfi, flag, 0);
        }
        else {
            /* Charge the buffer and wnd */
            if(tp->in_fast_recovery) {
                tp->first_good_ack = 1;
                if(ack >= tp->recover) {
                    tp->in_fast_recovery = 0;
                    tp->cwnd = tp->ssthresh;
                    tp->dup_ack_cnt = 0;
                }
            }
            else
                tp->dup_ack_cnt = 0;
        }

        tp->prev_ack_seq = ack;

        if(can_update) {
            if(no_hole != -1) {
                tp->rcv_next = tp->end_seq;
                tp->max_rcv_byte = tp->rcv_next;
            }

            if(tp->state == TCP_SYNA_SENT ||
                tp->state == TCP_SYN_SENT) {
                /* Dequeuing SYN packet from fwd-queue */
                free_head_skbuff_list(sfi, tx_queue);

                if(tp->state == TCP_SYN_SENT) {
                    if(no_hole != -1) {
                        tp->rcv_next++;
                        tp->max_rcv_byte = tp->rcv_next;
                        tp->rcv_wup = tp->rcv_next;
                    }

                    if(timer_pending(&(tp->rto_timer)))
                        del_timer_sync(&(tp->rto_timer));
                }
                tp->state = TCP_CONNECTED;
            }
            else if(tp->state == TCP_CLOSING) {
                /* Got FIN but there is a hole in the queue */
                if(no_hole == -1)
                    tp->state = TCP_FIN_WAIT;
            else if(two_way_xfer) {
               tp->state = TCP_FIN_RCVD;
            }
         }
         else if(tp->state == TCP_FIN_WAIT) {
            /* Got the missing packets. Can send FIN-ACK now */
            if(no_hole == -1 && rcv_queue->hole_in_queue == 0) {
               if(two_way_xfer) {
                  tp->state = TCP_FIN_RCVD;
               }
               else
                  tp->state = TCP_CLOSING;
            }
         }
         else if(tp->state == TCP_FIN_SENT) {
         }

         /* Do we need to probe for 0 window
          * Is adv_mss < MSS AND
          *    unsent packets in fwd_q AND
          *    sent probes < max probes that can be sent
          */
         if((ntohl(tcph->window) < tp->snd_mss) &&
            (tp->pkts_in_flight < get_queue_pkt_count(tx_queue)) &&
             tp->probes_out < TCP_RETR2) {
            if(timer_pending(&(tp->probe_timer)))
               del_timer_sync(&(tp->probe_timer));

            if(timer_pending(&(tp->rto_timer)))
               del_timer_sync(&(tp->rto_timer));

            pinfo->sfi = sfi;
            pinfo->flag = flag;
            pinfo->in_dev = skb->input_dev;

            init_probe_timer(pinfo, tp, 2);
         }
         else if((ntohl(tcph->window) >= tp->snd_mss) &&
                  tp->probes_out > 0) {
            if(timer_pending(&(tp->probe_timer)))
               del_timer_sync(&(tp->probe_timer));

            tp->probes_out = 0;
         }
         return 0;
```

```
        }
    }

bad_r_seq:
    printk(KERN_ALERT "Incoming seq < rcv_next. Dropping packet \n");
    return -1;
uninteresting_ack:
    printk(KERN_ALERT "Incoming ack < snd_una. Dropping packet \n");
    return -1;
}

/*
 * This function process the packet contents.
 * 1. Pull the TCP hdr and perform basic checks
 * 2. If from prev flow, enqueue the packet in fwd-q
 * 3. Calcualte tp->end_seq. This is used to set tp->rcv_next
 * 4. If an ACK & ! FIN
 *      a. ACK for fwd connection SYN
 *      b. ACK for prev connection SYN-ACK
 *      c. ACK for data packet
 * 5. If a SYN (assuming SYN's only come from src)
 *      a. prepare SYN-ACK
 *      b. prepare SYN for fwd connection
 * 6. If FIN (have to handle FIN of fwd connection)
 *      a. prepare FIN for prev connection
 */
void process_in_pkt(struct sk_buff *skb, struct split_flow_info *sfi,
     int flag)
{
    struct tcp_options_received tp;
    //struct tcp_opt tp;
    struct tcp_state *tps = NULL;
    struct tcp_state *fwd_tps = NULL;
    struct tcphdr *th;
    struct iphdr *iph;
    struct Qdisc *qdisc;
    struct net_device *dev;
    int ihl = skb->nh.iph->ihl*4;
    unsigned long data = (unsigned long *)sfi;

    skb->h.raw = skb->data + skb->nh.iph->ihl*4;

    if(!pskb_may_pull(skb, sizeof(struct tcphdr)))
        goto discard_it;

    th = skb->h.th;

    if(th->doff < (sizeof(struct tcphdr)/4))
       goto bad_packet;

    if(!pskb_may_pull(skb, th->doff*4))
       goto discard_it;

    th = skb->h.th;
    iph = skb->nh.iph;

    if(flag == PREV_FLOW) {
       tps = sfi->lhs_tcp_state;
       fwd_tps = sfi->rhs_tcp_state;
    }
    else if(flag == FWD_FLOW) {
       tps = sfi->rhs_tcp_state;
       fwd_tps = sfi->lhs_tcp_state;
    }

    tps->end_seq = ntohl(th->seq) + skb->len - (th->doff*4) - ihl;
    tps->ack_seq = ntohl(th->ack_seq);
    tps->ack_seq_tstamp = jiffies;

    tcp_parse_options(skb, &tp, 0);

    /* Is this a data packet ? */
    if(!tps->data_pkt_seen) {
       if((tps->end_seq - ntohl(th->seq)) > 0) {
          tps->data_pkt_seen = 1;
       }
    }

    if(th->ack && !th->fin && !th->rst) {
       /* SYN-ACK packet of fwd connection */
       if(tps->state == TCP_SYN_SENT) {
          /* Check if we need to use the default MSS */
          if(tp.mss_clamp != NULL)
             fwd_tps->mss_clamp = tp.mss_clamp;
          else
             fwd_tps->mss_clamp = 536;

          /* Check if the packet is coming from end host or previous
           * helper box */
          if((ntohs(skb->nh.iph->frag_off) & IPIP_FRAG_OP) ==
                                                                 IPIP_FRAG_OP)
                    tps->forwarding_option = IP_OVER_IP;
                 else
                    tps->forwarding_option = NO_IP_OVER_IP;

                 if(sfi->i2r_flow != NULL)
                    tps->next_hb_addr = get_nexthb_addr(sfi->i2r_flow, flag);

                 if(process_tcp_ack(sfi, skb, flag) == -1)
                    goto err_ack_proc;

                 /* Checking for window scaling option */
                 if(tp.wscale_ok == 1) {
                    tps->wscale_ok = 1;
                    tps->snd_wscale = tp.snd_wscale;
                    tps->wnd_clamp = 1048576U;
                 }
                 else {
                    tps->rcv_wscale = 0;
                    tps->wnd_clamp = 65535U;
                 }

                 /* Checking for ECN capability and changing to RED qdisc */
                 if(((iph->tos & 0x01) || (iph->tos & 0x02)) ||
                    (th->ece && !th->cwr)) {
                    tps->ecn_capable = 1;
                 }
                 else
                    tps->ecn_capable = 0;

                 /* Forward the SYN-ACK */
                 prepare_tcp_synack(sfi, flag);

                 tcp_init_mss_wnd(fwd_tps, fwd_tps->snd_mss, th->window);
                 tps->rcv_mss = fwd_tps->snd_mss;

                 /* Send back an ACK */
                 prepare_tcp_ack(sfi, th, skb->dev, flag);
              }
              /* Last packet of handshake */
              else if(tps->state == TCP_SYNA_SENT) {
                 if(process_tcp_ack(sfi, skb, flag) == -1) {
                    goto err_ack_proc;
                 }
              }
```

```
        else if(tps->state == TCP_CLOSING) {
            /* Got last ACK of 3-way connection termination
             * handshake. Connection terminated. Delete sfi node */
            if(flag == PREV_FLOW) {
                if(get_queue_pkt_count(sfi->i2r_queue) > 0)
                    prepare_fwd_data(data, flag);
            }
            else if(flag == FWD_FLOW) {
                if(get_queue_pkt_count(sfi->r2i_queue) > 0)
                    prepare_fwd_data(data, flag);
            }
        }
        /* ACK of data packet */
        else if(tps->state == TCP_CONNECTED ||
                tps->state == TCP_FIN_WAIT  ||
                tps->state == TCP_FIN_RCVD  ||
                tps->state == TCP_FIN_SENT) {
            if(process_tcp_ack(sfi, skb, flag) == -1)
                goto err_ack_proc;

            /* Checking if we can piggyback the ACK with data packet */
            if(flag == PREV_FLOW) {
                /* Check if the ACK can be piggybacked on a
                 * data packet */
                if(get_queue_pkt_count(sfi->r2i_queue) > 0) {
                    if(prepare_fwd_data(data, FWD_FLOW) == -1 &&
                        (tps->end_seq - ntohl(th->seq)) > 0 )
                        prepare_tcp_ack(sfi, th, skb->dev, flag);
                }
                else if(tps->state == TCP_CLOSING)
                    prepare_tcp_finack(sfi, th, skb->dev, 0, flag);
                else if(tps->state == TCP_FIN_RCVD)
                    tps->state = TCP_CLOSING;
                else
                    prepare_tcp_ack(sfi, th, skb->dev, flag);

                /* Check if we can fwd any data */
                if( ((fwd_tps->snd_next - fwd_tps->snd_una) <
                                                fwd_tps->rcv_wnd) &&
                     (get_queue_pkt_count(sfi->i2r_queue) > 0) )
                    prepare_fwd_data(data, flag);
            }
            else if(flag == FWD_FLOW) {
                if(get_queue_pkt_count(sfi->i2r_queue) > 0) {
                    if(prepare_fwd_data(data, PREV_FLOW) == -1 &&
                    (tps->end_seq - ntohl(th->seq)) > 0 )
                    prepare_tcp_ack(sfi, th, skb->dev, flag);
                }
                else if(tps->state == TCP_CLOSING)
                  prepare_tcp_finack(sfi, th, skb->dev, 0, flag);
                else if(tps->state == TCP_FIN_RCVD)
                  tps->state = TCP_CLOSING;
                else
                  prepare_tcp_ack(sfi, th, skb->dev, flag);

                /* Check if we can fwd any data */
                if( ((fwd_tps->snd_next - fwd_tps->snd_una) <
                                              fwd_tps->rcv_wnd) &&
                  (get_queue_pkt_count(sfi->r2i_queue) > 0) )
                  prepare_fwd_data(data, flag);
              }
            }
          }
          else if(th->syn) {
            /* Copy in forward queue */
            if(tps->state == TCP_LISTEN) {
              if(flag == PREV_FLOW)
                enqueue_packet(sfi, skb, sfi->i2r_queue, tps);
              else if(flag == FWD_FLOW)
                enqueue_packet(sfi, skb, sfi->r2i_queue, tps);

              /* Check if we need to use the default MSS */
              if(tp.mss_clamp != NULL)
                fwd_tps->mss_clamp = tp.mss_clamp;
              else
                fwd_tps->mss_clamp = 536;

              tps->dup_ack_cnt = 0;
              fwd_tps->dup_ack_cnt = 0;
              tps->state = TCP_SYN_RCVD;

              /* Check if packet is coming from source or previous
               * helper box */
              if((ntohs(skb->nh.iph->frag_off) & IPIP_FRAG_OP) ==
                                                    IPIP_FRAG_OP)
                tps->forwarding_option = IP_OVER_IP;
              else
                tps->forwarding_option = NO_IP_OVER_IP;

              if(sfi->i2r_flow != NULL)
```

```
                tps->next_hb_addr = get_nexthb_addr(sfi->i2r_flow, flag);

             /* Checking for window scaling option */
             if(tp.wscale_ok == 1) {
                tps->wscale_ok = 1;
                tps->snd_wscale = tp.snd_wscale;
             }

             /* Checking for ECN capability */
             if(((iph->tos & 0x01) || (iph->tos & 0x02)) ||
                (th->cwr && th->ece)) {
                tps->ecn_capable = 1;
             }
             else
                tps->ecn_capable = 0;

             /* Forward the SYN to [HB, dst] */
             prepare_tcp_syn(sfi, flag);

             tcp_init_mss_wnd(fwd_tps, fwd_tps->snd_mss, th->window);
             tps->rcv_mss = fwd_tps->snd_mss;
          }
       }
       else if(th->fin) {
          if(tps->state != TCP_FIN_SENT)
             tps->state = TCP_CLOSING;

          if(process_tcp_ack(sfi, skb, flag) == -1)
             goto err_ack_proc;

          if(tps->state == TCP_FIN_WAIT || tps->state == TCP_FIN_RCVD)
             prepare_tcp_ack(sfi, th, skb->dev, flag);
          else if(tps->state == TCP_CLOSING) {
             prepare_tcp_finack(sfi, th, skb->dev, 0, flag);
          }
          else if(tps->state == TCP_FIN_SENT) {
             if(!(tps->finack_retrans)) {
                tps->rcv_next += 1;
                tps->finack_retrans = 1;
             }

             /* Got FIN-ACK of FIN sent. Sending last ACK of handshake.
                Start TIME_WAIT period                                 */
             prepare_tcp_ack(sfi, th, skb->dev, flag);
             init_tw_timer(sfi, flag);
```

```

            if(timer_pending(&(tps->rto_timer)))
               del_timer_sync(&(tps->rto_timer));
            if(timer_pending(&(fwd_tps->rto_timer)))
               del_timer_sync(&(fwd_tps->rto_timer));

            tps->state = TCP_CLOSING;
         }
      }
      /* Done with the incoming skb. Free it */
      if(skb->dst) {
         if(atomic_read(&skb->dst->__refcnt) < 1)
            atomic_set(&skb->dst->__refcnt, 1);
      }

      if(skb) {
         kfree_skb(skb);
      }
      return;

   bad_packet:
      printk(KERN_ALERT "Bad packet format. Send reset \n");
          /* Will have to write my own function to send a RST.
             Check /ipv4/netfilter/ipt_REJECT.c:send_reset() */
          //send_reset(skb, 0);

   discard_it:
      printk(KERN_ALERT "Problem pulling TCP header in hook \n");
      return;
   err_ack_proc:
      if(skb->dst) {
         if(atomic_read(&skb->dst->__refcnt) < 1)
            atomic_set(&skb->dst->__refcnt, 1);
      }
      kfree_skb(skb);
      return;
   }

   /*
    * This is the hook main function
    */
   unsigned int ip_in_hook_filter(unsigned int hooknum,
                                  struct sk_buff **skb,
                                  const struct net_device *in,
                                  const struct net_device *out,
                                   int (*okfn)(struct sk_buff *))
{
   struct sk_buff *sb = *skb;
   struct iphdr *iph;
   struct tcphdr *tcph;
   struct split_flow_info *sfi;
   struct flow_detail *search_flow;
   __u32 d_addr, s_addr;
   int ihl;

   skb_linearize(sb, GFP_ATOMIC);

   iph = sb->nh.iph;

   /* Check if packet is tunnelled */
   if(iph->protocol == IPPROTO_IPIP) {
      if(flow_supported(iph->saddr, iph->saddr, 1)) {
         ihl = sb->nh.iph->ihl*4;
         sb->nh.raw = (struct iphdr *)(sb->data + ihl);
         __skb_pull(sb, ihl);
         iph = sb->nh.iph;
      }
   }
   tcph = (struct tcphdr *)(sb->data + iph->ihl*4);

   d_addr = iph->daddr;
   s_addr = iph->saddr;

   /* Check if packet is TCP and it belongs to destination pool */
   if(iph->protocol == IPPROTO_TCP) {
      if(flow_supported(s_addr, d_addr, 0)) {
         /* (src) -> (dst) */
         /* Get the header for the sfi linked list */
         if((slh = get_head_sfi()) == NULL)
            goto head_error;

         /* Get flow details from packet */
         search_flow = get_flow_details(sb);

         /* Search for the node in the linked list */
         if((sfi = search_sfi(slh, search_flow, PREV_FLOW)) == NULL) {
            if(tcph->syn) {
               /* Seeing the flow for first time.
                * Create a new node */
               sfi = create_sfinode(slh, sb);
```

```

                /* Get the MAC header details of the incoming
                 * packet */
                if(sb->mac.raw != NULL) {
                    get_hw_addr(sb, sfi, PREV_FLOW);
                }
                sfi->i2r_flow = get_flow_details(sb);
                sfi->r2i_flow = (struct flow_detail *)
                                    kmalloc(sizeof(struct flow_detail),
                                            GFP_ATOMIC);

                process_in_pkt(sb, sfi, PREV_FLOW);
            }
            else {
                /* Not SYN and no sfi node => send RST */
                prepare_tcp_reset(sb, sb->dev, 0);
                goto err_sfi_node;
            }
        }
        else {
            /* If SYN-ACK, get MAC details and flow details */
            if(tcph->syn) {
                /* Get the MAC header details of the incoming
                 * packet */
                if(sb->mac.raw != NULL) {
                    get_hw_addr(sb, sfi, PREV_FLOW);
                }
            }
            /* sfi contains a pointer to the flow node */
            process_in_pkt(sb, sfi, PREV_FLOW);
        }
        return NF_STOLEN;
    }
    else if(flow_supported(d_addr, s_addr, 0)) {
        /* (dst) -> (src) */
        /* Get the header for the sfi linked list */
        if((slh = get_head_sfi()) == NULL)
            goto head_error;

        /* Get flow details from packet */
        search_flow = get_flow_details(sb);

        if((sfi = search_sfi(slh, search_flow, FWD_FLOW)) == NULL) {
            if(tcph->syn) {
                /* Seeing the flow for first time.
```

```
                 * Create a new node */
                sfi = create_sfinode(slh, sb);

                /* Get the MAC header details of the incoming
                 * packet */
                if(sb->mac.raw != NULL) {
                   get_hw_addr(sb, sfi, FWD_FLOW);
                }
                sfi->r2i_flow = get_flow_details(sb);
                sfi->i2r_flow = (struct flow_detail *)
                                 kmalloc(sizeof(struct flow_detail),
                                         GFP_ATOMIC);

                process_in_pkt(sb, sfi, FWD_FLOW);
             }
             else {
                /* Not SYN and no sfi node => send RST */
                prepare_tcp_reset(sb, sb->dev, 0);
                goto err_sfi_node;
             }
          }
          else {
             /* If SYN-ACK, get MAC details and flow details */
             if(tcph->syn) {
                /* Get the MAC header details of the incoming
                 * packet */
                if(sb->mac.raw != NULL) {
                   get_hw_addr(sb, sfi, FWD_FLOW);
                }
             }
             process_in_pkt(sb, sfi, FWD_FLOW);
          }
          return NF_STOLEN;
       }
    }

    return NF_ACCEPT;

head_error:
   printk(KERN_ALERT "Got bad sfi_head. Dropping the packet\n");
err_sfi_node:
   printk(KERN_ALERT "sfi search failed. There should be a node \n");
   return NF_DROP;
}
```

```
/*
 * Initializing the hook module
 */
int init_module()
{
   /* Populating the nf_ip_in variable */
   nf_ip_in.hook = ip_in_hook_filter;
   nf_ip_in.hooknum = NF_IP_PRE_ROUTING;
   nf_ip_in.pf = PF_INET;
   nf_ip_in.priority = NF_IP_PRI_FIRST;

   /* Initializing the sfi linked list */
   printk(KERN_ALERT "\n\n\n\n\n\n");
   printk(KERN_ALERT "Going to initialize sfi linked list\n");
   init_head_sfi();

   /* Registering the hook */
   nf_register_hook(&nf_ip_in);

   populate_nw_table();

/*
   prev_sch = kmalloc(size, GFP_KERNEL);
   if (!prev_sch)
      printk(KERN_ALERT "Error during kmalloc of Qdisc \n");

   fwd_sch = kmalloc(size, GFP_KERNEL);
   if (!fwd_sch)
      printk(KERN_ALERT "Error during kmalloc of Qdisc \n");
*/
   return 0;
}

/*
 * Cleaning up
 */
void cleanup_module()
{
   printk(KERN_ALERT "Freeing up space from the linked list \n");
   cleanup_sfi_list(slh);
   nf_unregister_hook(&nf_ip_in);
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Rahul");
```

REFERENCES

[1] T. Ott, J. H. B. Kemperman, and M. Mathis, "The stationary behavior of ideal TCP congestion avoidance," August 1996.

[2] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation," in *ACM SIGCOMM*, September 1998.

[3] "The PingER Project," WWW, http://www-iepm.slac.stanford.edu/pinger, retrieved in February 2007.

[4] J. P. Martin-Flatin and S. Ravot, "TCP Congestion Control in Fast Long-Distance Networks," California Institute of Technology, Tech. Rep., July 2002, technical Report CALT-68-2398.

[5] A. Bakre and B. R. Badrinath, "I-TCP: Indirect TCP for Mobile Hosts." in *Proceedings of the 15th ICDCS*, June 1995, pp. 136–143.

[6] S. Koppartyi, S. V. Krishnamurthy, M. Faloutsos, and S. K. Tripathi, "Split TCP for Mobile Ad Hoc Networks." in *IEEE GLOBECOM*, 2002.

[7] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz, "Improving TCP/IP performance over Wireless Networks," in *ACM Conference on Mobile Computing and Networks*, November 1995.

[8] K. Brown and S. Singh, "M-TCP: TCP for mobile cellular networks," *ACM Computer Communication Review*, vol. 27, no. 5, 1997.

[9] Z. J. Haas and P. Agrawal, "Mobile-TCP: An Asymmetric Transport Protocol Design for Mobile Systems," in *Proc. IEEE ICC' 97*, June 1997.

[10] V. Jacobson, R. Braden, and D. Borman, *TCP Extension for High Performance.*, May 1992, RFC 1323.

[11] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, *TCP Selective Acknowledgement Options.*, April 1996, RFC 2018.

[12] S. Oueslati-Boulahia, A. Serhrouchni, S. Tohm, S. Baier, and M. Berrada, "TCP Over Satellite Links: Problems and Solutions," *Telecommunication Systems*, pp. 199 - 212, 2000.

[13] M. Allman, D. Glover, and L. Sanchez, *Enhancing TCP over Satellite Channels using Standard Mechanisms.*, January 1999, RFC 2488.

[14] K. K. Ramakrishnan, S. Floyd, and D. Black, *The Addition of Explicit Congestion Notification (ECN) to IP.*, September 2001, RFC 3168.

[15] V. Tsaoussidis and I. Matta, "Open Issues on TCP for Mobile Computing," *Journal of Wireless Communication and Mobile Computing*, February 2002.

[16] R. Durst, P. Feighery, and K. Scott, "Why not use the Standard Internet Suite for the Interplanetary Internet?" WWW, http://www.ipnsig.org/reports/TCP_IP.pdf, 2002, MITRE White Paper, retrieved in March 2007.

[17] R. Yavatkar and N. Bhagawat, "Improving End-to-End Performance of TCP over Mobile Internetworks," in *Mobile 94 Workshop Mobile Computing Syst Appl*, December 1994.

[18] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby, *Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations.*, June 2001, RFC 3135.

[19] W. Wei, C. Zhang, H. Zang, J. Kurose, and D. Towsley, "Inference and Evaluation of Split-connection Approaches in Cellular Data Networks," Department of Computer Science, University of Massachusetts, Amherst, Tech. Rep., 2006.

[20] T. Ott, "Transport Protocols in the TCP Paradigm and their Performance," *Telecommunication Systems*, vol. 30:4, pp. 351 – 385, 2005.

[21] S. Kent, *IP Authentication Header.*, December 2005, RFC 4302.

[22] S. Kent and K. Seo, *Security Architecture for the Internet Protocol.*, December 2005, RFC 4301.

[23] A. D. Maltz and P. Bhagwat, "TCP Splicing for Application Layer Proxy Performance," *Journal of High Speed Networks*, vol. 8, pp. 235–240, 1999.

[24] R. Yates, D. Raychaudhuri, S. Paul, and J. Kurose, "Postcards from the Edge: A Cache-and-Forward Architecture for the Future Internet," Future Internet Design (FIND) Project Cluster, WINLAB, Rutgers University, Tech. Rep., 2006.

[25] D. Bovet and M. Cesati, *Understanding the Linux Kernel.* O'Reilly, 2005.

[26] C. Benvenuti, *Understanding Linux Network Internals.* O'Reilly, 2005.

[27] "Linux Advanced Routing & Traffic Control," WWW, http://www.lartc.org/, retrieved in May 2007.

[28] "Netfilter talk by LaForge," WWW, http://kernelnewbies.org/Documents/Netfilter, retrieved in May 2007.

[29] S. Floyd, T. Henderson, and A. Gurtov, *The NewReno Modifications of TCP's Fast Recovery Algorithm.*, April 2004, RFC 3782.

[30] "Linux Test Project," WWW, http://ltp.sourceforge.net/, retrieved in February 2007.

[31] "The Linux Kernel Archives," WWW, http://www.kernel.org/, retrieved in May 2007.

[32] J. Postel., *Transmission Control Protocol.*, September 1981, RFC 793.

[33] V. Jacobson., "Congestion Avoidance and Control." in *SIGCOMM 88*, August 1988.

[34] M. Carson and D. Santay, "NistNet - a Linux-based Network Emulation Tool," in *Computer Communication Review*, June 2003.

[35] "tcpdump," WWW, http://www.tcpdump.org/, retrieved in May 2007.

[36] "GNU Binary Utilities: objdump," WWW, http://www.gnu.org/software/binutils/manual/html_chapter/binutils_4.html, retrieved in July 2007.

[37] K. Wehrle, F. Pählke, H. Ritter, D. Müller, and M. Bechler, *The Linux Networking Architecture.* Prentice Hall, 2005.

[38] R. Love, *Linux Kernel Development.* Novell Press, 2005.

[39] A. B. Forouzan, *TCP/IP Protocol Suite.* McGraw Hill, 2006.

[40] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers.* O'Reilly, 2005.

[41] "OProfile - a System Profiler for Linux," WWW, http://oprofile.sourceforge.net/, retrieved in July 2007.

[42] "Tuning with OProfile," WWW, http://people.redhat.com/wcohen/FedoraCore2OProfileTutorial.txt, retrieved in July 2007.

[43] M. Allman, V. Paxson, and W. Stevens, *TCP Congestion Control.*, April 1999, RFC 2581.

[44] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, *An Extension to the Selective Acknowledgement (SACK) Option for TCP.*, July 2000, RFC 2883.

[45] S. Floyd, "TCP and Explicit Congestion Notification." *ACM Computer Communication Review*, vol. V. 24 N. 5, pp. 10–23, October 1994.

[46] J. Wong and V. Leung, "Improving End-to-End Performance of TCP Using Link-Layer Retransmissions over Mobile Internetworks." in *Proceedings of ICC*, 1999, pp. 324–328.

[47] M. Liu and N. Ehsan, "Modeling TCP Performance with Proxies," *Journal of Computer Communications special issue on Protocol Engineering for Wired and Wireless Networks*, vol. Vol 27, pp. 961–975, June 2004.

[48] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A comparison of mechanisms for improving TCP performance over wireless links," *IEEE/ACM Transactions on Networking*, vol. Vol 5, pp. 756–759, December 1997.

[49] D. C. Feldmeier, A. J. McAuley, J. M. Smith, D. S. Bakin, W. S. Marcus, and T. M. Raleigh, "Protocol Boosters," *IEEE Journal on Special Aspects of Communication*, 1998.

[50] I. . WG, *Part 11:Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, Standard Specification, IEEE, 1999.

[51] C. Barakat, E. Altman, and W. Dabbous, "On TCP Performance in a Heterogeneous Network: A Survey," *IEEE Communications Magazine*, January 2000.

[52] R. Jain and T. Ott, "Design and Implementation of Split TCP in the Linux Kernel," in *IEEE, Globecom*, 2006.

[53] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end Arguements in System Design," *ACM Transactions on Computer Science*, vol. 2, pp. 277 - 288, 1984.