IMPROVING PERFORMANCE OF SMTP RELAY SERVERS

AN IN-KERNEL APPROACH

by

MAYURESH KASTURE

(Under the Direction of Kang Li)

ABSTRACT

With continuous increase in average size and number of e-mails on the Internet, load on SMTP servers is increasing. This work attempts to measure the improvement in latency and throughput of SMTP relay servers (focusing on CPU activities), that can be obtained by avoiding the expensive system call mechanism and user space-kernel space memory copy. We implement SMTP relay server as a kernel module to be able to call kernel functions (to avoid system calls) and to relay network buffers directly (to avoid memory copy). The current Linux kernel does not allow kernel modules to receive or send network buffers. Hence, we modified the Linux networking layer to incorporate such a framework. An equivalent user mode SMTP relay server was developed to compare the performances. This thesis discusses the details of this approach and presents the experiments conducted and the results obtained.

INDEX WORDS: Zero copy, Kernel system calls, SMTP, Linux kernel, Linux networking, TCP/IP

IMPROVING PERFORMANCE OF SMTP RELAY SERVERS

AN IN-KERNEL APPROACH

by

MAYURESH KASTURE

B.E., Sinhgad College of Engineering, India, 2005

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2010

IMPROVING PERFORMANCE OF SMTP RELAY SEVERS

AN IN-KERNEL APPROACH

by

MAYURESH KASTURE

Major Professor:	Kang Li
Committee:	Lakshmish Ramaswamy Shelby Funk

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2010

ACKNOWLEDGEMENTS

This work owes its success to the persuasion of Dr. Kang Li, who with his valuable guidance and timely suggestions really helped give this project its current shape. The thesis would not have been possible without his support and encouragement.

I would also like to thank my committee members Dr. Shelby Funk and Dr. Lakshmish Ramaswamy for their time and cooperation. Last but certainly not the least, I would like to acknowledge and appreciate the efforts of all the other staff/faculty members, who both directly and indirectly influenced this work.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

This chapter starts by presenting the idea behind this work. It explains how the system call mechanism and memory copy can degrade performance of a user mode application. The next sections give the precise problem definition and the approach we adopted to tackle the problem. Finally, it mentions the related work and gives outline of the remaining thesis.

1.1 Background And Motivation

Now-a-days, e-mails are being used massively. Since high bandwidth Internet is becoming available to practically everyone, use of multimedia in e-mails is increasing. This is resulting in increase in not only number of e-mails on the Internet but also average size of an e-mail. Hence, large e-mail providers and enterprises have to do considerable investment into SMTP (Simple Mail Transfer Protocol) servers. These e-mail providers and enterprises generally have few public SMTP relay servers. E-mails from outside are targeted to these relay servers. The servers then filter and sort the incoming e-mails and relay them to respective internal SMTP servers. By increasing their throughput, the number of servers required can be reduced. This work attempts to measure the improvements in latency and throughput that can be achieved by implementing an SMTP relay server in kernel for avoiding kernel-user mode switches and memory copy.

1.1.1 Kernel-User Mode Switch

To protect the system from users' mistakes and evildoings, every operating system employs many protection mechanisms like address spaces, access rights etc. One of them, and a very important one, is kernel mode-user mode distinction. Processes running at kernel mode enjoy unlimited privileges while processes running at user mode have restrictions. Hence, some tasks can be performed only by kernel mode processes e.g. reading/writing data from/to disk. To allow user mode processes to safely leverage these kernel functionalities, system call mechanism is employed.

But, system calls are more expensive than normal function calls since they involve heavy user mode-kernel mode transition[1][2]. When a system call is invoked by a user mode process, a software interrupt is initiated. The number identifying the system call and the corresponding parameters are stored in registers. The state of the user mode process is saved and then the control is transferred to the kernel function indicated by the number stored in the EAX register (in the case of x86 architecture).

1.1.2 Memory Copy

As a part of protection mechanisms mentioned in 1.1.1, memory range allocated to kernel mode processes and user mode processes is different. This prevents kernel memory corruption, accidental or otherwise, by user mode processes; But, necessitates an extra memory copy (from user space to kernel space and/or vice versa) when data is to be received from or sent to any peripheral or network device. If CPU (Central Processing Unit) is freed from this task of memory copy, performance of processes which involve heavy memory copy can be improved significantly[3].

1.2 Problem Definition

Widely used user mode SMTP servers face both the aforementioned problems, i.e., performance degradation due to system calls and memory copy.

System Calls: When an SMTP server is implemented at user mode, it needs to make system calls to establish communication and to transfer data to/from the other end. This results in performance degradation because of heavy user mode-kernel mode switch (required by the system call mechanism). If SMTP server is implemented in kernel, it can directly use the kernel functions and that helps in improving performance.

Memory Copy: Functionality of an SMTP relay server consists of three main steps: e-mails are received from senders, they undergo minimal processing for sorting and filtering and then they are sent to receivers. In this process, memory copy becomes a significant operation. As average e-mail size increases, memory copy consumes a considerable chunk of overall processing time. Thus, performance of SMTP relay servers can be improved by eliminating memory copy. User mode relay servers cannot avoid it. They have to follow the path shown in figure 1 – (a). But, kernel mode servers can (as shown in figure 1 – (b)).

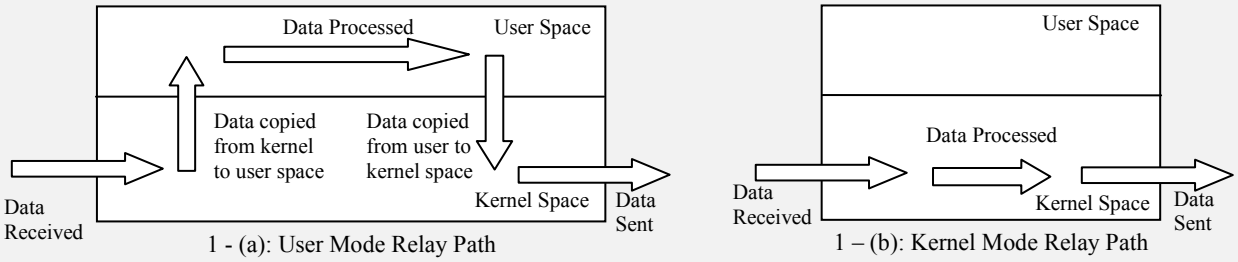


Figure 1: User Mode And Kernel Mode Data Relay Paths

Hence, performance of SMTP relay servers can be improved if they are moved from user mode to kernel mode.

The aim of this work is to measure the improvement in latency and throughput of kernel mode SMTP relay servers over that of user mode SMTP relay servers, achieved by avoiding memory copy and system calls, focusing mainly on CPU activities.

1.3 Approach And Method

The goal was approached following these steps:

Linux Kernel Modification: The Linux kernel does not have any mechanism with which a kernel module can receive or send a network buffer directly. Like user mode processes, it has to pass a memory buffer in which data of required size is copied. As a result, SMTP servers cannot really avoid memory copy with the current Linux kernel, even if it is implemented in kernel. Thus, first step was to understand the Linux network layer and modify the kernel to provide such a mechanism. Once this mechanism was in place, it was possible to write kernel mode servers that could send and receive network buffers directly and, thus, avoid copying data from these buffers to a user supplied buffer. The kernel modified as a part of this work is Linux 2.6.28.8.

SMTP Relay Server Implementations: To compare the performances of user mode and kernel mode SMTP relay servers, they needed to be implemented. After reading and understanding RFCs (Request For Comments) related to SMTP, both the servers were implemented. These servers are identical in code structure and full attempts have been made to make sure that no server gets undue advantage of anything except memory copy and system call avoidance.

Testing And Benchmarking: Once all the implementations were done, an experiment was conducted to measure the improvement in performance. The experiment compared latencies

and throughputs of both the SMTP servers (kernel mode and user mode). The test was conducted for e-mails of different sizes to see the effect of e-mail sizes on the improvement.

1.4 Related Work

kHTTPd (Kernel HTTP Daemon) and TUX (Threaded linUX http layer) are the web servers that run in kernel mode for performance improvement. It was noticed that a large amount of HTTP (Hypertext Transfer Protocol) traffic contained static data. For a web server, serving static data is nothing but just reading the corresponding file and transferring it over the network. No processing is required. This functionality can be safely and efficiently implemented in kernel. Efficiency improvement results from avoiding memory copy and system calls. This observation gave rise to kHTTPd that serves only static data and depends on user mode web servers like Apache or Zeus for serving dynamic data[4][5]. TUX goes a step further and performs network layer data caching to serve static data faster. TUX also accelerates dynamic data generation by invoking CGI (Common Gateway Interface) scripts directly from the kernel[6].

1.5 Outline

Chapter 2 describes the Linux network architecture. It first describes important data structures. Then, it gives an overview of how packets traverse up and down the Linux TCP (Transmission Control Protocol)/IP (Internet Protocol) stack followed by detailed explanation of TCP level send/receive mechanisms. Chapter 3 presents the implementation in detail. The first part of this chapter explains all the changes and additions to the Linux kernel 2.6.28.8, required to provide a mechanism that can be used by kernel mode relay servers to avoid memory copy. And second part describes implementation details of SMTP relay servers. Chapter 4 describes the setup used for the test. Results are compared and discussed. Chapter 5 presents the conclusion and explains how this work can be extended.

CHAPTER 2

THE LINUX NETWORK ARCHITECTURE

This chapter starts by mentioning and briefly explaining some important data structures that are used in Linux network architecture[7][9][11]. Then, it gives an overview of packets paths, up and down the Linux TCP/IP stack[7][8][9][10][11]. Finally, it explains the TCP receive and send processes in detail[7][8][10][11].

2.1 Important Data Structures

2.1.1 sk_buff

sk_buff is the most important data structure in Linux networking code. It is a network buffer holding a single packet of data being sent from/received by the machine. The network buffer has two parts: *sk_buff* data structure (which is metadata for the data packet being sent or received) and data block (linear and/or non-linear). Linear data block has space for headers as well as data. It is generally a page long. If more space is required for data, it is stored in paged fragments, called non-linear data blocks. Some important fields of *sk_buff* structure are,

***next*, *prev*:** Every *sk_buff* ends up in some or the other queue. These queues are implemented as doubly linked lists. *next* and *prev* are the pointers (pointing to the next and the previous *sk_buff* in the list respectively) that help to maintain the lists of *sk_buff*s.

***sk*:** *sk* is a pointer to the *sock* data structure of the socket which current *sk_buff* belongs to. It is NULL, when the node is neither source nor final destination for the packet.

***len*, *data_len*, *truesize*:** These fields are related to size of a network buffer. *len* is size of the data (linear and non-linear). *data_len* is size of the non-linear data part only. *truesize* is size of the whole buffer (i.e. size of *sk_buff* structure plus size of data part)

***cloned*:** Whenever a user, processing a network buffer shared between multiple users, needs to modify the *sk_buff* structure without modifying the data, it is more efficient to create another *sk_buff* pointing to same data block. This is called cloning a buffer. *cloned* bit is used to denote whether or not the network buffer is cloned. It is set to 1 in both the structures (original and cloned), if an *sk_buff* is cloned.

```
struct sk_buff
{
        struct sk_buff *next;
        struct sk_buff *prev;
        struct sock *sk;
        ...
        unsigned int len,  data_len;
        ...
        __u8..., cloned: 1, ...;
        ...
        void (*destructor)(struct sk_buff *skb);
        ...
        sk_buff_data_t tail;
        sk_buff_data_t end;
        unsigned char *head, *data;
        unsigned int truesize;
        atomic_t users;
};
```

Figure 2: sk_buff – Linux Network Buffer

***destructor*:** *destructor* is a pointer to the function that is called when the *sk_buff* structure is being released. Generally, this function is responsible for memory accounting.

***head*, *data*, *tail*, *end*:** These pointers point to the borders of linear data part. Linear data part of a network buffer is allocated to accommodate not only actual data but also headers at

various network layers. Hence, it is larger than actual data stored. *head* and *end* point to the start and end of the linear data block whereas *data* and *tail* point to the start and end of the data.

***users*:** *users* is a reference count for *sk_buff* structure (but, not for the data block of the buffer)

2.1.2 skb_shared_info

skb_shared_info stores the information about the non-linear data block of a network buffer. It is stored at the end of the linear data block. Thus, it is accessed using macro defined as,

*#define skb_shinfo(SKB) ((struct skb_shared_info *)(skb_end_pointer(SKB)))*

The important fields of *skb_shared_info* structure are,

```
struct skb_shared_info
{
        atomic_t dataref;
        unsigned short nr_frags;
        ...
        struct sk_buff *frag_list;
        ...
        skb_frag_t frags[MAX_SKB_FRAGS];
};
```

Figure 3: skb_shared_info

***dataref*:** *dataref* is a reference count for the data block of a network buffer.

***nr_frags, frags*:** These fields store information about non-linear data block of a network buffer. Non-linear data block is stored as paged fragments. *nr_frags* denotes number of page fragments and *frags* is an array with *nr_frags* elements, each of which is a structure *skb_frag_struct.* This structure stores information (page address, data offset and data size) about a single paged fragment.

***frag_list*:** *frag_list* is used to handle IP fragmentation. It points to a list of *sk_buff*s representing fragments of the original packet (which *frag_list* belongs to).

2.2 TCP/IP Send/Receive Paths

This section tries to give an overview of TCP/IP/Ethernet control flow in Linux 2.6.28 network stack. The focus is on data reception and transmission processes. Control flow during connection establishment and tear down is not covered. When data is received from another host or it is sent to another host, which functions process the packets and how packets climb up and down the network stack, is explained.

2.2.1 TCP/IP Receive Path

On reception of a packet by NIC (Network Interface Card), it is first DMAed (Direct Memory Access) into ring buffer[1]. When the packet is completely copied to the ring buffer, NIC receive interrupt is generated. The interrupt handler first creates a network buffer and copies the frame into it. Then, it sets *protocol* field of the network buffer to indicate the next protocol layer to which the packet belongs (ETH_P_IP, for IP) and prepares it for the link level use (e.g. sets data pointer to the start of the network layer header), by calling *eth_type_trans()*. The network buffer is then queued into CPU specific input queue (*softnet_data->input_pkt_queue*) by calling *netif_rx()*. *softnet_data* is a per-cpu variable (i.e. every CPU has a different instance of this variable). This eliminates the need for synchronization among different CPUs while manipulating it. *netif_rx()* first calls *napi_schedule()* to schedule a poll routine by queuing the device associated with current CPU's input queue (*softnet_data->backlog*)[2] into CPUs poll list (*softnet_data->poll_list*), if it is not already scheduled. *napi_schedule_prep()* checks if poll routine can be scheduled; and *__napi_schedule()* actually schedules the routine and raises

[1] Ring buffer is a circular queue of network buffers, maintained by net device drivers. There are separate ring buffers for reception and transmission.

[2] This discussion is focused on non-NAPI devices because even today most of the drivers have not adapted to NAPI framework. NAPI devices do not use common CPU device for scheduling poll routine. They enqueue their own devices instead. Also, they do not use common CPU input queue for incoming packets but they manage their own queues. However, poll list is shared between NAPI devices and non-NAPI devices.

NET_RX_SOFTIRQ. *netif_rx()* then queues the network buffer into CPU's input queue for deferred processing and interrupt handler returns. The processing of network buffer is an expensive operation and should not be done in interrupt handler. Thus, the processing is deferred.

Function registered with *NET_RX_SOFTIRQ* is *net_rx_action()*. Thus, it is invoked when softIRQ processing starts. It dequeues each device from the CPUs poll list and calls its poll function. *softnet_data->backlog*'s poll routine is initialized to *process_backlog()*. *process_backlog()* dequeues the packets queued onto CPUs input queue, one by one and sends them to *netif_receive_skb()* for further processing.

netif_receive_skb() sends packets to corresponding network layer handlers. It sends a copy of the buffer to each handler that is interested in packets with the current protocol and received at current network interface. So, along with the protocol specific handler, every sniffer that is interested in all the packets gets a copy of this buffer. *protocol* field initialized during interrupt handler is used in this function. The handler corresponding to IP is *ip_rcv()*.

ip_rcv() drops the packet if it is not destined for this host (i.e. if the interface has received the packet because it is in promiscuous mode). Then, it checks if the datagram is acceptable by doing following sanity tests,

- Length of the header is at least 20 bytes (*iph->ihl >= 5*)
- Length of the datagram is at least equal to the size of the header (*pskb_may_pull()*)
- IP Version is 4 (*iph->version == 4*)
- Checksum is correct (*ip_fast_csum()*)
- Length field in the datagram is not bogus (*skb->len >= iph->tot_len; iph->tot_len >= (iph->ihl*4)*)

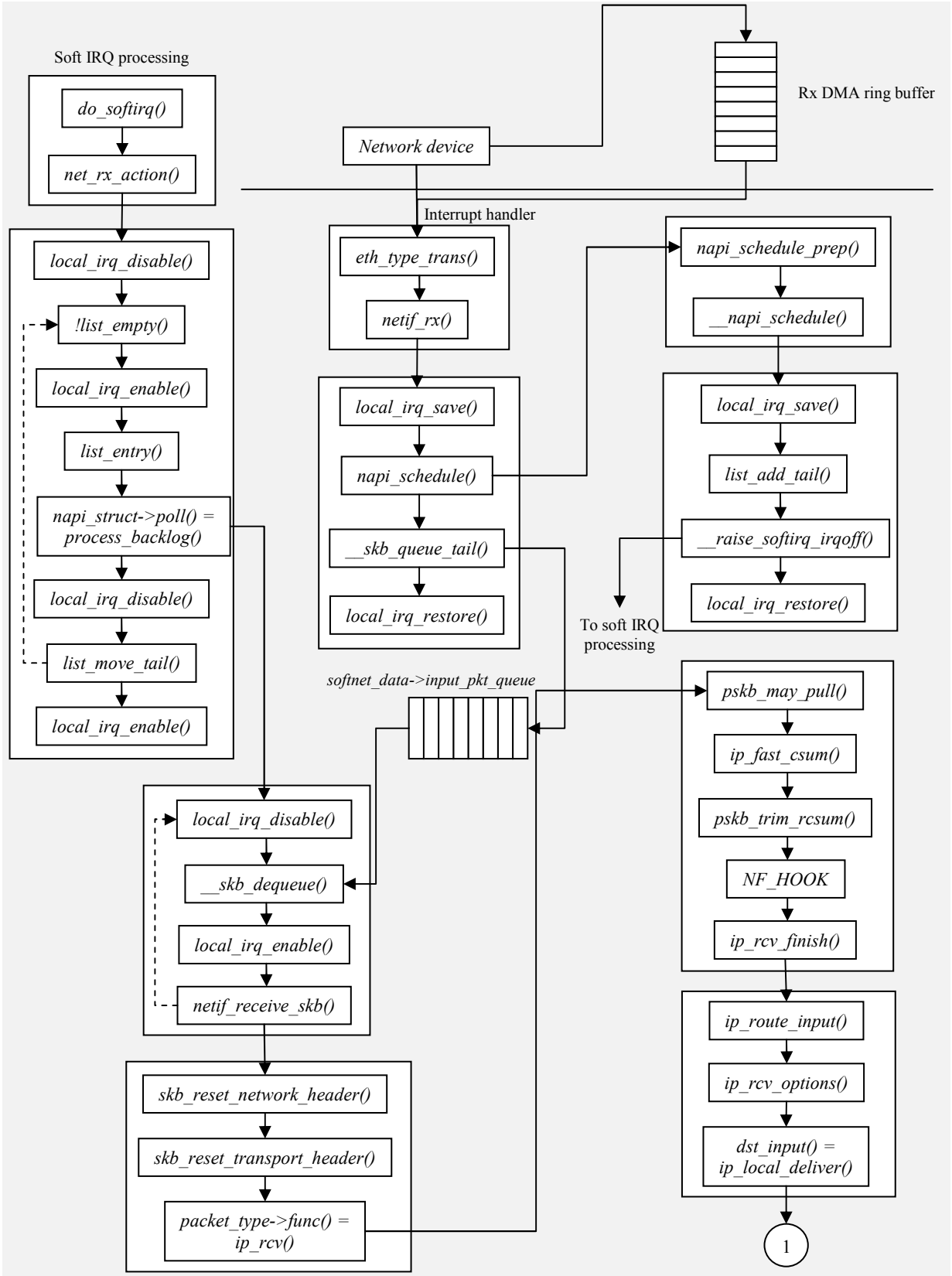


Figure 4 – (a): TCP/IP Receive Control Flow In Linux Kernel 2.6.28
(Original figure source: [7], with modifications to fit it for 2.6.28 kernel)

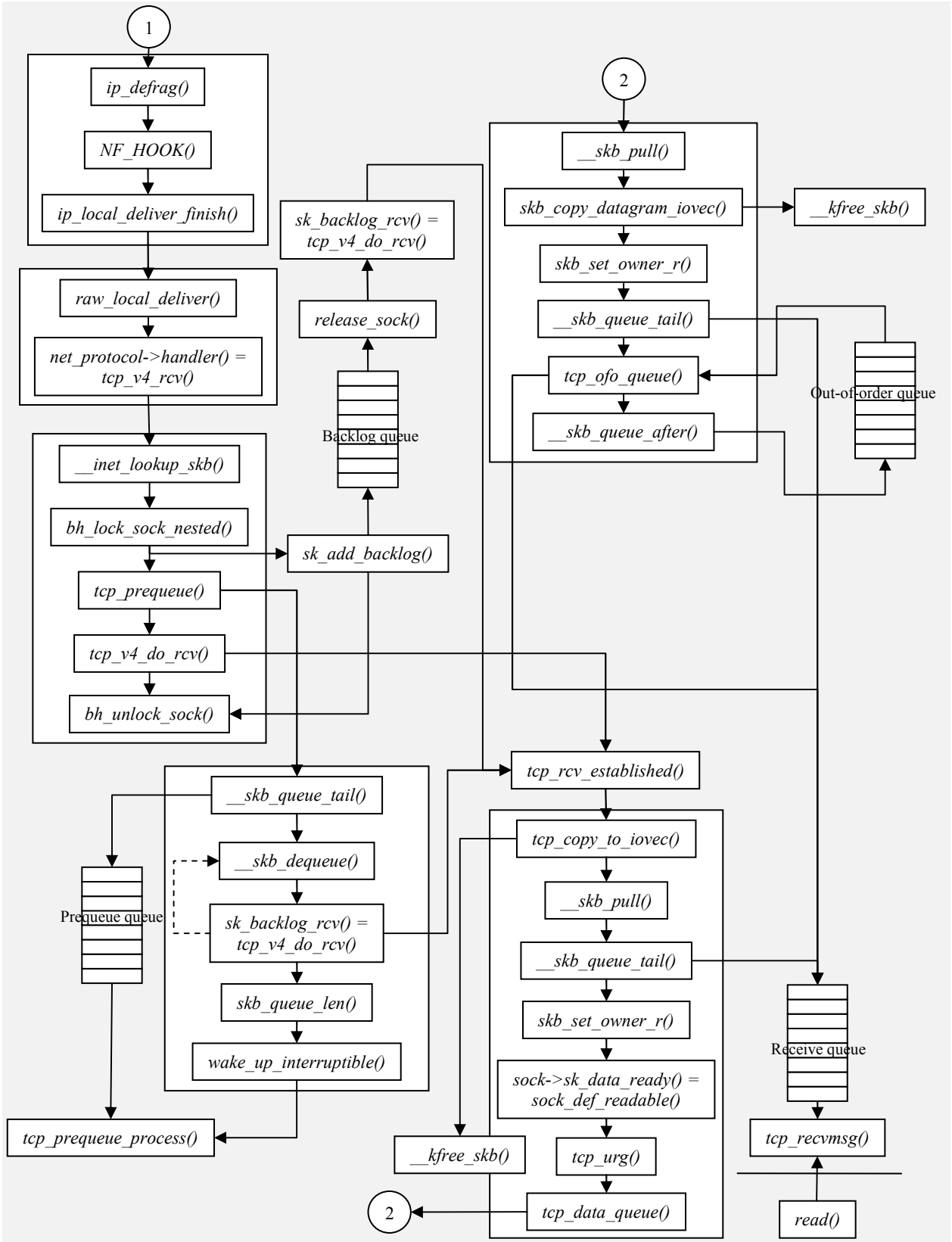


Figure 4 – (b): TCP/IP Receive Control Flow In Linux Kernel 2.6.28
(Original figure source: [7], with modifications to fit it for 2.6.28 kernel)

Once the datagram is accepted, it removes any padding that may have been added by the transport medium (*pskb_trim_rcsum()*). The packet is then sent through netfilter hook *NF_INET_PRE_ROUTING*. After pre-routing netfilter processing is done, control is passed to the callback routine, *ip_rcv_finish()*.

ip_rcv_finish() deals mainly with routing. It calls *ip_route_input()* to find the route of the packet. Then, if size of the IP header is greater than 20 bytes, IP options are processed in *ip_rcv_options()*. Packet is passed further by calling *dst_input()*. Network buffer's *dst* field was set in *ip_route_input()*. If packet needs to be forwarded, then *dst->input()* is set to *ip_forward()*, which takes care of forwarding the packet. But, if packet is targeted for the current host *dst->input()* is set to *ip_local_deliver()*.

ip_local_deliver() checks if the packet is a fragment. If it is, it calls *ip_defrag()*. It searches for other fragments of the same packet. If all the fragments have arrived, the packet is constructed else the fragment is stored. At this point, if a complete packet, reconstructed or otherwise, is available, then that packet is screened through netfilter hook *NF_INET_LOCAL_IN*. Callback routine in this case is *ip_local_deliver_finish()*. This is the last function in IP layer. It first calls *raw_local_deliver()* to deliver copies of the network buffer to eligible raw sockets. Then, it passes the packet up the stack to TCP layer by calling *net_protocol->handler()* which is set to *tcp_v4_rcv()* in case of TCP.

Before we look at the control flow transfer at TCP layer, it is important to understand TCP queues. There are total 4 queues managed by TCP -

1. **Receive queue:** The receive queue contains only in-order, completely processed packets. This queue is ready to serve users' data requests.

2. **Backlog queue:** The backlog queue is used to enquque packets when the socket is in use. It is processed whenever socket lock is released.
3. **Prequeue queue:** Whenever possible, packets are queued to prequeue queue before processing them. Packets are then processed in process context.
4. **Out-of-order queue:** Packets which arrive out of order are stored in this queue temporarily. Whenever a hole is filled, packets from this queue are transferred to the receive queue.

Now, let us look at TCP control flow. *tcp_v4_rcv()* first searches for the socket corresponding to the packet (*__inet_lookup_skb()*). If the socket is locked, the packet is queued onto the backlog queue (*sk_add_backlog()*). Otherwise an attempt is made to queue the packet on prequeue queue (*tcp_prequeue()*). A packet can be queued on prequeue queue only if there is currently some process waiting for data. If this fails, the packet is processed right away (*tcp_v4_do_rcv()*). In *tcp_prequeue()*, after queuing the packet into prequeue queue, if amount of memory consumed by the packets queued is greater than threshold *sk->sk_rcvbuf*, all the packets are processed right away (in the interrupt context). In this case also, *tcp_v4_do_rcv()* is called. But, if this threshold is not exceeded, as soon as first packet is queued in the prequeue queue, the process waiting for data, is woken up (*wake_up_interruptible()*).

Since the connection is in established state, *tcp_v4_do_rcv()* calls *tcp_rcv_established()*. *tcp_rcv_established()* first tries to copy the data directly to user memory (*tcp_copy_to_iovec()*). This is possible in a specific combination of conditions (e.g. – data is in-order, corresponding process is currently running etc.). If data cannot be copied to user buffer but the packet is in-order, then it is queued onto receive queue[3]. If packet is not in order or urgent data handling is to

[3] Actually, only being in order is not sufficient. All the conditions required for the fast path processing should be satisfied. But, to keep the discussion simple, only one condition is mentioned.

be done[4], the packet cannot be queued on receive queue. In that case, *tcp_data_queue()* is called for complete processing (called as "slowpath" processing). *tcp_data_queue()* again tries to copy data from given packet to user buffer. If that fails, packet is attempted to be queued onto receive queue (*__skb_queue_tail()*). Then, the out-of-order queue is processed (*tcp_ofo_queue()*) because current packet might have filled a hole and that might allow some packets from out-of-order queue to go onto receive queue. If the current packet is out of order, it is queued on the out-of-order queue (*__skb_queue_after()*).

After all the processing, every packet ends up in the receive queue and from there, data is served (*tcp_recvmsg()*) as user requests arrive.

2.2.2 TCP/IP Send Path

All the function calls to transfer data over TCP end up in *tcp_sendmsg()* which is responsible for mainly creating network buffers for the data to be sent. It first checks if the last buffer enqueued in the TCP transmit queue, has any space available for the data. If there is, then first that space is utilized. If the last buffer is full and there is still data to be sent, a new network buffer is allocated by calling *sk_stream_alloc_skb()*. It allocates space for data as well as for TCP, IP and link layer headers. The space for headers is pre-allocated to avoid copying the buffer every time extra space is needed for the headers. This way, adding a protocol header can be done by just changing few pointers. All protocol layers work on the same copy of the network buffer. Once the buffer is allocated, *skb_entail()* is called for initialization of fields related to TCP header and then, to queue it into TCP transmit queue. This function initializes start and end sequence number to the next unwritten byte because at this point the buffer does not contain any data. When data is copied into it, end sequence number is adjusted accordingly. It also initializes TCP flags to ACK since TCP segment is supposed to carry minimum ACK flag. After data has

[4] Basically, anything that results in slow path processing.

been copied into the buffers, transmit process is initiated, for buffers queued in transmit queue, by calling *tcp_push_one()* / *__tcp_push_pending_frames()* / *tcp_push()*. All these functions ultimately end up in tcp_write_xmit().

tcp_write_xmit() performs all the tests to see if the segment can be transmitted immediately. It checks if,

- current congestion window size allows current packet to be transmitted
- current packet is inside send window
- Nagle test permits this packet to be sent
- there is urgent data to be sent

Based on the results of these tests, it decides if or not the packet can be transmitted. If some criterion is not satisfied, the transmission is deferred. Otherwise *tcp_transmit_skb()* is called to transmit the segments off the transmit queue.

tcp_transmit_skb() is a function that actually transmits segments. It is used for both; initial transmission and possible later retransmissions. It first creates a clone of the segment because original segment can be removed from the transmit queue only after acknowledgement has been received. Segments queued onto transmit queue are headerless till now. Hence, main job of this function is to assign TCP header. TCP level checksum is calculated by calling *tcp_v4_send_check()*. Once these clones are ready for transmission, they are passed to the network layer by calling, in case of TCP/IP, *ip_queue_xmit()*.

ip_queue_xmit() is the entry door of IP. When a connection is initialized, corresponding route is calculated and it is cached with the socket so that it does not have to be recalculated for every packet. *ip_queue_xmit()* first calls *__sk_dst_check()* to check if the cached route is still valid. If it is not, a new route is calculated by calling *ip_route_output_flow()*.

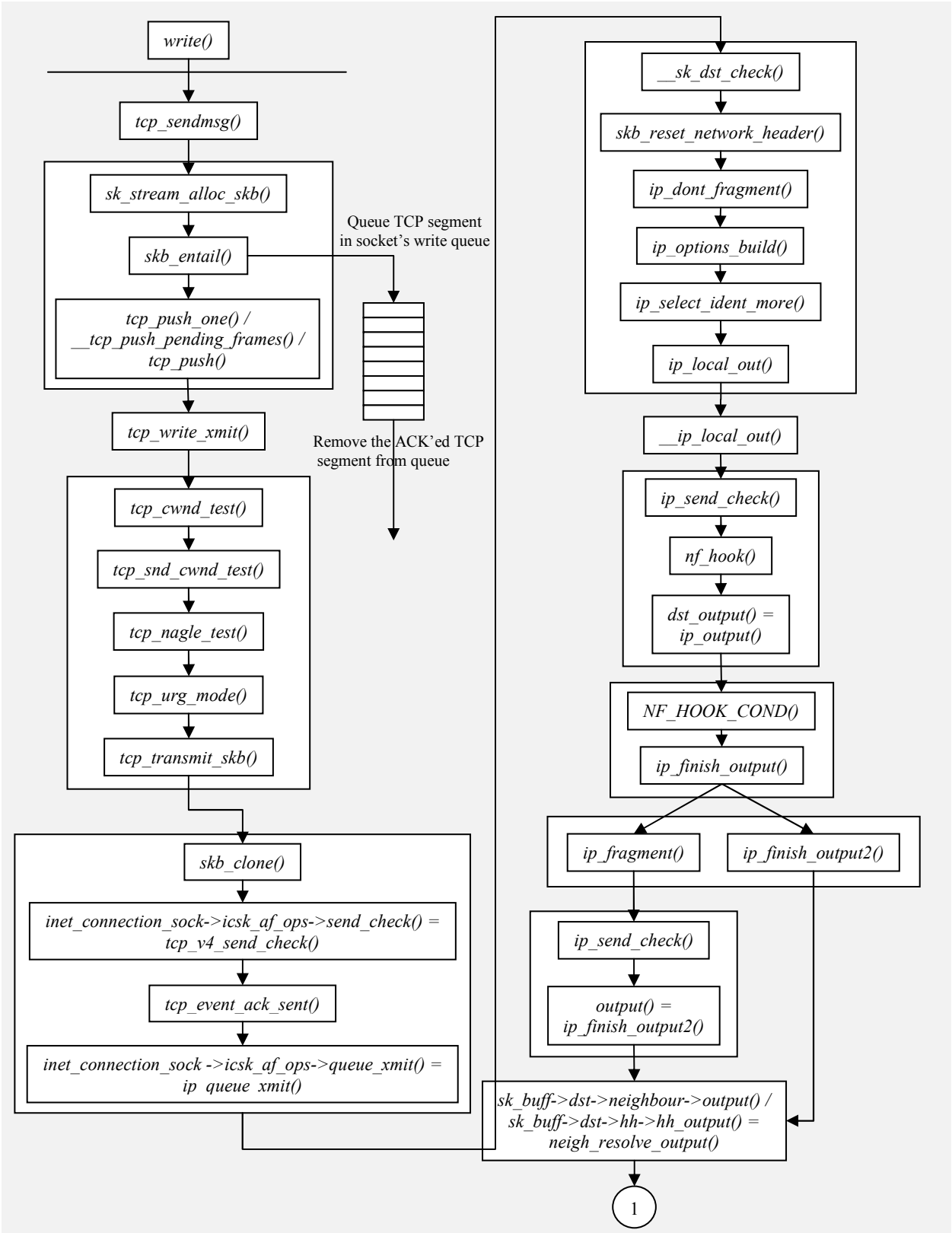


Figure 5 – (a): TCP/IP Send Control Flow In Linux Kernel 2.6.28
(Original figure source: [7], with modifications to fit it for 2.6.28 kernel)

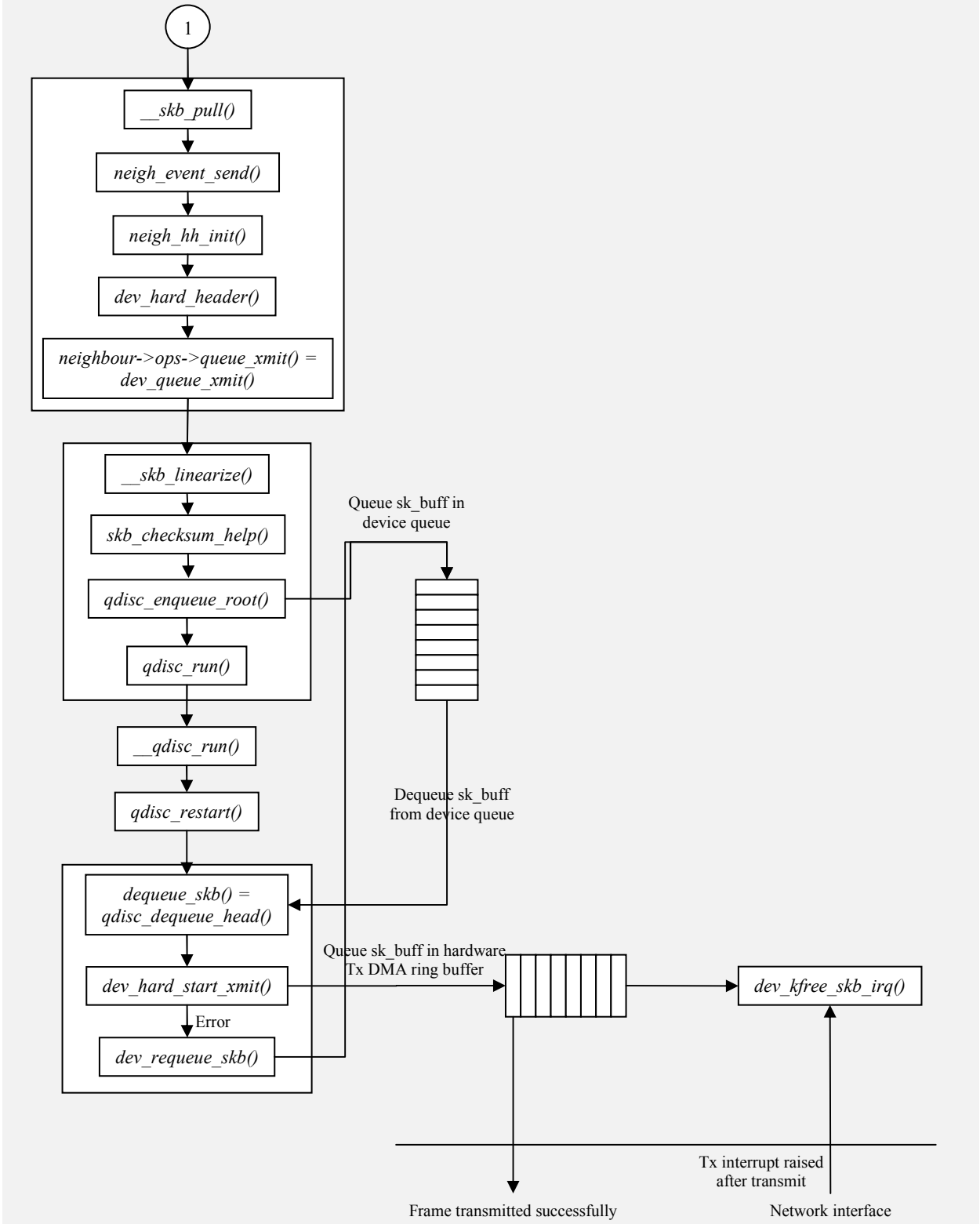


Figure 5 – (b): TCP/IP Send Control Flow In Linux Kernel 2.6.28
(Original figure source: [7], with modifications to fit it for 2.6.28 kernel)

Once we have the route information, it is stored into the packet (*skb->dst)*. Then, IP level header is initialized (*skb_reset_network_header()*). *ip_dont_fragment()* decides whether to set or reset "Don't Fragment" flag. Other IP header fields are also set. *ip_options_build()* takes care of IP options. *ip_select_ident_more()* sets the IP ID in the header based on whether the packet is likely to be fragmented. Then, IP level checksum is calculated by calling *ip_send_check()* in *__ip_local_out()*. Once IP header is built, packet is passed to netfilter *NF_INET_LOCAL_OUT*. This netfilter hook screens the packet against firewall rules set for the packets generated locally. After screening is done, callback routine *dst_output()* is called, which in case of TCP/IP, results in calling *ip_output()*.

ip_output() sets outgoing network interface and protocol (to ETH_P_IP, corresponding to IP protocol) in the packet and sets up a hook in netfilter *NF_INET_POST_ROUTING* to check the packet against post routing rules. Callback routine set for this hook is *ip_finish_output()*. This function checks if size of the packet is greater than MTU. If so, it calls *ip_fragment()* otherwise it calls *ip_finish_output2()*.

ip_fragment() checks if "Don't Fragment" flag for the packet is set. If so, it sends out a "destination unreachable" ICMP message with description "Packet needs to be fragmented. But, DF flag is set." If the flag is not set, it goes ahead with the fragmentation. Once the packet has been fragmented, IP checksum is recalculated and *ip_finish_output2()* is called.

Thus, irrespective of whether or not packet needs to be fragmented, control reaches *ip_finish_output2()*. It needs to find out the hardware address for the destination IP in case where a link layer being used is Ethernet. If the hardware cache corresponding to the route (*skb->dst->hh*) is initialized, *neigh_hh_output()* is called. It copies link level header from the cache and then calls *skb->dst->hh->hh_output()*. If the hardware address has already been cached and the

address is statically configured (i.e. if the state of the corresponding *neighbor* structure is *NUD_CONNECTED* which is defined as *(NUD_PERMANENT | NUD_NOARP | NUD_REACHABLE)*), *hh_output()* points to *dev_queue_xmit()* else it points to *neigh_resolve_output()*. If the hardware cache is not initialized, *skb->dst->neighbor->output()* is called which points to *neigh_resolve_output()*. Thus effectively, if the hardware address has already been figured out, the packet is directly passed down the stack for transmission (*dev_queue_xmit()*); otherwise first ARP tables must be searched for destination IP entry (*neigh_resolve_output()*).

neigh_resolve_output() initiates ARP request by calling *neigh_event_send()*. Once this request completes and the hardware address has been updated in *neighbor* structure, hardware cache for the route is also updated (*neigh_hh_init()*). *dev_hard_header()* is called to fill in the link level header and then, packet is passed down for transmission (*neigh->ops->queue_xmit()* = *dev_queue_xmit()*).

If a device does not have scatter-gather capability, then it cannot send fragmented packets. Hence, in that case *dev_queue_xmit()* tries to linearize the buffer (*__skb_linearize()*). Then, it makes sure that network layer checksum has been calculated. If the checksum is not yet calculated and device is not able to do it in hardware, it is calculated here (*skb_checksum_help()*). Finally, the packet is queued to device queue (*qdisc_enqueue_root()*). Once the packet is on the device queue, transmission is started by calling *qdisc_run()*. If it finds that device is already transmitting from the device queue, it returns immediately. Otherwise it calls *__qdisc_run()*. If some other process needs the CPU or if this device has been running for long time, *__qdisc_run()* calls *__netif_schedule()* which essentially schedules the device on the CPUs output queue (*softnet_data->output_queue*) and raises NET_TX_SOFTIRQ. Otherwise it

continues with the transmission by calling *qdisc_restart()*. *qdisc_restart()* dequeues packets (*qdisc_dequeue_head()*) from the device queue one by one and sends them for transmission. It tries to copy the packet onto DMA ring buffer (*dev_hard_start_xmit()*). If it returns *NETDEV_TX_BUSY*, packet is queued again on device queue (*dev_requeue_skb()*). NIC transmits packets from the DMA ring. When packet is transmitted, Tx interrupt is generated and in the corresponding interrupt handler, buffer allocated to the packet is released.

2.3 TCP Receive

TCP promises reliable, in-order data reception. Hence, TCP receive mechanism employs multiple queues to make sure that application receives data in order and efficiently.

2.3.1 TCP Receive Queues

TCP receive employs four queues,

1. **Receive queue:** In most of the cases, in-order segments are queued in receive queue and application requests are served through it. Segments in this queue are completely processed and ready for the data to be copied into user memory.
2. **Backlog queue:** If receive queue is full or socket is in use, then segments are queued in backlog queue and are processed later when backlog processing is invoked.
3. **Prequeue queue:** Receive queue contains only valid in-order segments. Process of determining if or not a segment is eligible to go onto receive queue is expensive and thus, it is preferable to do that in process context rather than in interrupt context. Prequeue queue is used to defer this processing until in the receiving process' context, if possible.
4. **Out-of-order queue:** When a segment cannot be queued into receive queue because it is out of order, it is queued in out-of-order queue till the hole fills in.

2.3.2 Processing Incoming Segments

2.3.2.1 tcp_v4_rcv()

tcp_v4_rcv() is the door where incoming segments enter for TCP processing. This function first performs checks to see if the segment is to be discarded. If not, it queues the segment in one of the queues.

```
int tcp_v4_rcv(struct sk_buff *skb)
{
 ...
      if (!sock_owned_by_user(sk)) {
          ...
            {
                  if (!tcp_prequeue(sk, skb))
                  ret = tcp_v4_do_rcv(sk, skb);
            }
      } else
            sk_add_backlog(sk, skb);
    ...
}
```

Figure 6: tcp_v4_rcv()

If the socket is in use, then the segment is queued in backlog queue. But, if the socket is not in use, then the segment is tried to be queued in prequeue queue. If this fails, then the segment is sent for further processing immediately by calling *tcp_v4_do_rcv()*. *tcp_v4_do_rcv()* calls *tcp_rcv_established()*, since the socket is in *TCP_ESTABLISHED* state[5].

2.3.2.2 tcp_rcv_established()

tcp_rcv_established() is divided into two parts: slow path processing and fast path processing.

[5] Since we are concerned here only with data send and receive (and not with connection establishment or teardown), we'll always assume connection to be in "established" state.

```
int tcp_rcv_established(struct sock *sk, struct sk_buff *skb, struct tcphdr *th, unsigned len)
{
        ...
        if ((tcp_flag_word(th) & TCP_HP_BITS) == tp->pred_flags && TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
                int tcp_header_len = tp->tcp_header_len;
                /* Check timestamp */
                if (tcp_header_len == sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) {
                        /* No? Slow path! */
                        if (!tcp_parse_aligned_timestamp(tp, th))
                                goto slow_path;
                        /* If PAWS failed, check it more carefully in slow path */
                        if ((s32)(tp->rx_opt.rcv_tsval - tp->rx_opt.ts_recent) < 0)
                                goto slow_path;
                }                                                                    (I)
        ...
        if (len <= tcp_header_len) {
                ...
                } else {
                        int eaten = 0;
                        int copied_early = 0;
                        ...
                        if (tp->copied_seq == tp->rcv_nxt &&  len - tcp_header_len <= tp->ucopy.len) {
                                ...
                                if (tp->ucopy.task == current &&  sock_owned_by_user(sk) && !copied_early) {
                                        __set_current_state(TASK_RUNNING);
                                        if (!tcp_copy_to_iovec(sk, skb, tcp_header_len))
                                                eaten = 1;
                                }
                                ...
                        }                                                            (II)
                        if (!eaten) {
                                ...
                                __skb_queue_tail(&sk->sk_receive_queue, skb);
                                ...
                        }
                        tcp_event_data_recv(sk, skb);
                        ...
                        return 0;                                                    (III)
                }
        }

slow_path:
        ...
        res = tcp_validate_incoming(sk, skb, th, 1);                                 (IV)
        ...
step5:
        ...
        /* Process urgent data. */
        tcp_urg(sk, skb, th);                                                        (V)

        /* step 7: process the segment text */
        tcp_data_queue(sk, skb);                                                     (VI)
        ...
        return 0;
        ...
}
```

Figure 7: tcp_rcv_established()

Slow path processing is usual TCP segment processing which takes care of all the cases like out of order data, urgent data etc., whereas fast path processing is minimal processing which includes mainly copying/queuing the segment, acknowledging the data and storing timestamp. If the segment satisfies required criteria, it becomes eligible for fast path processing or else it is sent for slow path processing(I). In fast path processing, segment is checked for the possibility of direct copy to user buffer. This can be done only if following criteria are met(II),

- Sequence is equal to next expected byte. If it is not, copying data to user memory at this stage will result in out of order data.
- Size of the segment is less than or equal to size of the data that is still to be copied to user buffer. Segments are not copied partially to the user buffer.
- Task which requested the data is scheduled currently.
- Socket is owned by someone.

The last two conditions ensure that the routine is called from the context of the process requesting the data and not from interrupt context (a segment is queued in backlog queue, in interrupt context, if socket is in use.) or context of any other process. If any of these criteria is not met, the segment is queued into receive queue and function returns(III).

In slow path processing, it validates the incoming segment(IV), does the urgent data handling(V) and then, queues segment in the appropriate queue i.e. in receive queue or out-of-order queue (in *tcp_data_queue()*(VI)).

2.3.3 Reading Data From Receive Queue

For reading data from socket, receiving process issues a system call (*read()*, *recvmsg()* etc.). In case of TCP sockets, all these calls end up in *tcp_recvmsg()*.

2.3.3.1 tcp_recvmsg()

tcp_recvmsg() copies data from receive queue to user buffer. Figure 8 shows important fragments of *tcp_recvmsg()*. The focus of these fragments is on queues processing and data copy to user buffer.

In this function, queues are processed after taking a lock on the socket(I). Queues are processed in sequence; receive queue first, then prequeue queue and finally, backlog queue. The moment lock is granted, data is present in receive queue only. Since receiver is not yet installed, there cannot be data in prequeue queue (*tcp_prequeue()* queues packets into prequeue queue only if receiver is installed). And since the lock is granted, there cannot be any data in backlog queue either (data is queued in backlog queue only when lock is held by someone on the socket and backlog queue is processed while releasing the lock which queues the packets in receive queue, if receiver is not installed). Hence, receive queue can be processed directly.

First a segment containing the next required byte is found out by looping through receive queue(II). If such a segment is found, then code jumps to its processing otherwise it proceeds to backlog queue processing. Next, current process is installed as a receiver, if none is already installed(III). Then, data in the prequeue queue is processed, if it is not empty(IV). In the first iteration, there will not be any data in the prequeue queue, since receiver has been installed after acquiring the lock and lock has not been released since then (data is inserted in prequeue queue when receiver is installed and socket is not in use by anyone). If more (or equal) bytes have been copied than the *target* (*target* is the minimum number of bytes that should be copied before returning. It can be smaller than user requested size, in case of non-blocking calls, for example), then backlog queue is processed by releasing and acquiring the lock again. Releasing the lock processes the backlog queue(V).

```
int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
        size_t len, int nonblock, int flags, int *addr_len)
{
        ...
        lock_sock(sk);                                                          (I)
        /* Urgent data needs to be handled specially. */
        if (flags & MSG_OOB)
                goto recv_urg;
        ...
        do {
                ...
                skb = skb_peek(&sk->sk_receive_queue);
                do {
                        ...
                        offset = *seq - TCP_SKB_CB(skb)->seq;
                        ...
                        if (offset < skb->len)
                                goto found_ok_skb;
                        if (tcp_hdr(skb)->fin)
                                goto found_fin_ok;
                        skb = skb->next;
                } while (skb != (struct sk_buff *)&sk->sk_receive_queue);       (II)
                /* Well, if we have backlog, try to process it now yet. */
                ...
                tcp_cleanup_rbuf(sk, copied);
                if (!sysctl_tcp_low_latency && tp->ucopy.task == user_recv) {
                        /* Install new reader */
                        if (!user_recv && !(flags & (MSG_TRUNC | MSG_PEEK))) {
                                user_recv = current;
                                tp->ucopy.task = user_recv;
                                tp->ucopy.iov = msg->msg_iov;
                        }                                                       (III)
                        tp->ucopy.len = len;
                        ...
                        if (!skb_queue_empty(&tp->ucopy.prequeue))
                          goto do_prequeue;                                     (IV)
                }
                if (copied >= target) {
                        /* Do not sleep, just process backlog. */
                        release_sock(sk);
                        lock_sock(sk);                                          (V)
                } else
                        sk_wait_data(sk, &timeo);                               (VI)
                ...
                if(user_recv) {
                        ...
                        if (tp->rcv_nxt == tp->copied_seq &&
                                !skb_queue_empty(&tp->ucopy.prequeue)) {
do_prequeue:
                                tcp_prequeue_process(sk);
                        }
                }
                continue;                                                       (VII)

        found_ok_skb:
                /* Ok so how much can we use? */
                used = skb->len - offset;
                if (len < used)
                        used = len;                                             (VIII)
```

Figure 8 – (a): tcp_recvmsg()

```
                ...
                if (!(flags & MSG_TRUNC)) {
                        ...
                        err = skb_copy_datagram_iovec(skb, offset, msg->msg_iov, used);   (IX)
                        ...
                }
                *seq += used;
                copied += used;
                len -= used;
                tcp_rcv_space_adjust(sk);                                                 (X)

        skip_copy:
                ...
                if (used + offset < skb->len)
                        continue;

                if (tcp_hdr(skb)->fin)
                        goto found_fin_ok;
                if (!(flags & MSG_PEEK)) {
                        sk_eat_skb(sk, skb, copied_early);
                        copied_early = 0;
                }
                continue;                                                                 (XI)

        found_fin_ok:
                /* Process the FIN. */
                ++*seq;
                if (!(flags & MSG_PEEK)) {
                        sk_eat_skb(sk, skb, copied_early);
                        copied_early = 0;
                }
                break;                                                                    (XII)
        } while (len > 0);

        if (user_recv) {
                if (!skb_queue_empty(&tp->ucopy.prequeue)) {
                        ...
                        tp->ucopy.len = copied > 0 ? len : 0;
                        tcp_prequeue_process(sk);
                        ...
                }                                                                         (XIII)
                tp->ucopy.task = NULL;
                tp->ucopy.len = 0;
        }
        ...
        release_sock(sk);
        return copied;                                                                    (XIV)
        ...
}
```

Figure 8 – (b): tcp_recvmsg()

If number of bytes copied is less than the target, then code waits for more data[(VI)]. After this if-else condition, there can be data in prequeue queue. Hence, that is processed now, which

may end up adding segments into receive queue. Now, code jumps again to the start of the loop (where receive queue will be searched for a segment containing the next required byte)(VII).

If a segment containing the required byte is found, code jumps to *found_ok_skb*. Here first, size of the data to be copied is adjusted(VIII). Then, data is copied to user buffer(IX). Next sequence number (*seq*), no of bytes copied (*copied*) are adjusted. *tcp_rcv_space_adjust()* adjusts the size of TCP receive buffer space(X). Then, if all the data from the segment is copied and flags do not indicate *MSG_PEEK*, the segment is removed from the queue and is released (*sk_eat_skb()*). The loop continues(XI).

If finish packet is found, code reaches *found_fin_ok*, where *sk_eat_skb()* is called if *MSG_PEEK* is not specified and code breaks out of the loop(XII).

Here, prequeue queue is processed again, if not empty, to copy more data to user buffer if possible (and allowed) and to empty prequeue queue before the next call(XIII).

In the end, receiver is uninstalled, lock is released and function returns(XIV).

2.4 TCP Send

2.4.1 Writing Data To Write Queue For Sending

2.4.1.1 tcp_sendmsg()

Any system call that writes data on TCP socket ends up in *tcp_sendmsg()*. This function splits user buffer into segments, enqueues these segments onto transmit queue and then, pushes them off the transmit queue for transmission.

The send processing starts by acquiring the lock on the socket(I). Then, MSS (Maximum Segment Size) is obtained (*tcp_send_mss()*)(II).

```
int tcp_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg, size_t size)
{
        ...
        lock_sock(sk);                                                          (I)
        ...
        mss_now = tcp_send_mss(sk, &size_goal, flags);                          (II)
        ...
        while (--iovlen >= 0) {
                int seglen = iov->iov_len;
                unsigned char __user *from = iov->iov_base;
                iov++;
                while (seglen > 0) {
                        ...
                        skb = tcp_write_queue_tail(sk);

                        if (!tcp_send_head(sk) || (copy = size_goal - skb->len) <= 0) {

new_segment:
                                ...
                                skb = sk_stream_alloc_skb(sk, select_size(sk), sk->sk_allocation);
                                ...
                                if (sk->sk_route_caps & NETIF_F_ALL_CSUM)
                                        skb->ip_summed = CHECKSUM_PARTIAL;
                                skb_entail(sk, skb);
                                copy = size_goal;
                        }                                                       (III)
                        ...
                        if (skb_tailroom(skb) > 0) {
                                if (copy > skb_tailroom(skb))
                                        copy = skb_tailroom(skb);
                                if ((err = skb_add_data(skb, from, copy)) != 0)
                                        goto do_fault;                          (IV)
                        } else {
                                ...
                                if (skb_can_coalesce(skb, i, page, off) && off != PAGE_SIZE) {
                                        ...
                                        merge = 1;
                                } else if (i == MAX_SKB_FRAGS ||
                                        (!i &&!(sk->sk_route_caps & NETIF_F_SG))) {
                                        ...
                                        goto new_segment;
                                } else if (page) {
                                        if (off == PAGE_SIZE) {
                                                put_page(page);
                                                TCP_PAGE(sk) = page = NULL;
                                                off = 0;
                                        }
                                } else
                                        off = 0;
                                ...
                                if (!page) {
                                        if (!(page = sk_stream_alloc_page(sk)))
                                                goto wait_for_memory;
                                }
                                err = skb_copy_to_page(sk, from, skb, page, off, copy);   (V)
                                ...
                        }
```

Figure 9 – (a): tcp_sendmsg()

```
                        ...
                        tp->write_seq += copy;
                        TCP_SKB_CB(skb)->end_seq += copy;
                        ...
                        from += copy;
                        copied += copy;
                        if ((seglen -= copy) == 0 && iovlen == 0)
                                goto out;                                          (VI)
                        if (skb->len < size_goal || (flags & MSG_OOB))
                                continue;                                          (VII)
                        if (forced_push(tp)) {
                                tcp_mark_push(tp, skb);
                                __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
                        } else if (skb == tcp_send_head(sk))
                                tcp_push_one(sk, mss_now);
                        continue;                                                  (VIII)
                        ...
                }
        }

out:
        if (copied)
                tcp_push(sk, flags, mss_now, tp->nonagle);
        ...
        release_sock(sk);
        return copied;                                                             (IX)
        ...
}
```

Figure 9 – (b): tcp_sendmsg()

Now, the two nested loops create segments using all the data from user buffer. First, last segment in transmit queue is checked for empty space. In case transmit queue is empty or size of the last segment is more than *size_goal* (calculated in *tcp_send_mss()*), new segment needs to be created. *sk_stream_alloc_skb()* allocates a segment. Then, hardware capability for calculating checksum is checked and segment is marked accordingly (If hardware does not support checksumming, then data checksum is calculated while copying data to segment). Then, newly created segment is queued at the end of the transmit queue. After this part, code is same for both cases: last segment in the transmit queue is chosen or new segment is created. At this point, *skb* is pointing to the last segment in the transmit queue and it has space for data(III).

If space is available in the *skb*'s linear data block (*skb_tailroom() > 0*), then data that can fit into the block, is copied there(IV). If there is no space in the linear data block, then last paged fragment of the non-linear data block is checked. If there is not enough space in any of these, then a new fragment needs to be added to the non linear data block. But, if number of fragments have reached the maximum allowed number (*MAX_SKB_FRAGS*) or if hardware does not support non-linear data block (*!(sk->sk_route_caps & NETIF_F_SG)*), then new segment needs to be allocated. At the end of these checks, if a paged fragment is available for copying data into, *page* will not be NULL. So, if *page* is NULL, then a new fragment is allocated. In either case, data is copied to the fragment (either previously available or newly created). As mentioned before, if required, checksum is also calculated while copying the data(V).

If all the data in the buffer has been copied, code jumps to *out*, where all the segments on transmit queue are pushed for transmission, lock is released and function returns(VI)(IX). Otherwise if size of the current segment is less than *size_goal* (This can happen mostly in case of non-linear *sk_buff*s), more data can fit into segment. Hence, iteration continues without pushing that segment(VII).

When code reaches here, there is no space left in the segment. *forced_push()* is called to check if current segment is to be marked for PUSH. If yes, then it is marked for PUSH (*tcp_mark_push()*) and all the segments in transmit queue are pushed for transmission. If current segment is not to be marked for PUSH, but if it is the only segment in the transmit queue, then it is pushed for transmission by calling *tcp_push_one()*. This function tries to transmit the segment immediately (because, this is the only segment on the transmit queue, no ACK is pending). And the loop continues(VIII).

2.4.1.2 sk_stream_alloc_skb()

sk_stream_alloc_skb() is called in *tcp_sendmsg()* to allocate new *sk_buff*.

```
struct sk_buff *sk_stream_alloc_skb(struct sock *sk, int size, gfp_t gfp)
 {
            struct sk_buff *skb;
            /* The TCP header must be at least 32-bit aligned.  */
            size = ALIGN(size, 4);
            skb = alloc_skb_fclone(size + sk->sk_prot->max_header, gfp);
            if (skb) {
                        if (sk_wmem_schedule(sk, skb->truesize)) {
                                    /*
                                    * Make sure that we have exactly size bytes
                                    * available to the caller, no more, no less.
                                    */
                                    skb_reserve(skb, skb_tailroom(skb) - size);
                                    return skb;
             }
              __kfree_skb(skb);
       } else {
             ...
       }
       return NULL;
 }
```

Figure 10: sk_stream_alloc_skb()

size specifies size of the data in linear data block. Then, new *sk_buff* is allocated by calling *alloc_skb_fclone()* which in turn calls *__alloc_skb()*. If sufficient forward allocated memory is not available, then allocated *sk_buff* is freed and the function returns NULL. Otherwise, *skb_reserve()* is called to create sufficient headroom by shifting data and tail pointers.

2.4.1.3 __alloc_skb()

This is the function which actually allocates *sk_buff*. It starts with allocating *sk_buff* structure. It also allocates space for linear data block. Then, linear data block related fields of *sk_buff* are initialized. Corresponding *skb_shared_info* structure is also initialized to reflect no fragments in non-linear data block.

```
struct sk_buff *__alloc_skb(unsigned int size, gfp_t gfp_mask, int fclone, int node)
 {
        ...
        /* Get the HEAD */
        skb = kmem_cache_alloc_node(cache, gfp_mask & ~__GFP_DMA, node);
        ...
        size = SKB_DATA_ALIGN(size);
        data = kmalloc_node_track_caller(size + sizeof(struct skb_shared_info), gfp_mask, node);
        ...
        memset(skb, 0, offsetof(struct sk_buff, tail));
        skb->truesize = size + sizeof(struct sk_buff);
        atomic_set(&skb->users, 1);
        skb->head = data;
        skb->data = data;
        skb_reset_tail_pointer(skb);
        skb->end = skb->tail + size;
        /* make sure we initialize shinfo sequentially */
        shinfo = skb_shinfo(skb);
        atomic_set(&shinfo->dataref, 1);
        shinfo->nr_frags  = 0;
        shinfo->gso_size = 0;
        shinfo->gso_segs = 0;
        shinfo->gso_type = 0;
        shinfo->ip6_frag_id = 0;
        shinfo->tx_flags.flags = 0;
        shinfo->frag_list = NULL;
        memset(&shinfo->hwtstamps, 0, sizeof(shinfo->hwtstamps));
        ...
 out:
        return skb;
        ...
 }
```

Figure 11: __alloc_skb()

CHAPTER 3

IMPLEMENTATION

This chapter discusses the implementation done as part of this work. First part of this chapter explains the additions and modifications done in the Linux 2.6.28.8 network stack code to provide the framework required by the kernel mode SMTP server to avoid memory copy[2][7][8][9][10][11][12]. The second part explains the implementation of SMTP relay servers[2][11][12][13][14][15].

3.1 Kernel Code Changes

New TCP receive function (*tcp_recvskbuff()*) is added in the Linux kernel which instead of returning data copied into supplied buffer, returns list of received *sk_buff*s. Similarly, a new TCP send function (*tcp_sendskbuff()*) is also added which accepts a list of *sk_buff*s to be transmitted. Apart from these, some functions are added and some are modified to support these two main functions. Urgent data handling and OOB (Out Of Band) data handling is not done in these functions since these features are not required by SMTP relay code.

3.1.1 TCP Receive Changes

Only one function (*tcp_recvskbuff()*) has been added in the TCP receive code and minor modifications to *tcp_rcv_established()* and *tcp_data_queue()* are done to prevent direct copy to the user buffer and to skip urgent data handling.

3.1.1.1 Additions

3.1.1.1.1 tcp_recvskbuff()

tcp_recvskbuff() maintains the same structure as *tcp_recvmsg()*, in fact code is exactly same as *tcp_recvmsg()* only with required modifications. Code fragments in this section describe the modifications.

```
        /* Urgent data needs to be handled specially. */
//      if (flags & MSG_OOB)
//              goto recv_urg;

...

        if (tp->urg_data && tp->urg_seq == *seq) {
//              if (copied)
//                      break;
//              if (signal_pending(current)) {
//                      copied = timeo ? sock_intr_errno(timeo) : -EAGAIN;
//                      break;
//              }
        }

...

//      if (tp->urg_data) {
//              u32 urg_offset = tp->urg_seq - *seq;
//              if (urg_offset < used) {
//                      if (!urg_offset) {
//                              if (!sock_flag(sk, SOCK_URGINLINE)) {
//                                      ++*seq;
//                                      offset++;
//                                      used--;
//                                      if (!used)
//                                              goto skip_copy;
//                              }
//                      } else
//                              used = urg_offset;
//              }
//      }

...

//skip_copy:
//      if (tp->urg_data && after(tp->copied_seq, tp->urg_seq)) {
//              tp->urg_data = 0;
//              tcp_fast_path_check(sk);
//      }
//      if (used + offset < skb->len)
//              continue;

...

//recv_urg:
//      err = tcp_recv_urg(sk, timeo, msg, len, flags, addr_len);
//      goto out;
```

These code fragments from *tcp_recvmsg()* have been commented out in *tcp_recvskbuff()* so as to disable urgent data handling.

```
//          if ((chunk = len - tp->ucopy.len) != 0) {
//          NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMBACKLOG, chunk);
//          len -= chunk;
//          copied += chunk;
```

This code fragment, which appears thrice in *tcp_recvmsg()*, is required after backlog queue and prequeue queue processing. During this processing, provided that all the required conditions are met (described in section 2.3.2.2), data from segments in the prequeue queue can be directly copied into user buffer. *chunk* has the size of data copied to user buffer as a part of the recent processing. This code fragment accounts for this data in *tcp_recvmsg()*. But, since direct copy to user buffer has been disabled, this fragment is not required in *tcp_recvskbuff()*.

```
struct sk_buff *last = head->prev;
__skb_unlink(skb, &sk->sk_receive_queue);
skb->destructor = NULL;
sock_rfree(skb);
last->next = skb;
skb->next = (struct sk_buff *)head;
head->prev = skb;
skb->prev = last;
head->qlen++;
```

This code fragment is used in place of call to *sk_eat_skb()*. *sk_eat_skb()* first dequeues the segment from receive queue. Then, it calls *__kfree_skb()* to release the memory allocated to the segment. This call results in the destructor (*sock_rfree()* in this case) of the segment being called. *sock_rfree()* uncharges the socket for the memory released by the current segment and makes room, in the receive queue, of equal size. *tcp_recvskbuff()* needs to carry out all these operations but it should not free the memory associated with the segment. *__skb_unlink()* dequeues segment from receive queue. *sock_rfree()* does the memory uncharging and then, segment is queued in the list to be returned. Destructor is now set to NULL, since *sock_rfree()* has been explicitly called for this segment.

One more difference between *tcp_recvmsg()* and *tcp_recvskbuff()* is about the upper limit put on the data returned. *tcp_recvmsg()* strictly follows the maximum size specified by the caller while returning data. But, *tcp_recvskbuff()* might sometimes return more data than specified because it cannot return partial segments.

3.1.1.2 Changes

Other changes, that are required to support *tcp_recvskbuff()*, are concerned with commenting out code fragments that copy data from segments directly to user buffer without queuing them into receive queue and with commenting out urgent data processing.

3.1.1.2.1 tcp_data_queue()

```
//      if (tp->ucopy.task == current &&
//         tp->copied_seq == tp->rcv_nxt && tp->ucopy.len &&
//         sock_owned_by_user(sk) && !tp->urg_data) {
//                 int chunk = min_t(unsigned int, skb->len,  tp->ucopy.len);
//                 __set_current_state(TASK_RUNNING);
//
//                 local_bh_enable();
//                 if (!skb_copy_datagram_iovec(skb, 0, tp->ucopy.iov, chunk)) {
//                         tp->ucopy.len -= chunk;
//                         tp->copied_seq += chunk;
//                         eaten = (chunk == skb->len && !th->fin);
//                         tcp_rcv_space_adjust(sk);
//                 }
//                 local_bh_disable();
//      }
```

This is a code fragment from *tcp_data_queue()*, which copies data directly to user buffer without queuing segment into receive queue. This needs to be commented out to eliminate the memory copy completely.

3.1.1.2.2 tcp_rcv_established()

```
//      if (tp->ucopy.task == current &&
//      sock_owned_by_user(sk) && !copied_early) {
//                 __set_current_state(TASK_RUNNING);
//                 if (!tcp_copy_to_iovec(sk, skb, tcp_header_len))
//                         eaten = 1;
//      }
```

This is similar code fragment from *tcp_rcv_established()*, that copies data directly to user buffer and needs to be commented out for previously explained reasons.

```
//tcp_urg(sk, skb, th);
```

This line is also commented, which does urgent data handling.

3.1.2 TCP Send Changes

There are no changes in the existing code for TCP send. But, some functions have been added.

3.1.2.1 Additions

3.1.2.1.1 tcp_sendskbuff()

tcp_sendskbuff() is a variation of *tcp_sendmsg()* that accepts list of *sk_buff*s to be transmitted. The major difference between the two is that *tcp_sendskbuff()* does not need to allocate space for data part of network buffers.

```
new = sk_stream_alloc_skb_wo_data(sk, orig, sk->sk_allocation);
if (!new)
        goto wait_for_memory;
kfree_skb(orig);
```

tcp_sendskbuff() uses newly added *sk_stream_alloc_skb_wo_data()* instead of *sk_stream_alloc_skb()* to allocate *sk_buff*s. The new function differs from original one so as not to allocate memory for the linear data block. Once the new *sk_buff* is allocated and points to data blocks of the original *sk_buff*, original *sk_buff* is released and new *sk_buff* is queued to transmit queue for transmission.

tcp_sendmsg() calculates the checksum, if necessary, while copying the data to *sk_buff*. But, the copy itself is avoided in *tcp_sendskbuff()*. So, it calculates checksum separately, if necessary.

3.1.2.1.2 sk_stream_alloc_skb_wo_data()

sk_stream_alloc_skb_wo_data() is a variation of *sk_stream_alloc_skb()*, that has to take care of setting up data pointers properly since it calls *alloc_skb_fclone_wo_data()* (which in turn calls *__alloc_skb_wo_data()*) that does not allocate space for linear data block. Hence, *sk_stream_alloc_skb_wo_data()* calls *skb_set_data_part()* which initializes data pointers and *skb_shared_info* structure to point to linear and non-linear data blocks of the original segment passed to this function.

```
struct sk_buff *sk_stream_alloc_skb_wo_data(struct sock *sk, struct sk_buff *orig_skb, gfp_t gfp)
{
        struct sk_buff *skb;
        skb = alloc_skb_fclone_wo_data(gfp);
        if (skb)
        {
                skb_set_data_part(skb, orig_skb);
                if (sk_wmem_schedule(sk, skb->truesize))
                {
                        return skb;
                }
                __kfree_skb(skb);
        }
        else
        {
                ...
        }
        return NULL;
}
```

Figure 12: sk_stream_alloc_skb_wo_data()

3.1.2.1.3 __alloc_skb_wo_data()

__alloc_skb_wo_data() is a variation of *__alloc_skb()* that does not allocate space for linear data block and hence, does not initialize linear or non-linear data pointers in *sk_buff* and corresponding *skb_shared_info*.

```
struct sk_buff *__alloc_skb_wo_data(gfp_t gfp_mask, int fclone, int node)
{
        ...
        cache = fclone ? skbuff_fclone_cache : skbuff_head_cache;

        /* Get the HEAD */
        skb = kmem_cache_alloc_node(cache, gfp_mask & ~__GFP_DMA, node);
        if (!skb)
                goto out;

        memset(skb, 0, offsetof(struct sk_buff, users));
        atomic_set(&skb->users, 1);
        ...
out:
        return skb;
}
```

Figure 13: __alloc_skb_wo_data()

3.1.2.1.4 skb_set_data_part()

skb_set_data_part() accepts two *sk_buff*s and initializes data pointers of one *sk_buff* and its *skb_shared_info* structure to point to linear and non-linear data block of other *sk_buff*.

```
void skb_set_data_part(struct sk_buff *new_skb, struct sk_buff *orig_skb)
{
#define C(x) new_skb->x = orig_skb->x

        C(truesize);
        C(head);
        C(data);
        C(tail);
        C(end);
        C(len);
        C(data_len);

        new_skb->cloned = 1;
        new_skb->destructor = NULL;
        new_skb->hdr_len = orig_skb->nohdr ? skb_headroom(orig_skb) : orig_skb->hdr_len;
        new_skb->nohdr = 0;

        atomic_inc(&(skb_shinfo(orig_skb)->dataref));
        orig_skb->cloned = 1;
#undef C
}
```

Figure 14: skb_set_data_part ()

3.2 SMTP Relay Server Code Outline

SMTP (Simple Mail Transfer Protocol) is a protocol used to transfer e-mails from one host to another efficiently and reliably. As the name suggests, the protocol is designed to be very simple. Step-locked conversation with only one client at a time (no thread of an SMTP server can be engaged in conversations with more than one SMTP clients simultaneously) makes it very easy to implement. Client initiates a conversation by establishing a two-way transmission channel with the server. Then, they communicate with a sequence of commands and replies. With each successful pair of a command and a reply, server and client enter a new state. Not all commands are valid in each state. If an invalid command is sent by the client, server sends back a reply indicating that the command is invalid and both remain in the same state. Section 3.2.2 explains the SMTP state diagrams implemented as a part of this work. When the conversation is complete, either successfully or unsuccessfully, the connection is torn down.

3.2.1 SMTP Commands And Replies

SMTP conversation is nothing but a sequence of commands and replies. SMTP commands are used by a client to direct the conversation with the server. All commands end with <CRLF> (“\r\n”). Some commands are followed by parameters. In this case, command and parameters are separated by <SP> (“ ”). Commands are not case sensitive. They can be issued in upper, lower or mixed case. In response to every command, server sends back a reply. These replies help in maintaining the lock-stepped nature of the communication by keeping client aware of the server state. Replies are formatted as three digit code followed by <SP> followed by text corresponding to the reply and ending with <CRLF>. Generally, three-digit code is sufficient for a client to identify the reply. Text is for human users.

HELO: HELO command is used to start the conversation. It requires client to send its identification as a parameter. Using this identification, server decides if or not the conversation should be continued. If server decides to continue the conversation, it sends back "250 Action Okay" reply. In this reply, server identifies itself to the client. But, if server decides not to continue the conversation, it sends back an appropriate error reply.

MAIL_FROM: MAIL_FROM command informs the server of reverse path (sender's ID). In response to this command, server clears reverse path buffer, forward path buffer and mail data buffer. Then, it adds the reverse path from the command parameter into reverse path buffer. If all this process is successful, server sends back "250 Action Okay" reply otherwise it sends back appropriate error reply code.

RCPT_TO: RCPT_TO command informs the server of forward path (recipients' IDs). Each command specifies only one ID. Hence, if there are multiple recipients, this command must be repeated once for each recipient. Server appends these IDs one by one to forward path buffer and sends back "250 Action Okay" if successful and appropriate error reply code otherwise.

DATA: DATA command transfers the e-mail contents. Server sends "354 Start Mail Input" reply in response to DATA command. From this point onwards, until client sends <CRLF>.<CRLF> (i.e. a line containing only "."), all the text is assumed to be e-mail data and is copied into mail data buffer. When server detects <CRLF>.<CRLF>, it is treated as end of e-mail and server sends back "250 Action Okay" reply, if successful.

QUIT: QUIT proposes connection teardown. In response to QUIT, server must send "221 Service Closing" reply and then, close the connection. Server must not close the connection with the client until it receives QUIT command and it responds with "221", even in case of

errors. Similarly, client must not close the connection until it sends QUIT command and gets "221" reply. If connection is terminated prematurely, current transaction is discarded.

RSET: RSET command is used to reset the transaction. In response to this command, server aborts current transaction by clearing all the buffers and sends back "250 Action Okay" reply. Note that this command does not close the connection; it just aborts the current transaction.

3.2.2 State Diagrams

As explained previously, during any SMTP conversation, server and client go through sequence of states. This section gives an overview of server side and client side state diagrams implemented as a part of this work.

Server Side State Diagram: Figure 15 shows the server side state diagram corresponding to the code implemented. Server goes through *Begin*, *Wait*, *Envelope Created*, *Setting Recipients*, *Writing Mail*, *Mail Completed* and *Closed* states. A server starts in *Closed* state. When it receives a connection request from client and the connection is established, it goes to *Begin* state. In this state, it waits for mail communication to start. Client initiates mail communication by sending HELO command. In response to this command, the server goes in *Wait* state. In this state, all the buffers (reverse path, forward path and mail) are cleared. While the mail communication is in progress, if a client ever sends HELO/RSET command, server transitions immediately to this state clearing all the buffers (indicated by dashed arrows in figure 15). In *Wait* state, when server receives MAIL_FROM command, it initializes reverse path buffer and goes to *Envelope Created* state. 'Envelope' contains address of the original sender, addresses of recipients etc. So now, part of the envelope is filled and it waits for RCPT_TO commands to fill up the remaining envelope. On reception of first RCPT_TO, server enters

Setting Recipients state. It accepts all the remaining RCPT_TO commands in this state and completes the mail envelope. Once it gets the DATA command, provided that at least one of the RCPT_TOs was successful, it moves on to *Writing Mail* state where it loops to get the mail data and fill it in the mail buffer until it receives end of mail sequence, a line containing only a period. On reception of end of mail data sequence, it goes to *Mail Completed* state and waits for QUIT command to terminate the mail conversation. On reception of QUIT command, it goes back to *Closed* state. In fact, if a server receives QUIT in any state, it terminates the connection and goes back to *Closed* state (dashed-dotted arrows). In any state, if command processing is not successful (because either command sent by client was erroneous or server was unable to process the command), server sends back an error reply describing the error and stays in the same state.

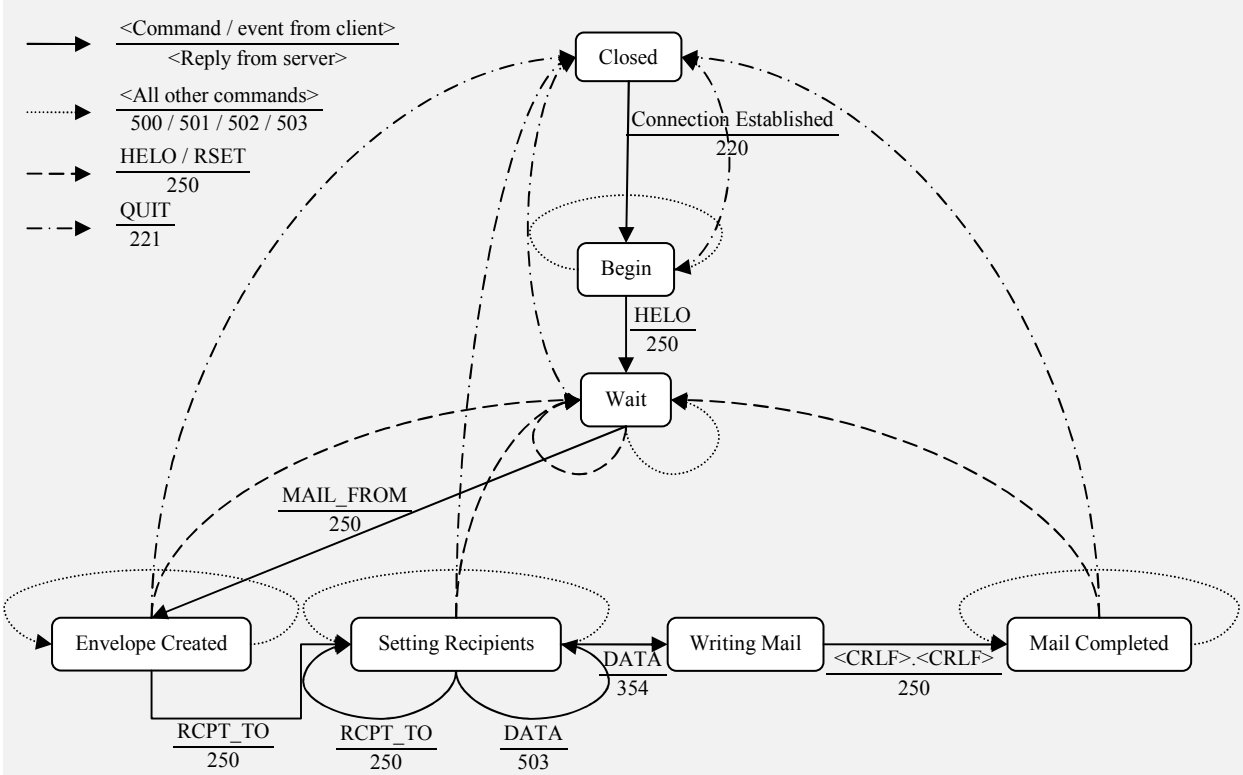


Figure 15: SMTP Server Side State Diagram

Client Side State Diagram: Figure 16 shows client side state diagram for SMTP communication. Client also starts in *Closed* state. After sending a connection request to the server, it moves to *Begin* state. When the server responds with "220 Service Ready" reply, client sends out HELO and goes to *Wait* state. On server's positive reply, it sends MAIL_FROM and transitions to *Setting Recipients* state.

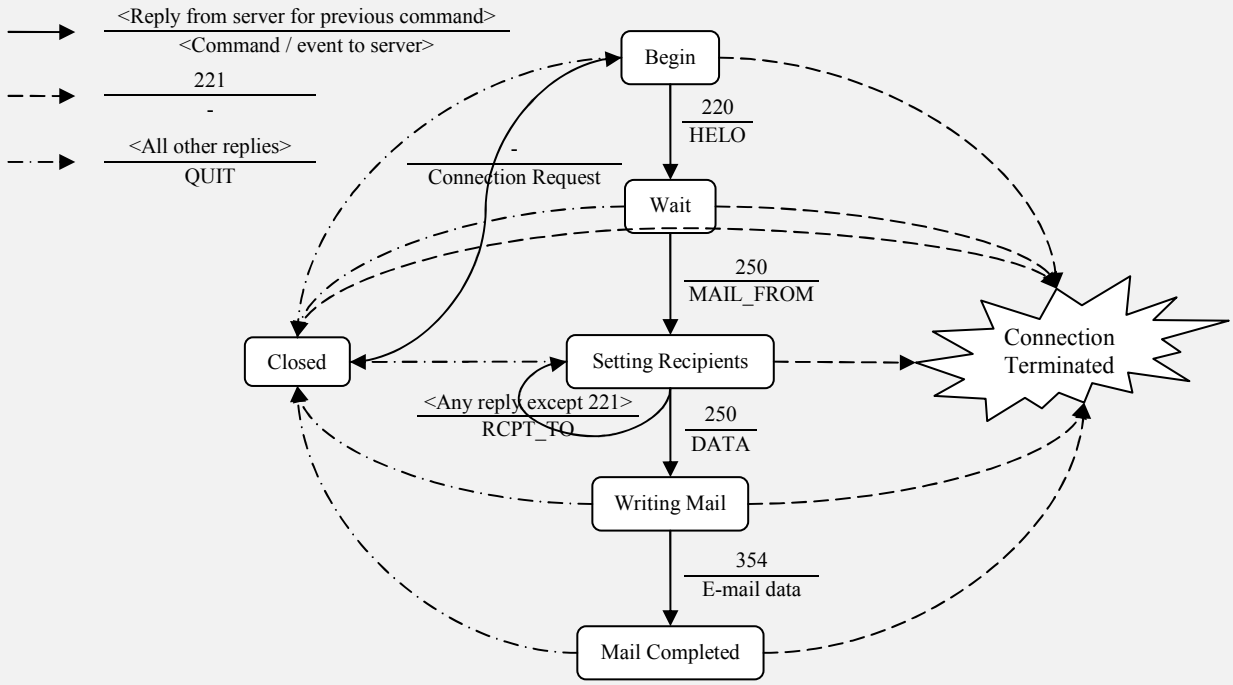


Figure 16: SMTP Client Side State Diagram

If the reply from server for MAIL_FROM was positive, it sends RCPT_TOs one by one until all recipient IDs are sent to server. It continues sending recipient IDs even if server replies with error messages for some RCPT_TOs. Once all the recipients are communicated to the server, it sends DATA command and goes into *Writing Mail* state. On receiving "354 Start Mail Data" reply, it sends out mail data ending in a line containing only a period and transitions to *Mail Completed* state. Then, it sends QUIT command and goes to *Closed* state. Finally, the connection is terminated when server replies to QUIT by "221 Service Closing" reply. In fact, no

matter what state client is in, when it receives “221” reply from the server, it terminates the connection (indicated by dashed arrows). When client encounters an unexpected reply, it sends QUIT command and moves to *Closed* state (indicated by dashed-dotted arrows), effectively terminating the communication.

CHAPTER 4

EXPERIMENTS AND RESULTS

This chapter discusses setup and results of the test conducted to evaluate kernel mode server. The test compares e-mail level latencies and throughputs of both the SMTP relay servers.

4.1 Setup

The test aims to compare the performances of user mode and kernel mode SMTP relay servers at e-mail level when CPU activities are concerned. Thus, to reduce the effect of round trip time on measurements, only one machine has been used for this test i.e. e-mail sender, SMTP relay server and receiver all are on the same machine. This machine is 3.0 GHz Pentium 4 Dual Core machine with 1GB of memory. Figure 17 shows the setup used for the test.

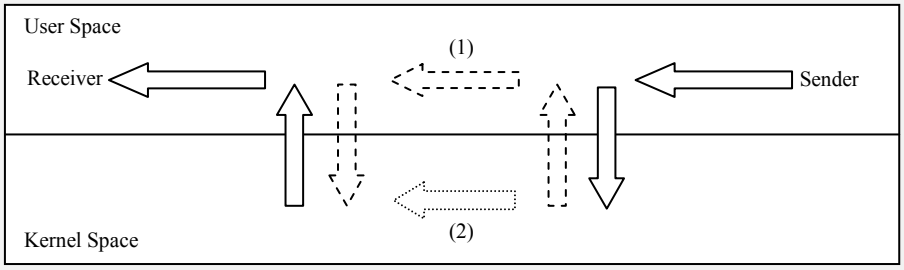


Figure 17: Test Setup

SMTP relay server developed for the test, is a single threaded relay server which first starts SMTP conversation with the sender, accepts an e-mail completely and finishes the conversation. Then, it starts SMTP conversation with the receiver, sends the e-mail to the receiver, finishes the conversation and goes back to receive next e-mail (Figure 18). This cycle

continues until all the e-mails are relayed. For every e-mail, start time and end time are noted. The difference between start time of the first e-mail and end time of the last e-mail is considered as processing time of all the e-mails. The processing times corresponding to both the servers, are compared.

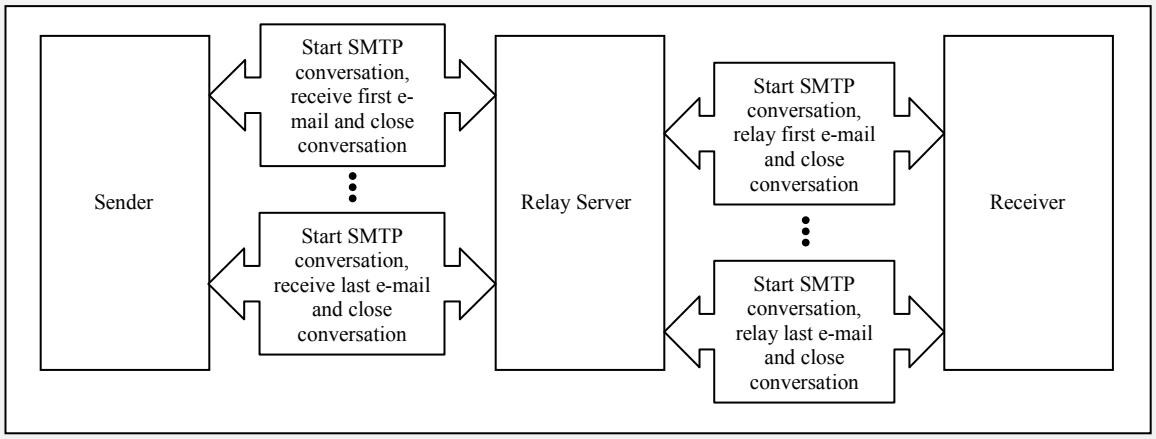


Figure 18: SMTP Relay

4.2 Results

1000 e-mails were relayed in each run. Such runs were conducted for e-mails of different sizes (10MB, 1MB, 500KB, 100KB, 50KB) to see the effect of e-mail size on the improvement. 3 runs were conducted for each e-mail size. Results show that kernel mode relay server consistently relayed e-mails in less time. And as expected, performance improvement increased with increase in e-mail size.

Table 1 shows improvement in latency. Kernel mode relay server relayed e-mails in 18% to 48% less time as compared to user mode relay servers. As a result, speedup obtained ranged from about 1.22x to 1.93x. For 50KB e-mails, reduction in latency is about 18% to 20.5%. For 100KB e-mails, it is 27% to 28%. For 500KB e-mails, nearly 31%; for 1MB e-mails, nearly 33% and for 10MB e-mails, reduction is about 47% to 48%.

Table 1: Kernel-User Mode SMTP Relay Servers Performance Comparison (Latency)

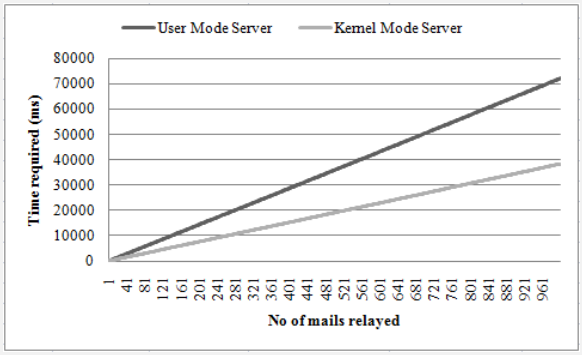
E-mail Size	Reading 1 (1000 e-mails)			Reading 2 (1000 e-mails)			Reading 3 (1000 e-mails)		
	User Mode Latency (ms)	Kernel Mode Latency (ms)	Improve-ment (%)	User Mode Latency (ms)	Kernel Mode Latency (ms)	Improve-ment (%)	User Mode Latency (ms)	Kernel Mode Latency (ms)	Improve-ment (%)
			Speedup			Speedup			Speedup
10MB	72077.90	38340.59	46.81	73183.09	37889.59	48.23	73119.30	37908.07	48.16
			1.8799			1.9315			1.9289
1MB	6186.87	4109.82	33.57	6176.32	4125.46	33.21	6167.83	4116.70	33.26
			1.5054			1.4971			1.4982
500KB	3420.99	2365.10	30.87	3496.69	2405.15	31.22	3522.32	2434.62	30.88
			1.4464			1.4538			1.4468
100KB	1212.63	869.58	28.29	1176.36	856.34	27.20	1217.37	879.34	27.77
			1.3945			1.3737			1.3844
50KB	908.35	726.20	20.05	888.62	728.51	18.02	893.85	708.98	20.68
			1.2508			1.2198			1.2608

Table 2 shows improvement in throughput. Numbers of e-mails that are completely relayed in given time are compared to measure the improvements in throughput. The time periods used for 10MB, 1MB, 500KB, 100KB, 50KB e-mails are 30 sec, 4 sec, 2 sec, 0.5 sec and 0.5 sec respectively. Numbers of e-mails transferred, by user mode and kernel mode servers, in these time periods are noted and compared. For 10MB e-mails, throughput improved by about 88% to 92%. For 1MB, improvement is about 51%. For 500KB, it is about 45% to 47%. For 100KB, 30% to 33% and for 50KB, it is 21% to 29%. It is observed that speedups resulted due to throughputs improvement are almost same as speedups resulted due to latencies improvement.

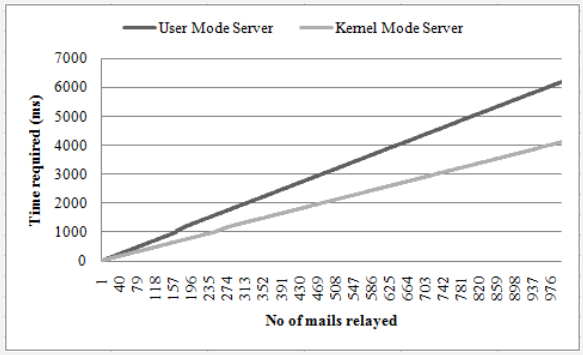
Table 2: Kernel-User Mode SMTP Relay Servers Performance Comparison (Throughput)

E-mail Size	Reading 1			Reading 2			Reading 3		
	User Mode (# e-mails relayed)	Kernel Mode (# e-mails relayed)	Improve-ment (%)	User Mode (# e-mails relayed)	Kernel Mode (# e-mails relayed)	Improve-ment (%)	User Mode (# e-mails relayed)	Kernel Mode (# e-mails relayed)	Improve-ment (%)
			Speedup			Speedup			Speedup
10MB (30 sec)	418	784	87.56	412	792	92.23	412	791	91.99
			1.88			1.92			1.92
1MB (4 sec)	642	972	51.40	644	969	50.47	644	970	50.62
			1.51			1.50			1.51
500KB (2 sec)	583	844	44.77	562	824	46.62	557	811	45.60
			1.45			1.47			1.46
100KB (0.5 sec)	429	572	33.33	439	580	32.12	426	552	29.58
			1.33			1.32			1.30
50KB (0.5 sec)	534	674	26.22	554	670	20.94	549	706	28.60
			1.26			1.21			1.29

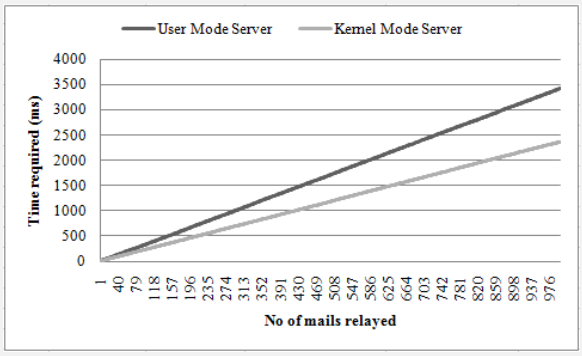
Figure 19 shows graphs (no. of e-mails relayed vs. time required) for the first reading (from table 1) of each e-mail size. These are almost straight line graphs showing constant



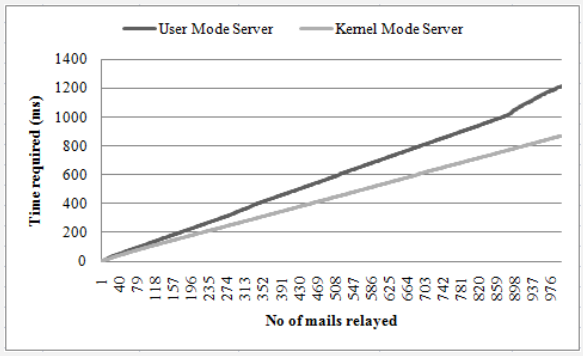
10MB 1000 e-mails



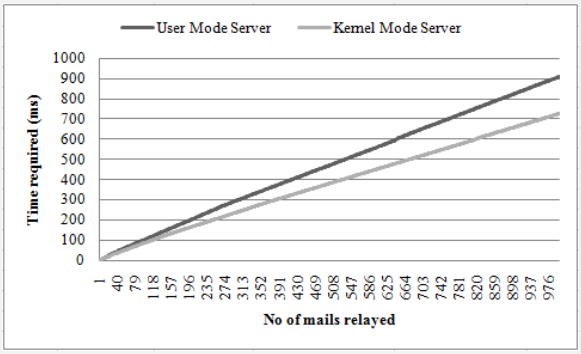
1MB 1000 e-mails



500KB 1000 e-mails



100KB 1000 e-mails



50KB 1000 e-mails

Figure 19: Kernel-User Mode SMTP Relay Servers Performance Comparison (Graphs)

improvement in latency and throughput. This shows that number of e-mails relayed does not

have any impact on the improvement. The nature of the graphs confirms the almost identical speedups, obtained due to improvements in latency and throughput.

4.3 Discussion

Improvement In Latency: The results show that e-mails' processing times can be reduced by avoiding memory copy and system calls. Time required for user mode server to relay 1000 e-mails is more than time required for kernel mode server to relay same number of e-mails, in every case. Hence, it is clear that latency for relaying e-mails is reduced. The improvement in latency is approximately 18% to 48% depending on e-mail size. And resulting speedup varies from 1.22x to 1.93x.

Improvement In Throughput: In case of SMTP relay servers, latency is not of vital importance. In most cases, it doesn't matter if an e-mail is delivered to the recipient a few seconds late. But, if throughput of the server is improved, that allows relaying more e-mails in less time, reducing the need for adding more machines in case of heavily loaded relay servers. We saw that the improvement in throughput is approximately 21% to 92% depending on e-mail size. Resulting speedup varies from 1.21x to 1.92x which is almost identical to speedups resulted due to improvement in latencies.

Improvement Vs. E-mail Size: Runs conducted for different sized e-mails show that for large e-mails, improvement is noticeably better. The trend of improved performance with increase in e-mail size can be clearly seen in results. For 50KB e-mails, throughput improvement is around 21-29% whereas for 10MB e-mails, throughput improvement is about 88-92%. In case of latencies also, improvement for 50KB e-mails is about 18-20.5% and improvement for 10MB e-mails is about 47-48%. This is because as e-mail size increases, percentage of memory copy in overall processing increases. Thus, eliminating memory copy helps e-mails with larger sizes

more than e-mails with smaller sizes. Since the average e-mail size on the Internet has been increasing continuously and the trend will continue for at least some years, SMTP relay servers can really benefit from increase in improvement with increase in e-mail size.

Multithreaded SMTP Servers On Network: Note that the test has been conducted on a single machine (i.e. sender, relay server and receiver all on the same machine) with single threaded relay servers. Hence, RTTs (Round Trip Time) are very small. In real scenarios, there will be multi-threaded servers dealing with much longer RTTs. The results will more or less still hold since the improvement that we have obtained, is a result of freeing CPU from memory copy operations and system calls overheads. Effectively, we are making CPU do less work. Hence, even while dealing with long RTTs, if there is sufficient load on relay servers, saving CPU time will lead to increase in throughput. When one thread will be blocked for data due to longer RTT, CPU can schedule another thread for which data is available; hence, compensating for longer RTT. Context switching between threads may affect the performance improvement a little bit. But, that will not be considerable since even user mode servers will need to context switch whenever data is not available. In case of lightly loaded servers, performance of both the servers (kernel mode and user mode) will be almost same since there may not be enough threads to schedule to compensate for longer RTT. The time saved by avoiding memory copy and system calls, will be wasted in waiting for data. But then, you don't really gain anything by improving throughput of a lightly loaded SMTP server.

CHAPTER 5

SUMMING UP

This chapter presents the conclusion of this work and mentions few ways in which it can be extended.

5.1 Conclusion

This work was aimed at measuring the improvement in latency and throughput of SMTP relay server, achieved by avoiding memory copy and heavy system calls. Results obtained reflected considerable improvement. For small sized e-mails, about 18-20.5% of improvement was observed in latency whereas for large e-mails, the improvement observed reached to approximately 47-48%. In case of throughputs, for small sized e-mails, improvement obtained was about 21-29% whereas for large sized e-mails, improvement was about 88-92%. The improvement was consistent in all the readings. It was argued that even if measurements have been taken on a single machine with a single threaded SMTP relay server, this improvement will more or less reflect in real life scenario too, provided that relay machine is heavily loaded. Thus, these techniques can be useful in developing SMTP servers that can be used in production environment.

5.2 Further Work

This section talks about some ways in which the work in this thesis can be extended.

The SMTP server developed as a part of this work is bare minimum, which just accepts data from one end and relays it to other end. Even though it follows basic SMTP communication

pattern, it does not implement the extension RFCs. One can develop a server that confirms to extensions and then, take the measurements. There are many more RFCs than just the basic RFC, for SMTP. Functionality suggested by these RFCs can be included to measure the improvement for full-fledged SMTP servers.

The measurements can also be improved by actually developing a multi-threaded server and testing it on multiple machines. A key factor in this test is to make sure that relay machine is sufficiently loaded. Otherwise all the CPU time saved by avoiding memory copy and system calls will be wasted in just waiting for the data. Measurements taken this way will reflect the improvement in a real-life scenario.

REFERENCES

[1] Rubini, Alessandro. "Making System Calls From Kernel Space." *Linux Magazine.* November 15, 2000. http://www.linux-mag.com/id/651.

[2] Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel, 3rd Edition.* Sebastopol: O'Reilly Media, Inc., 2005.

[3] Palaniappan, Sathish K., and Pramod B. Nagaraja. "Efficient data transfer through zero copy." *developerWorks: IBM's resource for developers and IT professionals.* September 02, 2008. http://www.ibm.com/developerworks/library/j-zerocopy/.

[4] Bar, Moshe. "kHTTPd, a Kernel-Based Web Server." *Linux Journal.* August 1, 2000. http://www.linuxjournal.com/article/4132.

[5] Ven, Arjan van de. *kHTTPd – Linux HTTP Accelerator.* http://www.fenrus.demon.nl/.

[6] Lever, Chuck, Marius Aamodt Eriksen, and Stephen P. Molloy. *An analysis of the TUX web server.* Technical Report, Ann Arbor: Center for Information Technology Integration, University of Michigan, 2000.

[7] Seth, Sameer, and M. Ajaykumar Venkatesulu. *TCP/IP Architecture, Design and Implementation in Linux.* Hoboken: Wiley-IEEE Computer Society Press, 2008.

[8] Herbert, Thomas F. *The Linux TCP/IP Stack: Networking for Embedded Systems.* Hingham: Charles River Media, Inc., 2004.

[9] Benvenuti, Christian. *Understanding Linux Network Internals.* Sebastopol: O'Reilly Media, Inc., 2005.

[10] Wu, Wenji, Matt Crawford, and Mark Bowden. "The Performance Analysis of Linux Networking – Packet Receiving." *The International Journal of Computer Communications* (Elsevier) 30, no. 5 (2007): 1044-1057.

[11] Free Electrons. *Linux Cross Reference.* http://lxr.free-electrons.com/.

[12] Corbet, Jonathan, Greg Kroah-Hartman, and Alessandro Rubini. *Linux Device Drivers, 3rd Edition.* Sebastopol: O'Reilly Media, Inc., 2005.

[13] Postel, Jonathan B. "RFC 0821: SIMPLE MAIL TRANSFER PROTOCOL." *The Internet Engineering Task Force (IETF).* August 1982. http://www.ietf.org/rfc/rfc0821.txt?number=0821.

[14] Network Working Group. "RFC 2821: Simple Mail Transfer Protocol." *The Internet Engineering Task Force (IETF).* Edited by J. Klensin. April 2001. http://www.ietf.org/rfc/rfc2821.txt?number=2821.

[15] Paxson, Vern. "State Diagrams." *EECS at UC Berkeley.* Fall 2007. http://www.eecs.berkeley.edu/~jortiz/courses/ee122/presentations/StateDiagrams.ppt.

[16] Birkedal, Erlend. *Late data choice with the Linux TCP/IP stack.* Master Thesis, Oslo: University of Oslo, Department of Informatics, 2006.

APPENDIX A

ABBREVIATIONS USED

Table 3: List Of Abbreviations Used In This Thesis

SMTP	Simple Mail Transfer Protocol
CPU	Central Processing Unit
kHTTPd	Kernel HTTP Daemon
TUX	Threaded linUX http layer
HTTP	Hypertext Transfer Protocol
CGI	Common Gateway Interface
TCP	Transmission Control Protocol
IP	Internet Protocol
NIC	Network Interface Card
DMA	Direct Memory Access
MSS	Maximum Segment Size
OOB	Out Of Band
CRLF	Carriage Return followed by Line Feed (“\r\n”)
SP	Space (“ ”)
RTT	Round Trip Time
RFC	Request For Comments