Solutions to Selected Exercises of Open Source Implementations

Open Source Implementation 2.1: 8B/10B Encoder

Exercises

Find the code segment in 8b10_enc.vhd related to 3B/4B coding switch in Figure 2.14 and show us which line of code controls the output timing, i.e., falling or rising edge of the *clk* signal.

Answer (1 hour):

Line 270:

elsif SBYTECLK'event and SBYTECLK ='0' //0 means falling edge

Open Source Implementation 2.2: IEEE 802.11a Transmitter with OFDM

Exercises

Calculate the output bits and states when one encodes these bits using the convolutional encoder in Figure 2.46. Summarize in Table 2.9 how the state and output value change with each iteration.

Answer (2 hours):

Iteration	1	2	3	4	5	6	7	8	9	10
Input bit	0	1	1	0	1	1	0	0	0	0
Shift Regs [543210]	000000	000000	100000	110000	011000	101100	110110	011011	001101	000110
Output [A,B]	00	11	10	10	11	01	00	01	00	10

Open Source Implementation 3.1: Checksum

Exercises

1. The TTL field of an IP packet is subtracted by 1 when the IP packet passed through a router, and thus the checksum value after the subtraction must be changed. Please find an efficient algorithm to re-compute the new checksum value. (Hint: see RFC 1071 and 1141)
2. Explain why the IP checksum does not cover the payload in its computation.

Answer (1.5 hours):

1. See RFC 1071 and 1141 for details.

   Let C be the original checksum and C' be the new one. Let m be the original 16-bit integer that consists of the TTL field and the protocol id field (see the IPv4 header) and m' be the integer of the same fields but with TTL decreased by 1. For 2's complement machines, the new 1's complement of the checksum can be computed using follow equation:

   ~C' = ~(C + (-m) + m') = ~C + (m - m') = ~C + m + ~m'

2. IP checksum only provides extra protection on the IP header. The payload is left to the transport layer to protect.

Open Source Implementation 3.2: CRC32

Exercises

1. Could the algorithm in eth_src.v be easily implemented in software? Justify your answer.
2. Why do we use CRC-32 rather than the checksum computation in the link layer?

Answer (1 hour):

1. Yes, but the performance is not as good due to its bit-oriented operations where each operation would cost an instruction cycle if implemented in software.
2. CRC is more robust to a number of errors than checksum and easy to implement in hardware.

Open Source Implementation 3.3: Link-Layer Packet Flows in Call Graphs

Exercises

Explain why the CPU load could be lowered if using the new net_rx_action()function at high traffic loads.

Answer (1 hour):

With the old net_rx_action()function, each arriving frame would trigger a hardware interrupt, which increases CPU load during heavy traffic. With the new net_rx_action()function, only the first frame of a burst of frames would trigger an interrupt. For the subsequent frames, the kernel calls net_rx_action() to poll the arriving frames.

Open Source Implementation 3.4: PPP Drivers

Exercises

Discuss why the PPP functions are implemented in software, while the Ethernet functions are implemented in hardware.

Answer (0.5 hour):

There are no time-critical operations in PPP, while in Ethernet there are several time-critical operations such as inter-frame gap, jamming time, and back-off time. Only time-critical operations need to be implemented in hardware.

Open Source Implementation 3.5: CSMA/CD

Exercises

1. If the Ethernet MAC operates in the full-duplex mode (very common at present), which components in the design should be disabled?

2. Since the full-duplex mode has a simpler design than the half-duplex mode, and the former's efficiency is higher than the latter's, why do we still bother implementing half duplex mode in the Ethernet MAC?

Answer (0.5 hour):

1. In the TX module, disable the monitoring of CarrierSense and Collision signals.
2. The Ethernet interface might be attached to a hub instead of a switch. If it is a hub, the interface must work in the half-duplex mode.

Open Source Implementation 3.6: IEEE 802.11 MAC Simulation with NS-2

Exercises

1. Why is the send() function called from recv()?
2. Why should a sending frame wait for a random period of time?

Answer (1 hour):

1. recv() handles an incoming frame from both physical layer and upper layer, and send() is called by recv() when there is a frame to transmit.
2. To reduce the probability of repeated collisions during retransmissions.

Open Source Implementation 3.7: Self-Learning Bridging

Exercises

1. Trace the source code and find out how the aging timer works.
2. Find out how many entries are there in the fdb hash table of your Linux kernel source.

Answer (1 hour):

1. We use the aging timer to set the length of time that an entry can stay in the MAC address table, from the time the entry was used or last updated.
2. #define BR_HASH_BITS 8
   #define BR_HASH_SIZE (1 << BR_HASH_BITS)
   Thus, the size is 2^8 = 256.

Open Source Implementation 3.8: Spanning Tree

Exercises

1. Briefly describe how the BPDU frame is propagated along the topology of spanning trees.
2. Study the br_root_selection() function to see how a new root is selected.

Answer (2 hours):

1. The root bridge generates a hello BPDU to its children periodically while the other switches receive the BPDU from the root port, update the topology information, and the forward the BPDU to the other ports.

2. This function, br_root_selection() in net/bridge/br_stp.c, selects the root port of a bridge. The function iterates over all ports, starting with the smallest port number, and it checks for whether the conditions for the root port are met (br_should_become_root_port()).Subsequently, the path cost to the root bridge is compared. If the costs are equal, then the information from the net_bridge_port structure is considered.

Open Source Implementation 3.9: Probing I/O ports, Interrupt Handling and DMA

Exercises

1. Explain how tasklet is scheduled by studying the tasklet_schedule() function call.
2. Enumerate a case in which polling is preferable than interrupting.

Answer (2 hours):

1. The scheduled tasklets are held in two per-processor structures (linked lists of tasklet_struct structures): tasklet_vec (regular) and tasklet_hi_vec (high priority). To schedule a tasklet use tasklet_schedule():
   - If state is TASKLET_STATE_SCHED, it is already scheduled, so the function can return.
   - Save state of interrupt system, and disable local interrupts.
   - Add tasklet to head of the tasklet_vec or tasklet_hi_vec, which is unique to each processor on the system.
   - Raise the TASKLET_SOFTIRQ or HI_SOFTIRQ so tasklet can execute in the near future by do_softirq().
   - Restore interrupts to their previous state and return.
2. Consider a router that is connected to a WAN via a channel service unit/data service unit (CSU/DSU). The router and CSU/DSU may be connected via a V.35 interface cable. If a loss of physical connectivity occurs between the router and the CSU/DSU (say the cable is broken or has been pulled out inadvertently), the router software should be signaled. Interrupts appear to be the best option here. However, spurious and transient loss of physical connectivity should be distinguished from the permanent loss of connectivity. So the communications software may need to poll for the status of the connection periodically once it has been signaled via the interrupt about the loss of connectivity.

Open Source Implementation 3.10: The Network Device Driver in Linux

Exercises

1. Explain how the frame on the network device is moved into the sk_buff structure

(see ne2k_pci_block_input()).

2. Find out the data structure in which a device is registered.

Answer (1.5 hours):

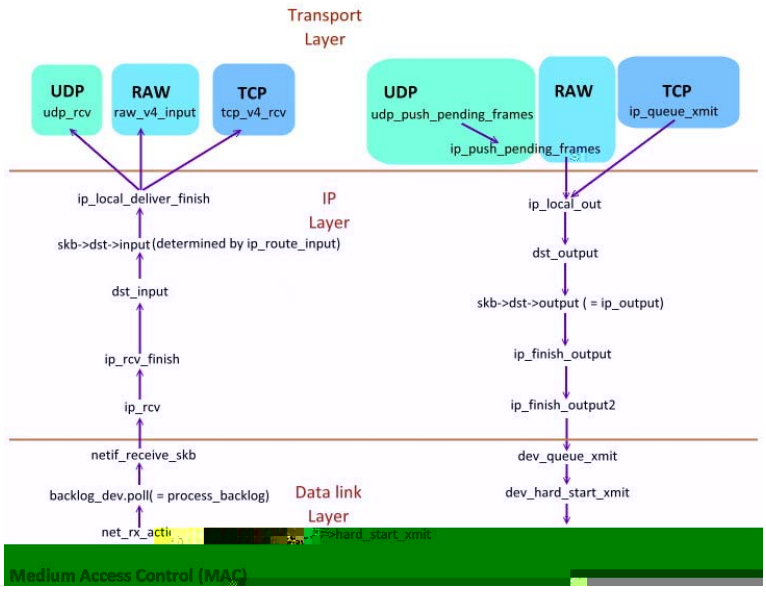
1. When the network interface receives the frame, it will notify the kernel with an interrupt. The kernel then calls the corresponding handler, ei_interrupt(). The ei_interrupt() function determines which type the interrupt is, and calls the ei_receive() function because the interrupt stands for frame reception. The ei_receive() function will call ne2k_pci_block_input() to move the frame from the network interface to the system memory and fill the frame into the sk_buff structure. The netif_rx() function will pass the frame to the upper layer, and the kernel then proceeds to the next task.
2. The data structure in which a device is registered is: net_device
   The net_device data structure is associated with the information about a network device. When a network interface is initialized, the space of this structure for that interface is allocated and registered.

Open Source Implementation 4.1: IP-Layer Packet Flows in Call Graphs

Exercises

Trace the source code along the reception path and transmission path to observe the details of function calls on these two paths.

Answer (1.5 hours):



Reception path

- In net_dev_init(), the queue->backlog_dev.poll is initialized as follows:queue->backlog_dev.poll = process_backlog.
- net_rx_action() is the interrupt handling routine for the interrupt

NET_RX_SOFTIRQ. It will check the poll_list to see if a device if waiting for polling. If yes, the registered routine for the device is called; otherwise, the system will call the default routine process_backlog().

- process_backlog() will call__skb_dequeue to retrieve SKB from the device, and then call netif_receive_skb().
- netif_receive_skb() will decide when to send the packet. If forwarding is required, netif_receive_skb() will pass the packet to the bridge. Otherwise, the packet is passed to process routine for upper layer protocols. For example, it will call ip_rcv() to pass the packet to the IP protocol.
- ip_rcv() will call the NF_HOOK function. When it finishes, it will call the ok_fn() which is link to the ip_rc_finish() function.
- In ip_rcv_finish(), ip_route_input() is called to perform routing. If the result of routing is to forward the packet to next hop (router), then ip_forward() is called. Otherwise, the input pointer points the ip_local_deliver() function.
- In ip_local_deliver(), there is also a NF_HOOK function. It eventually calls the ip_local_deliver_finish() function (hooked to ok_fu()).
- The ip_local_deliver_finish() will the upper layer protocol function to further process the packet. The upper layer protocol can be found by **skb**->**nh.iph**->**protocol()**. Finally, it uses following statement to call the upper layer protocol handler: ret = ipprot->handler(skb). For example, for UDP, the handler is udp_rcv(). Therefore, it will call the udp_rcv() function.

UDP:

- udp_sendmsg() calls dp_push_pending_frames() for simple encapsulation.
- udp_push_pending_frames() first calls ip_push_pending_frames(). After that , it calls ip_local_out()- > __ip_local_out- > dst_output.

TCP:

- When sending data through a TCP socket, tcp_sendmsg() is called to send data in units of segment.
- tcp_sendmsg() first checks if the data needs to be sending immediately (even the size of data is less than MSS) using the forced_push() function. If yes, it calls tcp_sendmsg()->__tcp_push_pending_frames->tcp_write_xmit()->tcp_transmit_skb(). Otherwise, it call tcp_push_one()->tcp_transmit_skb()-> icsk->icsk_af_ops->queue_xmit(skb, 0)->ip_queue_xmit.

- The ip_queue_xmit() function will also call ip_local_out().

- Eventually, the dst_output() function is called which calls

skb->dst->output(skb). For IP packets, it will cann ip_output().

- The ok_fn of one of the NF_HONOK_COND hooks of ip_output() is ip_finish_output().
- ip_finish_output() will then call ip_finish_output2()-> hh->hh_output(skb). After calling hh->hh_output(skb), dev_queue_xmit()->dev_hard_start_xmit().dev_hard_start_xmit() is called to check if GSO(Generic Segmentation Offload) is required. GSO denotes offload the segmentation operation to the network interface card (NIC). If not, or no segmentation is required, it will then call dev->hard_start_xmit(skb) which hands the segment to the NIC. Otherwise, the packet is divided into several segments.

Open Source Implementation 4.2: IPv4 Packet Forwarding

Exercises

1. Use an example to trace *__ip_route_output_key() and write down how* routing cache is searched.

2. Trace

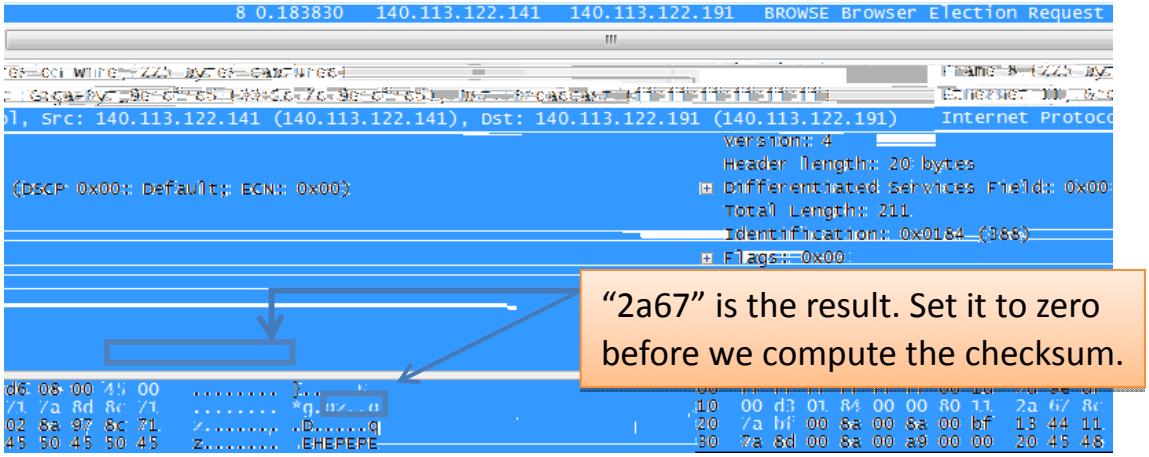
__ip_route_output_key()

- rt_hash_code : returns a hash value which will be used by __ip_route_output_key() to look for routing information from the cache.
- ip_route_output_slow: it calls fib_lookup() to look up the routing table and store the result into cache.
- fib_lookup: will look up the routing table.
- The ip_route_output_key() function will call rt_hash_code() first to obtain a hash value. The rt_hash_code() will use source and destination addresses as the input to the hash function. The hash value is then used to search the rt_hash_table, i.e., the routing cache. If found, it returns 0. If not, ip_r

```
        /* Compute Internet Checksum for "count" bytes
         *              beginning at location "addr".
         */
    register long sum = 0;
     while( count > 1 )    {
        /*    This is the inner loop */
            sum += * (unsigned short) addr++;
            count -= 2;
    }
        /*    Add left-over byte, if any */
    if( count > 0 )
            sum += * (unsigned char *) addr;
        /*    Fold 32-bit sum to 16 bits */
    while (sum>>16)
        sum = (sum & 0xffff) + (sum >> 16);
    checksum = ~sum;
}
```

(2)

We use the captured packets by Wireshark for reference.



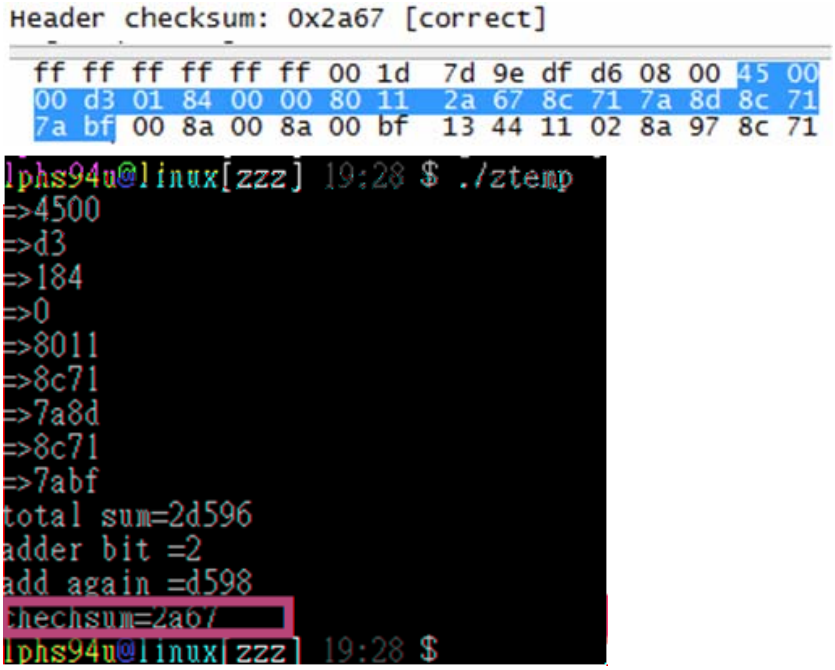
How to compute:

(1) Divide the header into 16-bit blocks. Use 2's complement addition to add all 16-bit blocks (8 blocks), store the result in a 32-bit word.

(2) Add the carry (bits higher than the 16th bit) back to the 16-bit result

(3) Compute the 1's complement of the 16-bit result

Example:



(1)

4 5 0 0

0 0 d 3

0 1 8 4

0 0 0 0

8 0 1 1

0 0 0 0

8 c 7 1

7 a 8 d

8 c 7 1

+) 7 a b f

---

2 d 5 9 6

(2)

d 5 9 6

+) 0 0 0 2

---

d 5 9 8

(3)

~(d 5 9 8)= 2 a 6 7

Open Source Implementation 4.4: IPv4 Fragmentation

Exercises

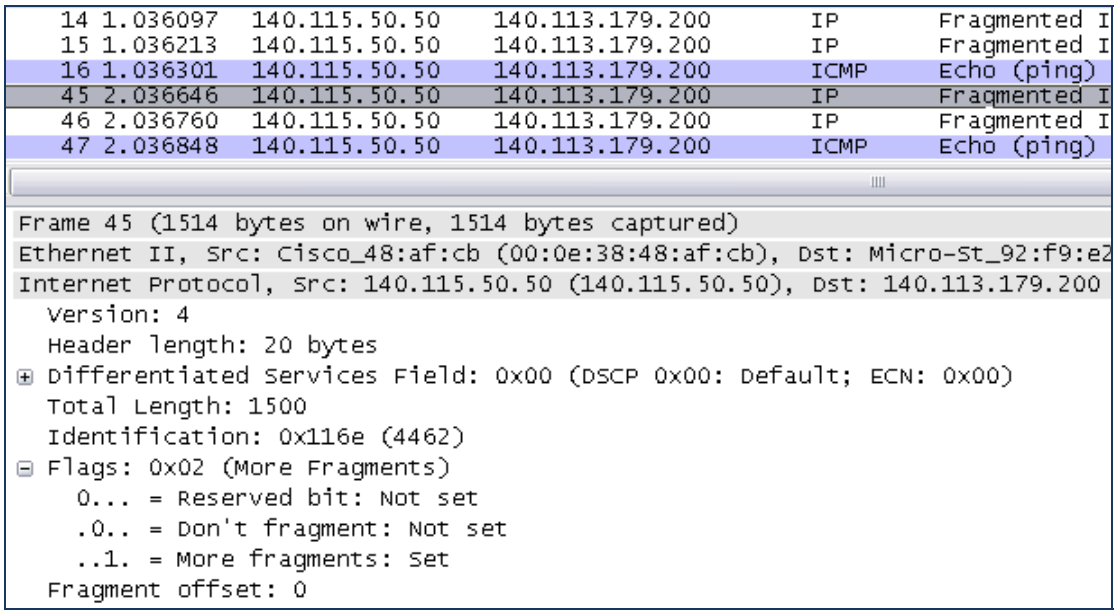
Use wireshark to capture some IP fragments and observe identifier, more flag, and offset fields in their headers.

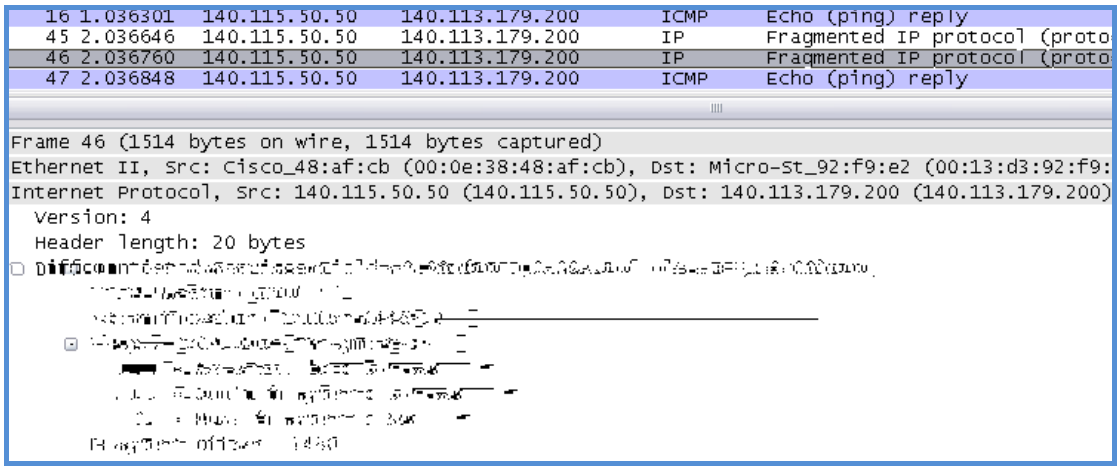
Answer (1 hour):

In the following example, we observe a packet with identification 0x116e is fragmented into three fragments (frame 45~47). As we can see from the captured fragments, the more bit of the first two fragments are set to 1 while that of the last fragment is set to zero. Offest of these three fragments are 0, 1480, 2960, respectively.

➢

	Identification	Flag	Offset
1	0x116e(4462)	02	0
2	0x116e(4462)	02	1480
3	0x116e(4462)	00	2960





```
   45 2.036646   140.115.50.50    140.113.179.200       IP       Fragmented IP protoc
   46 2.036760   140.115.50.50    140.113.179.200       IP       Fragmented IP protoc
   47 2.036848   140.115.50.50    140.113.179.200       ICMP     Echo (ping) reply

Frame 47 (1082 bytes on wire, 1082 bytes captured)
Ethernet II, Src: Cisco_48:af:cb (00:0e:38:48:af:cb), Dst: Micro-St_92:f9:e2 (00:13:
Internet Protocol, Src: 140.115.50.50 (140.115.50.50), Dst: 140.113.179.200 (140.113
  Version: 4
  Header length: 20 bytes
⊞ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
  Total Length: 1068
  Identification: 0x116e (4462)
⊟ Flags: 0x00
    0... = Reserved bit: Not set
    .0.. = Don't fragment: Not set
    ..0. = More fragments: Not set
  Fragment offset: 2960
```

Open Source Implementation 4.5: NAT

Exercises

Trace *adjust_tcp_sequence() and explain how to adjust sequence number* of TCP packets when packets are changed due to address translation.

Answer (1.5 hours):

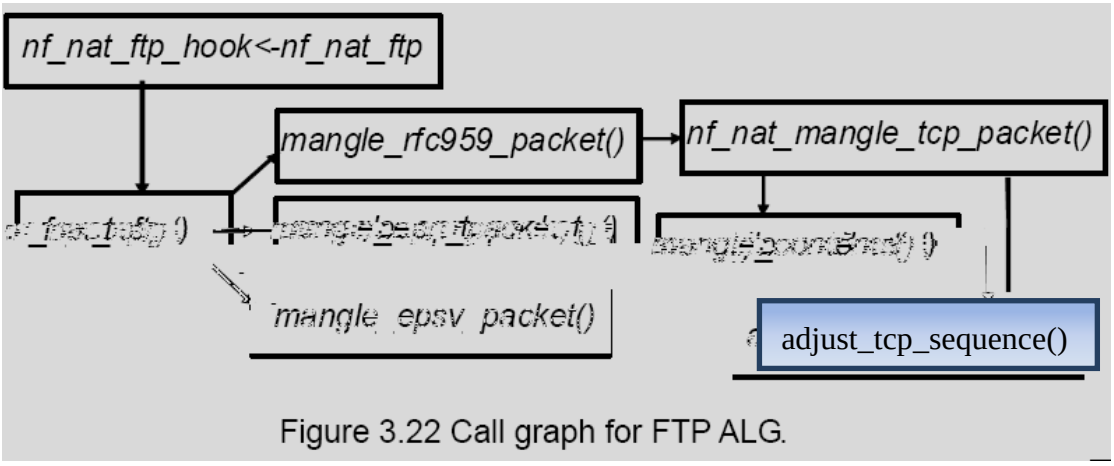


Figure 3.22 Call graph for FTP ALG.

- As seen in the above figure, mangle_rfc959_packet() will modify the FTP commands according to the new IP address and port number. It then calls nf_nat_mangle_tcp_packet() to modify the packet content. If the length of the new packet is different from the original packet, it sets the IPS_SEQ_ADJUST_BIT flag and then calls adjust_tcp_sequence().
- In *adjust_tcp_sequence()*, this_way is a variable of data type ip_nat_seq.

It first checks the following condition:

if (this_way->offset_before == this_way->offset_after || before(this_way->correction_pos, seq))

- If offset_before==offset_after, it means the packet has not been initialized;
- The before() function will calculate and check whether correction_pos – seq is less than 0; if yes,it means seq is larger than correction_pos, the packet needs to be corrected.

If the condition is true,

- this_way->correction_pos = seq;
  Set correction_pos to seq,
- this_way->offset_before = this_way->offset_after;
  Set offset_before to offset_after;
- this_way->offset_after += sizediff;
  The offset_after is increased by rep_len - match_len.

```
struct nf_nat_seq{
  position of the last TCP sequence
        number modification
    u_int32_t correction_pos;
 /* sequence number offset before
   and after last modification */
 int16_t offset_before, offset_after;
 }
```

Open Source Implementation 4.6: ARP

Exercises

The function, *__neigh_lookup()*, is a common function which implements hash buckets. Use free text search or cross reference tool to find out which functions call *__neigh_lookup()*. Trace *neigh_lookup()* and explain how to lookup an entry from hash buckets.

Answer (1 hour):

(1) arp_process() function calls __niegh_lookp() function to search the hash_buckets using source IP address as the hash key.

(2) *arp_process()* first calls *__niegh_lookp()* to find the corresponding entry in the arp table. It then calls *neigh_update()* to update the status of this entry.

◆ Data structure of a Hash Table

A hash table consists of an array of buckets, each bucket consists of a list of slots, each slot can store a record of data.

In *neigh_lookup()*, the hash(pkey, dev) is called to obtain the index of the bucket where pkey is the source IP address and dev is the network interface device. *hash_buckets[hash_val]* is the list of slots which have records that have the same hash value. By matching the pkey with the primary key of each slot, the correct record will be returned if match is found. Otherwise, it returns NULL.

Source code of neigh_lookup() is given as follows:

```
static inline struct neighbour *
__neigh_lookup(struct neigh_table *tbl, const void *pkey, struct net_device *dev, int creat)
{
        struct neighbour *n = neigh_lookup(tbl, pkey, dev);

        if (n || !creat)
                return n;

        n = neigh_create(tbl, pkey, dev);
        return IS_ERR(n) ? NULL : n;
}

struct neighbour *neigh_lookup(struct neigh_table *tbl, const void *pkey,
                               struct net_device *dev)
{
        struct neighbour *n;
        int key_len = tbl->key_len;
        u32 hash_val = tbl->hash(pkey, dev) & tbl->hash_mask;

        NEIGH_CACHE_STAT_INC(tbl, lookups);

        read_lock_bh(&tbl->lock);
        for (n = tbl->hash_buckets[hash_val]; n; n = n->next) {
                if (dev == n->dev && !memcmp(n->primary_key, pkey, key_len)) {
                        neigh_hold(n);
                        NEIGH_CACHE_STAT_INC(tbl, hits);
                        break;
                }
        }
        read_unlock_bh(&tbl->lock);
        return n;
```

360 }

dev: device (Driver Model device interface)
pkey: source IP address

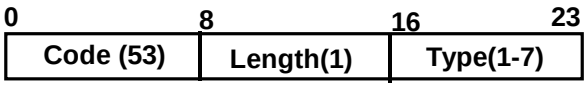
Open Source Implementation 4.7: DHCP

Exercises

Trace *ic_bootp_recv()* and explain how the option field of DHCP is processed. Search IETF RFC documents to find out newly defined DHCP options after RFC 2132.

Answer (1.5 hours):

(1) The additional configuration information is handled by *ic_do_bootp_ext()*. Currently, only code 1 (subnet mask), 3 (default gateway), 6(DNS server), 12(host name), 15(domain name), 17(root path), 26(interface MTU), 42(NIS domain name), are processed.



Let us use subnet mask as an example. In ic_bootp_recv(), it calls ic_do_bootp_ext() with a parameter *opt which points to the address of the “Code” field of the header of DHCP option field. The ic_do_bootp_ext() functionuses a switch statement to process the code (i.e., *opt). Based on the code, it then pass the type field as the parameter to be passed to the external function. For example, for code=1 (i.e., subnet mask), it calls memcpy(&ic_netmask, ext+1, 4) to set the pointer to the Type field.

(2)

Code:	0	Pad Option
Code:	1	Subnet Mask
Code:	2	Time Offset
Code:	3	Routers
Code:	4	Time Server Option
Code:	5	Name Server Option
Code:	6	Domain Name Servers
Code:	7	Log Server Option
Code:	8	Cookie Server Option
Code:	9	LPR Server Option
Code:	10	Impress Server Option
Code:	11	Resource Location Server Option
Code:	12	Host Name
Code:	13	Boot File Size Option
Code:	14	Merit Dump File
Code:	15	Domain Name
Code:	16	Swap Server
Code:	17	Root Path

Code: 18 Extensions Path
Code: 19 IP Forwarding Enable/Disable Option
Code: 20 Non-Local Source Routing Enable/Disable
Code: 21 Policy Filter Option
Code: 22 Maximum Datagram Reassembly Size
Code: 23 Default IP Time-to-Live
Code: 24 Path MTU Aging Timeout Option
Code: 25 Path MTU Plateau Table Option
Code: 26 Interface MTU
Code: 27 All Subnet are Local Option
Code: 28 Broadcast Address Option
Code: 29 Perform Mask Discovery Option
Code: 30 Mask Supplier Option
Code: 31 Perform Router Discovery Option
Code: 32 Router Solicitation Address Option
Code: 33 Static Route Option
Code: 34 Trailer Encapsulation Option
Code: 35 ARP Cache Timeout Option
Code: 36 Ethernet Encapsulation Option
Code: 37 TCP Default TTL Option
Code: 38 TCP Keepalive Interval Option
Code: 39 TCP Keepalive Garbage Option
Code: 40 NIS Domain Name
Code: 41 NIS Option
Code: 42 Network Time Protocol Servers Option

Code:	70	POP3 Server Option
Code:	71	NNTP Server Option
Code:	72	Default WWW Server Option
Code:	73	Default Finger Server Option
Code:	74	Default IRC Server Option
Code:	75	StreetTalk Server Option
Code:	76	SYMA Server Option
Code:	255	End Option

Open Source Implementation 4.8: ICMP

Exercises

Write a pseudo code for the traceroute program given that you are able to call the ICMP functions in the kernel.

Answer (0.5 hour):

```
Procedure traceroute {
    For (ttl=1; ttl<256; ttl++) {
      Send an ICMP echo request message to the destination with TTL=ttl;
      If an ICMP echo reply message is received {
        exit(0); //destination has reached
      }
      else if an ICMP time exceeded message is received {
        printout the source address of the ICMP time exceeded message (router)
          and the latency from the packet until the ICMP message is received
      }
    else check unexpected error
    }
}
```

Open Source Implementation 4.9: RIP

Exercises

Trace route_node_get() and explain how to find the route_node based on the prefix.

Answer (0.5 hour):

Source code: /zebra-0.95a/lib/table.c

The route_node_get() function will retrieve the routing information from the routing table. Two parameters are passed to the function: table (struct route_table *table) and p (struct prefix *p). In this function, three variable of data type struct route_node* are declared: new, node, and match. node is set to table -> top. The prefix_match(&node->p) function is used to check if the prefix is same as the node's prefix. The p.prefixlen is used to check if the node exist.

Open Source Implementation 4.10: OSPF

Exercises

Trace the source code of Zebra and explain how the shortest path tree of each area is maintained.

Answer (1 hour):

Source code: /zebra-0.95a/ospfd/ospf_spf.c

The Dijkstra's algorithm is implemented in *ospf_spf_calculate()* (Calculating shortest-path tree for each area). A router will build a shortest path tree rooted at itself. When it receives link state advertisement, it calls *ospf_spf_calculate()*. Based on the algorithm we shown in the text, it maintains a list of nodes to be added to the tree. It calls ospf_spf_next() and ospf_vertex_add_parent() to get the next node to be added to the tree, i.e., the node with minimum cost in the list. It then calls ospf_spf_register() to add the node to the shortest path tree.

After removing the node from the list and adding it to the shortest path tree, it continues calling ospf_spf_next() to get next node until the list is empty.

Open Source Implementation 4.11: BGP

Exercises

In this exercise, you are asked to explore the prefix length distribution of current BGP routing table. First, browsing the URL at http://thyme.apnic.net/current/, you will find some interesting analysis of BGP Routing Table seen by APNIC routers. In particular, "number of prefixes announced per prefix length" will let you know the number of routing entries of a backbone router and the distribution of prefix length of these routing entries.

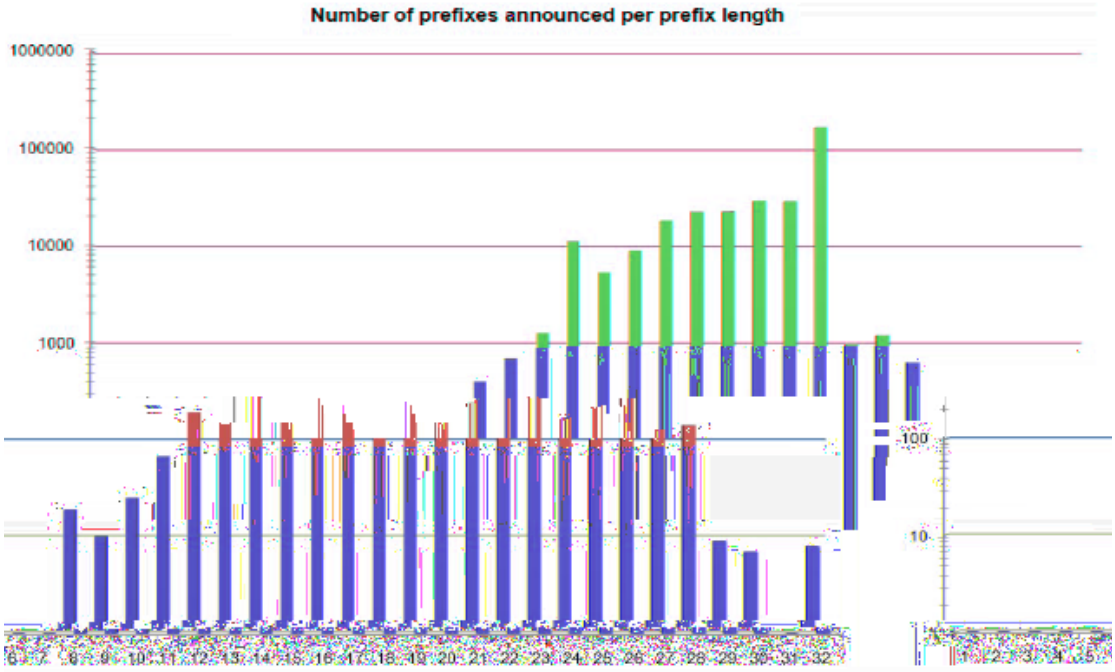
1. How many routing entries does a backbone router own on the day you visit the URL?
2. Draw a graph to show the distribution of prefix length (length varies from 1 to 32) in a logarithmic scale because the number of prefixes announced varies from 0 to tens of thousands.

Answer (0.5 hour):

1. In May 2010, the routing entries are more than 320,000 already.
2. As that of statistics retrieved in May 2010:

```
Number of prefixes announced per prefix length (Global)
-------------------------------------------------------

 /1:0         /2:0         /3:0         /4:0         /5:0         /6:0
 /7:0         /8:19        /9:10        /10:25       /11:67       /12:191
/13:398      /14:688      /15:1263     /16:11059    /17:5268     /18:8959
/19:18275    /20:22497    /21:22635    /22:29398    /23:28962    /24:167645
/25:956      /26:1199     /27:629      /28:141      /29:9        /30:7
/31:0        /32:8
```



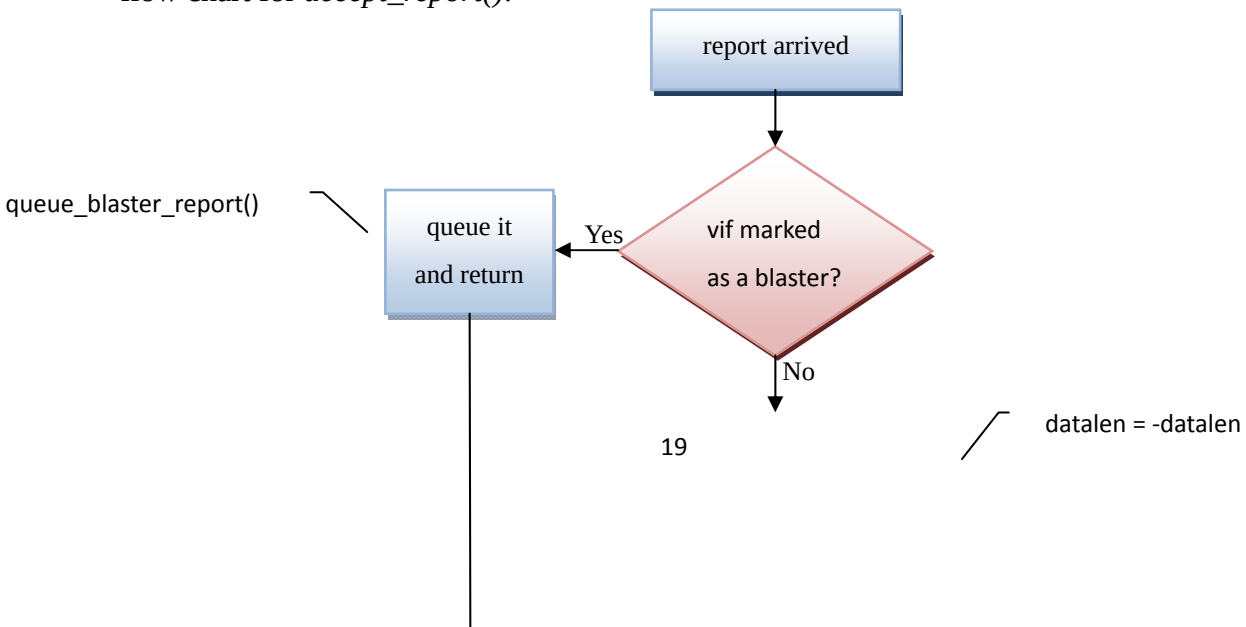
Open Source Implementation 4.12: Mrouted

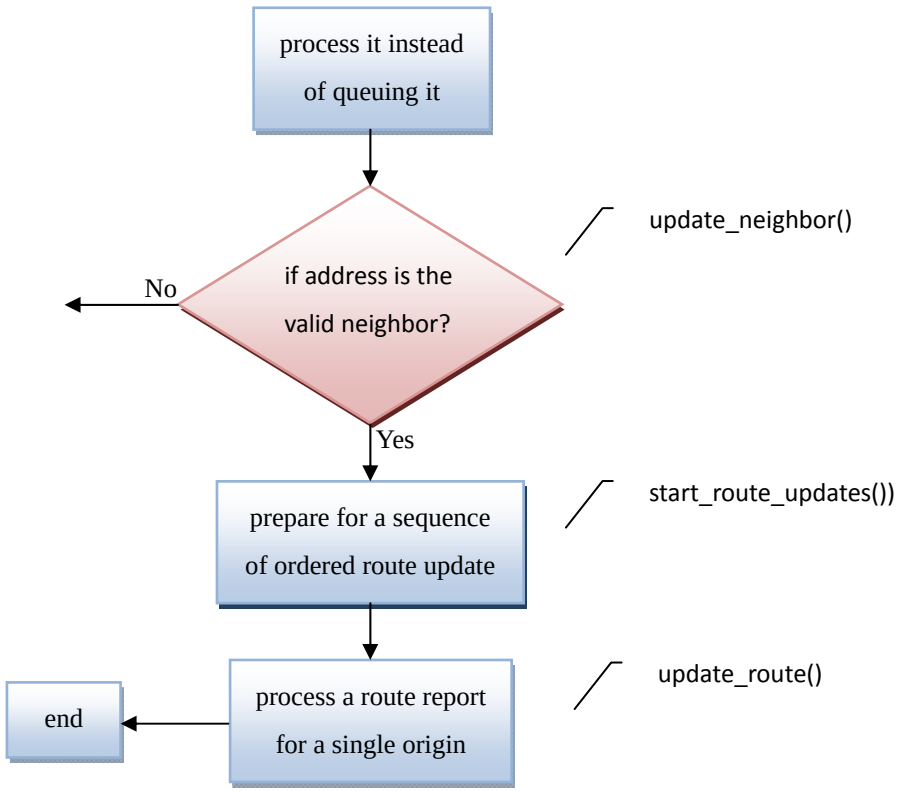
Exercises

Trace the following three functions accept_report(), update_route(), and accept_prune() in the source code of mrouted and draw their flow charts, respectively. Compare the flow charts you draw with the DVMRP protocol introduced in this section.

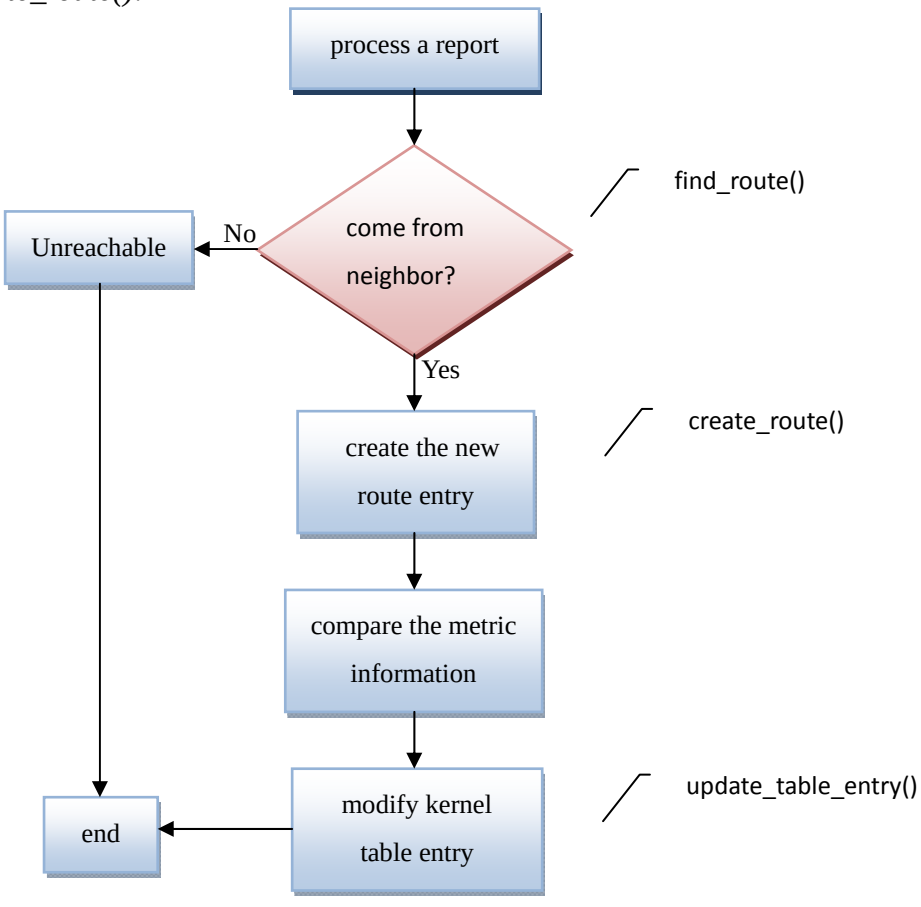
Answer (6 hours):

flow chart for *accept_report()*:

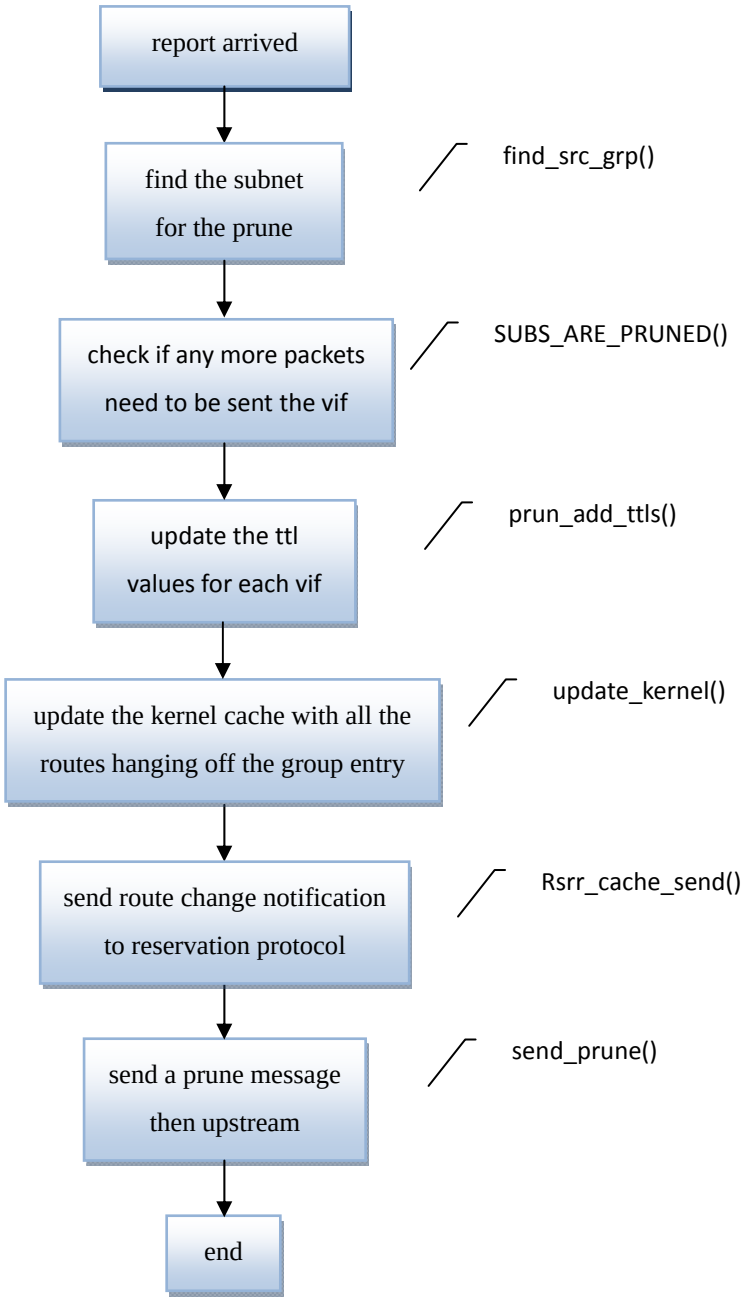




flow chart for *update_route()*:



flow chart for *accept_prune()*:



Open Source Implementation 5.1: Transport-Layer Packet Flows in Call Graphs

Exercises

1. With the call graph shown in Figure 5.3, you can trace udp_sendmsg() and tcp_sendmsg() to figure out how exactly these functions are implemented.
2. Explain what the two big "while" loops in tcp_sendmsg() are intended for? Besides, why are such loop structures not shown in udp_sendmsg()?

Answer (0.5 hour):

tcp_sendmsg():

```
        err = -EPIPE;
        if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))
                goto do_error;
        while (--iovlen >= 0) {
                int seglen = iov->iov_len;
                unsigned char __user *from = iov->iov_base;
                iov++;
                while (seglen > 0) {
                        int copy;
                        skb = tcp_write_queue_tail(sk);
                        ...
                }
        }
```

The first while loop checks if the queue is full. If not, i.e., iovlen >= 0, then it continues. The second while loop checks if the there are still data to be sent, i.e., seglen > 0. If yes, it continues writing the data to the tail of the queue of the socket.

udp_sendmsg():

Since there is no flow control when sending data using UDP, udp_sendmsg() does notcheck whether the queue is full. It simply sends the data to the queue of the socket. No while loop is used in udp_sendmsg().

Open Source Implementation 5.2: UDP and TCP Checksum

Exercises

If you look at the definition of sk_buff in the sk_buff, you may find its memory space is shared with another two variables: csum_start and csum_offset. Could you figure out the usages of the two variables and why both variables share the same 4-byte space with csum?

Answer (3 hours):

- csum_start is the offset from the address of skb->head to the address of the checksum field. csum_soffset is the offset from the beginning of the address of checksum to the end.
- Before version 2.6.22, the Linux kernel sets the csum and csum_offset to be an union data structure (shared memory). The rationale is that they will not be used simultaneously. The csum is a temporary variable for calculating the checksum while csum_offset is the offset of the checksum field after checksum is computed. Therefore, they will not be used simultaneously.
- After version 2.6.22, csum_start,csum_offset, and csum are declared as an union

data structure for the same reason: they will not be used simultaneously. The calculation result temporarily stored in csum will be copied to checksum. Therefore, the 4-byte memory of csum can be used by csum_start and csum_offset(csum_start and csum_offset each requires 16 bits).

Open Source Implementation 5.3: TCP Sliding Window Flow Control

Exercises

In tcp_snd_test(), there is another function tcp_init_tso_segs() called before the three check functions mentioned above. Explain what this function is for.

Answer (1 hour):

```
static int tcp_init_tso_segs(struct sock *sk, struct sk_buff *skb,
                             unsigned int mss_now)
{
        int tso_segs = tcp_skb_pcount(skb);
        if (!tso_segs || (tso_segs > 1 && tcp_skb_mss(skb) != mss_now)) {
                tcp_set_skb_tso_segs(sk, skb, mss_now);
                tso_segs = tcp_skb_pcount(skb);
        }
        return tso_segs;
}
```

TSO denotes “TCP Segmentation Offload.” tcp_init_tso_segs() calls tcp_skb_pcount() to obtain the value of GSO(Generic Segmentation Offload). If tso_segs equals to 0 or it is larger than 1 but the value of GSO is different from MSS, it calls tcp_set_skb_tso_segs() to recalculate the value of tso_segs. The new value of tso_segs is returned as a parameter to the tcp_write_xmit() function. This would allow NIC to know the value of the offload in order to speed up the processing of the packet.

Open Source Implementation 5.4: Tcp Slow Start and Congestion Avoidance

Exercises

The current implementation in tcp_cong.c provides a flexible architecture that allows replacing the Reno’s slow-start and congestion-avoidance with others.

1. Explain how this allowance is achieved.
2. Find an example from the kernel source code which changes the Reno algorithm through this architecture.

Answer (2 hours):

1. To replace the Reno’s congestion control with a new one, we can set new cong_avoid, ssthresh functions into the tcp_reno data structure:

struct tcp_congestion_ops tcp_reno

The tcp_reno_cong_avoid() function starts from line 359 in tcp_cong.c:

```
void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 in_flight)
360{
361             struct tcp_sock *tp = tcp_sk(sk);
362
363             if (!tcp_is_cwnd_limited(sk, in_flight))
364                     return;
365
366             /* In "safe" area, increase. */
367             if (tp->snd_cwnd <= tp->snd_ssthresh)
368                     tcp_slow_start(tp);
369
370             /* In dangerous area, increase slowly. */
371             else if (sysctl_tcp_abc) {
372                     /* RFC3465: Appropriate Byte Count
373                      * increase once for each full cwnd acked
374                      */
375                     if (tp->bytes_acked>=tp->snd_cwnd*tp->mss_cache) {
376                             tp->bytes_acked-=tp->snd_cwnd*tp->mss_cache;
377                             if (tp->snd_cwnd < tp->snd_cwnd_clamp)
378                                     tp->snd_cwnd++;
379                     }
380             } else {
381                     tcp_cong_avoid_ai(tp, tp->snd_cwnd);
382             }
383}
```

This function is set to the cong_avoid field of the tcp_reno data structure.

```
struct tcp_congestion_ops tcp_reno = {
        .flags          = TCP_CONG_NON_RESTRICTED,
        .name           = "reno",
        .owner          = THIS_MODULE,
        .ssthresh       = tcp_reno_ssthresh,
        .cong_avoid     = tcp_reno_cong_avoid,
        .min_cwnd       = tcp_reno_min_cwnd,
```

};

We can do the same change for TCP Vegas. In tcp_vegas.c, we can replace the **cong_avoid** field of the tcp_vegas data structure with the new function..

```
static struct tcp_congestion_ops tcp_vegas = {
        .flags          = TCP_CONG_RTT_STAMP,
        .init           = tcp_vegas_init,
        .ssthresh       = tcp_reno_ssthresh,
        .cong_avoid     = tcp_vegas_cong_avoid,
        .min_cwnd       = tcp_reno_min_cwnd,
        .pkts_acked     = tcp_vegas_pkts_acked,
        .set_state      = tcp_vegas_state,
        .cwnd_event     = tcp_vegas_cwnd_event,
        .get_info       = tcp_vegas_get_info,
        .owner          = THIS_MODULE,
        .name           = "vegas",
};
```

Open Source Implementation 5.5: TCP Retransmit Timer

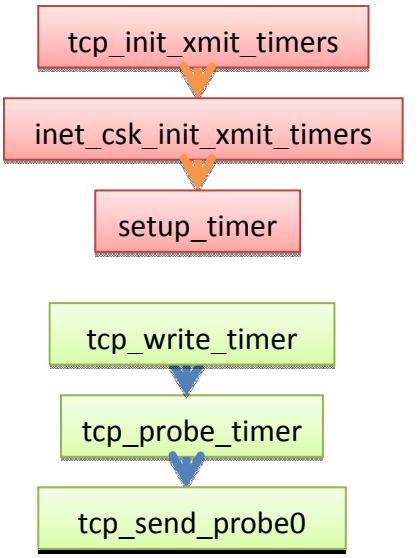
Exercises

Figure 5.27 shows how to update srtt and mdev based on m and their previous values. Then, do you know where and how the initial values of srtt and mdev are given?

Answer (2 hours):

In tcp_clean_rtx_queue(), seq_rtt is set to -1 as follows:

```
static int tcp_clean_rtx_queue(struct sock *sk, int prior_fackets,
3177                                        u32 prior_snd_una)
3178{
3179        struct tcp_sock *tp = tcp_sk(sk);
3180        const struct inet_connection_sock *icsk = inet_csk(sk);
3181        struct sk_buff *skb;
3182        u32 now = tcp_time_stamp;
3183        int fully_acked = 1;
3184        int flag = 0;
3185        u32 pkts_acked = 0;
3186        u32 reord = tp->packets_out;
3187        u32 prior_sacked = tp->sacked_out;
```



- Firstly, we can see that tcp_init_xmit_timers() calls inet_csk_init_xmit_timers(). It is the main function to hook the timer_list.
- In inet_csk_init_xmit_timers(), it calls setup_timer() which in turn calls tcp_write_timer() and tcp_keepalive_timer(). The former hooks the struct timer_list icsk_retransmit_timer while the later hooks the struct timer_list icsk_delack_timer. That is, tcp_keepalive_timer is directly hooked to the timer_list . For tcp_probe_timer(), it is called indirectly through tcp_write_timer(), not directly hooked to the timer_list .
- In net/ipv4/tcp_timer.c,tcp_write_timer() will call tcp_probe_timer(). Specifically, it is the "case ICSK_TIME_PROBEO" of the switch (event) statement. The case is true under zero window probe.

```
static void tcp_probe_timer(struct sock *sk)
{
struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        int max_probes;

        if (tp->packets_out || !tcp_send_head(sk)) {
           /* if tp->packets_out is not zero, the timer is set already */
           /* tcp_send_head() checks if there are data to be sent */
           icsk->icsk_probes_out = 0;
           /* number of probes sent */
           return;
        }
        max_probes = sysctl_tcp_retries2;
```

```
/* set the maximum number of probes to be sent */

if (sock_flag(sk, SOCK_DEAD)) {    /* is the socket closed? */
   const int alive = ((icsk->icsk_rto << icsk->icsk_backoff) < TCP_RTO_MAX);
            /* calculate the value of alive */
   max_probes = tcp_orphan_retries(sk, alive);

   if (tcp_out_of_resources(sk, alive || icsk->icsk_probes_out <= max_probes))
      return;
}
```

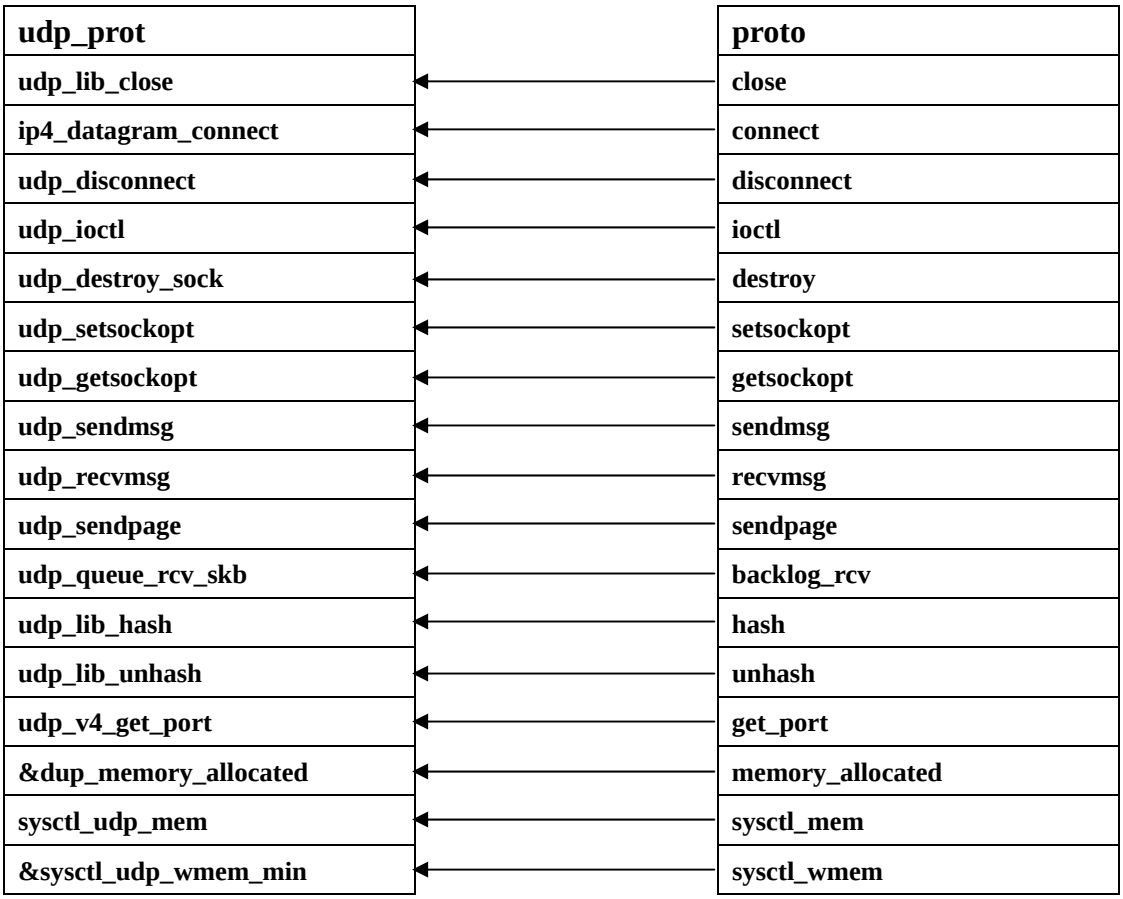
Open Source Implementation 5.7: Socket Read/Write Inside out

Exercises

As shown in Figure 5.41, the structure proto in the structure sock provides a list of function pointers which link to the necessary operations of a socket, e.g. connect, sendmsg, and recvmsg. By linking different sets of functions to the list, a socket can send or receive data over different protocols. Find out and read the function sets of other protocols such as UDP.

Answer (0.5 hour):

UDP: at ipv4/udp.c



&sysctl_udp_rmem_min	←	sysctl_rmem
sizeof(struct udp_sock)	←	obj_size
udp_hash	←	h.udp_hash
compat_udp_setsockopt	←	compat_setsockopt
compat_udp_getsockopt	←	compat_getsockopt

DCCP: at net/dccp/ipv4.c

inet_dccp_ops		proto
PF_INET	←	family
THIS_MODULE	←	owner
inet_release	←	release
inet_bind	←	bind
inet_stream_connect	←	connect
sock_no_socketpair	←	socketpair
inet_accept	←	accept
inet_getname	←	getname
dccp_poll	←	poll
inet_ioctl	←	ioctl
inet_dccp_listen	←	listen
inet_shutdown	←	shutdown
sock_common_setsockopt	←	setsockopt
sock_common_getsockopt	←	getsockopt
inet_sendmsg	←	sendmsg
sock_common_recvmsg	←	recvmsg
sock_no_mmap	←	mmap
sock_no_sendpage	←	sendpage
compat_sock_common_setsockopt	←	compat_setsockopt
compat_sock_common_getsockopt	←	compat_getsockopt

Open Source Implementation 5.8: Bypassing the Transport Layer

Exercises

Modify and compile the above example to dump the fields of the MAC header into a file and identify the transport protocol for each received packet. Note that you need to have the root privilege of the machine to run this.

Answer (1 hour):

```
#include <stdio.h>
#include <unistd.h>
```

```
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <linux/if_ether.h>
#include <net/ethernet.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <net/if.h>
#include <stdlib.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <string.h>
#include <netinet/ip.h>

int main()
{
        int n;
        int fd;
        char buf[2048];

        unsigned char *ethHead;

        struct ether_header *peth;
        struct iphdr *pip;

        if( (fd=socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) == -1 )
        {
                printf("fail to open socket\n");
                return 1;
        }
        while(1)
        {
                n = recvfrom(fd, buf, sizeof(buf), 0, 0, 0);
                ethHead = buf;

                if(n>0)
                        printf("\nrecv %d bytes\n", n);

                /* Adapt the MAC addr. */
                printf("Source MAC addr.:
%02x:%02x:%02x:%02x:%02x:%02x\t", ethHead[0], ethHead[1], ethHead[2],
ethHead[3], ethHead[4], ethHead[5]);
                printf("Dest. MAC addr.:
%02x:%02x:%02x:%02x:%02x:%02x\t", ethHead[6], ethHead[7], ethHead[8],
ethHead[9], ethHead[10], ethHead[11]);

                peth = (struct ether_header *)ethHead;
                ethHead = ethHead+sizeof(struct ether_header);
                pip = (struct iphdr *)ethHead;
                ethHead = ethHead+sizeof(struct ip);
                /* Adapt protocol type */
                switch(pip->protocol)
                {
```

```
                        case IPPROTO_TCP:
                                printf("TCP packets\n");
                                break;
                        case IPPROTO_UDP:
                                printf("UDP packets\n");
                                break;
                        case IPPROTO_ICMP:
                                printf("ICMP packets\n");
                                break;
                        default:
                                printf("Unknown packets\n");
                                break;
                        }
                }
                return 0;
}
```

Experiment results:

```
================================================================
=====
recv 60 bytes
Source MAC addr.: 00:0c:29:5e:02:8d        Dest. MAC addr.: 00:05:5d:f4:c0:57
TCP packets

recv 298 bytes
Source MAC addr.: 00:05:5d:f4:c0:57        Dest. MAC addr.: 00:0c:29:5e:02:8d
TCP packets
================================================================
=====
```

Reference:

http://lazyflai.blogspot.com/2009/02/linuxsniffer.html

http://blog.csdn.net/haoahua/archive/2008/12/24/3597247.aspx

Open Source Implementation 5.9: Making Myself Promiscuous

Exercises

Take a look on network device drivers to figure out how ndo_change_rx_flags() and ndo_set_rx_mode() are implemented. If you cannot find out their implementations, then where is the related code in the driver to enable the promiscuous mode?

Answer (2 hours):

net/8021q/vlan_dev.c

```
static void vlan_dev_change_rx_flags(struct net_device *dev, int change)
{
        struct net_device *real_dev = vlan_dev_info(dev)->real_dev;
        if (change & IFF_ALLMULTI)
                dev_set_allmulti(real_dev, dev->flags & IFF_ALLMULTI ? 1 : -1);
        if (change & IFF_PROMISC)
                dev_set_promiscuity(real_dev, dev->flags & IFF_PROMISC ? 1 : -1);
}
```

```
static void vlan_dev_set_rx_mode(struct net_device *vlan_dev)
{
    dev_mc_sync(vlan_dev_info(vlan_dev)->real_dev, vlan_dev);
    dev_unicast_sync(vlan_dev_info(vlan_dev)->real_dev, vlan_dev);
}
```

These functions in vlan_dev.c are for implementation of virtual LAN. In vlan_dev_change_rx_flags(), the passed in parameter "change" together with the IFF_PROMISC flag decide whether to change the NIC to promiscuous mode. The actual setting of NIC is done by the dev_set_promiscuity() function.

Open Source Implementation 5.10: Linux Socket Filter

Exercises

If you read the man page of tcpdump, you will find that tcpdump can generate the BPF code in the styles of human readable or C program fragment, according to your given filtering conditions, e.g. tcpdump –dd host 192.168.1.1. Figure out the generated BPF code first. Then, write a program to open a raw socket (see Open Source Implementation 5.8), turn on the promiscuous mode (see Open Source Implementation 5.9), use setsockopt to inject the BPF code into BPF, and then observe whether you indeed receive from the socket only the packets matching the given filter.

Answer (2 hours):

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <linux/if_ether.h>
#include <net/ethernet.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <net/if.h>
#include <stdlib.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <string.h>
#include <netinet/ip.h>
#include <linux/filter.h>

int main()
{
```

```
int n;
int fd;
char buf[2048];

unsigned char *ethHead;

struct ether_header *peth;
struct iphdr *pip;

struct ifreq ethreq;
struct sock_fprog Filter;
struct sock_filter code[] = {
        { 0x28, 0, 0, 0x0000000c },
        { 0x15, 0, 4, 0x00000800 },
        { 0x20, 0, 0, 0x0000001a },
        { 0x15, 8, 0, 0x8c71b3ff },
        { 0x20, 0, 0, 0x0000001e },
        { 0x15, 6, 7, 0x8c71b3ff },
        { 0x15, 1, 0, 0x00000806 },
        { 0x15, 0, 5, 0x00008035 },
        { 0x20, 0, 0, 0x0000001c },
        { 0x15, 2, 0, 0x8c71b3ff },
        { 0x20, 0, 0, 0x00000026 },
        { 0x15, 0, 1, 0x8c71b3ff },
        { 0x6, 0, 0, 0x00000060 },
        { 0x6, 0, 0, 0x00000000 },
};

if( (fd=socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) == -1 )
{
printf("fail to open socket\n");
        return 1;
}
/* set promiscuous mode*/
strncpy(ethreq.ifr_name, "eth0", IFNAMSIZ);
ioctl(fd, SIOCGIFFLAGS, &ethreq);
ethreq.ifr_flags |= IFF_PROMISC;
ioctl(fd, SIOCGIFFLAGS, &ethreq);

/* BPF */
Filter.len = 14;
Filter.filter = code;
setsockopt(fd, SOL_SOCKET, SO_ATTACH_FILTER, &Filter, sizeof(Filter));

while(1)
{
        n = recvfrom(fd, buf, sizeof(buf), 0, 0, 0);
        ethHead = buf;

        if(n>0)
                printf("\nrecv %d bytes\n", n);

        /* Adapt the MAC addr. */
        printf("Source MAC addr.:
%02x:%02x:%02x:%02x:%02x:%02x\t", ethHead[0], ethHead[1], ethHead[2],
ethHead[3], ethHead[4], ethHead[5]);
        printf("Dest. MAC addr.:
```

```
%02x:%02x:%02x:%02x:%02x:%02x\t", ethHead[6], ethHead[7], ethHead[8], ethHead[9], ethHead[10], ethHead[11]);

                peth = (struct ether_header *)ethHead;
                ethHead = ethHead+sizeof(struct ether_header);
                pip = (struct iphdr *)ethHead;
                ethHead = ethHead+sizeof(struct ip);
                switch(pip->protocol)
                {
                case IPPROTO_TCP:
                        printf("TCP packets\n");
                        break;
                case IPPROTO_UDP:
                        printf("UDP packets\n");
                        break;
                case IPPROTO_ICMP:
                        printf("ICMP packets\n");
                        break;
                default:
                        printf("Unknown packets\n");
                        break;
                }
        }
        return 0;
}
```

Results of experiments:

```
================================================================
=====
recv 92 bytes
Source MAC addr.: ff:ff:ff:ff:ff:ff        Dest. MAC addr.: 00:30:6e:d7:a1:eb      U
DP packets

recv 92 bytes
Source MAC addr.: ff:ff:ff:ff:ff:ff        Dest. MAC addr.: 00:30:6e:d7:a1:eb      U
DP packets

recv 92 bytes
Source MAC addr.: ff:ff:ff:ff:ff:ff        Dest. MAC addr.: 00:30:6e:d7:a1:eb      U
DP packets
================================================================
=====
```

Reference:

http://www.360doc.com/content/061028/09/13362_243074.html

Open Source Implementation 6.1: BIND

Exercises

1. Find the .c file and the lines of code that implement the iterative resolution.

2. Find which RRs are looked up in forward query and reverse query, respectively, on one of your local hosts.

3. Retrieve all RRs in your local name server with dig.

Answer (2 hours):

1. bind-9.7.0b3\bin\named\query.c
   - ✓ Implementation can be found in query_find(), at line 3709-5099.
     - The iterative query is processed in line 4120-4188. The following gives the operation overview: When NS receives an iterative query, it checks its local database (cache) to see if it has the answer. If yes, it returns the non-authoritative answer. If not, it returns a list of NSs in its local cache that may know the answer. The requester can send requests to these NSs for the desired answer.

2.

"forward query":

As a forward query example, use the "dig www.cs.nctu.edu.tw" command to request for the IP address of the domain name "www.cs.nctu.edu.tw". The ANSWER SECTION is given as follows:

```
;; ANSWER SECTION:
www.cs.nctu.edu.tw.      44      IN      A       140.113.235.47
```

"reverse query":

As a reverse query example, use "dig –x 140.113.235.47"to query the domain name of the IP address 140.113.235.47. The ANSWER SECTION is given as follows:

```
;; ANSWER SECTION:
47.235.113.140.in-addr.arpa. 229179 IN  PTR     wwwcs.cs.nctu.edu.tw.
```

(3) To get all domain records (if allowed by administrator):

dig domain-name axfr

Open Source Implementation 6.2: qmail

Exercises

1. Find the .c files and the lines of code that implement qmail-smtpd, qmail-remote, and qmail-pop3d.

2. Find the exact structure definition of the qmail queue in an object of the qmail structure.

3. Find how e-mails are stored in the mailbox and mail directory.

Answer (4 hours):

1.

- implementing qmail-smtpd:
  - ➢ qmail-smtpd.c 中
    - ✓ line 65-69 "smtp_greet(code)"
    - ✓ line 70-73 "smtp_help(arg)"

- ✓ line 74-77 “smtp_quit(arg)”
- ✓ line 225-229 “smtp_helo(arg)”
- ✓ line 230-234 “smtp_ehlo(arg)”
- ✓ line 235-239 “smtp_rset(arg)”
- ✓ line 240-249 “smtp_mail(arg)”
- ✓ line 250-265 “smtp_rcpt(arg)”
- ✓ line 368-395 “smtp_data(arg)”
- ✓ line 411-421 “main()”

- implementing qmail-remote:
  - qmail-remote.c
    - ✓ line 89-93 “outhost()”
    - ✓ line 97-104 “dropped()”
    - ✓ line 134-141 “get(ch)”
    - ✓ line 166-176 “outsmtptext()”
    - ✓ line 178-190 “quit(prepend,append)”
    - ✓ line 219-274 “smtp()”
    - ✓ line 279-309 “addrmangle(saout,s,flagalias,flagcname)”
    - ✓ line 311-327 “getcontrols()”
    - ✓ line 329-427 “main(argc,argv)”
- implementing qmail-pop3d:
  - qmail-pop3d.c
    - ✓ line 149-162 “pop3_stat(arg)”
    - ✓ line 164-170 “pop3_rset(arg)”
    - ✓ line 172-178 “pop3_last(arg)”
    - ✓ line 180-197 “pop3_quit(arg)”
    - ✓ line 210-218 “pop3_dele(arg)”
    - ✓ line 255-274 “pop3_top(arg)”
    - ✓ line 290-306 “main(argc,argv)”

2.

qmail queue is defined as one of the records of the data structure “struct qmail”; it is declared to be char buf[1024].

3.

The difference between mailbox and mail directory is that the former stores all mails in a file while the later stores one mail in one file and all mails (files) in one directory.

Open Source Implementation 6.3: Apache

Exercises

1. Find which .c file and lines of code implement prefork. When is prefork invoked?
2. Find which .c file and lines of code implement cookie persistence
3. Find which .c files and lines of code implement HTTP request handling and response preparation.

Answer (1 hour):

1.

Implemented in Server/mpm/prefork.c(Line 1343):

static void prefork_hooks(apr_pool_t *p)

Invoked in Server/mpm/prefork.c (Line 1489):

```
module AP_MODULE_DECLARE_DATA mpm_prefork_module = {
    MPM20_MODULE_STUFF,
    ap_mpm_rewrite_args,        /* hook to run before apache parses args */
    NULL,                       /* create per-directory config structure */
    NULL,                       /* merge per-directory config structures */
    NULL,                       /* create per-server config structure */
    NULL,                       /* merge per-server config structures */
    prefork_cmds,               /* command apr_table_t */
    prefork_hooks,              /* register hooks */
};
```

2.

Modules/metadata/Mod_usertrack.c (line 208)

static int spot_cookie(request_rec *r)

3.

In Modules/metadata/Mod_headers.c

Line 499: header_cmd().

Open Source Implementation 6.4: wu-ftpd

Exercises

1. How and where are the control and data connections of an FTP session handled concurrently? Are they handled in the same process or two processes?
2. Find which .c file and lines of code implement active mode and passive mode. When is the passive mode invoked?

Answer (2 hours):

1. When there is a need for data transfer, such as file transfer or list of a directory, the data connection is established. During the data transfer, both data and control connections will co-exist. The data connection is closed when the data transfer is done. A new data connection will be established when a new data transfer is requested. Both data and control connections are handled by the same process.
2. passive mode：

   The default mode is active mode, so there is no dedicate function for active mode FTP. Implementation of active mode starts from line 567 in the main() function. The passive mode is implemented by the passive(void) function which can be

found in /src/Ftpd.c, line 160.

Open Source Implementation 6.5: Net-SNMP

Exercises

1. Find which .c files and lines of code implement set operation.
2. Find out the exact structure definition of an SNMP session.

Answer (2 hours):

1. The set operation is implemented by the function netsnmp_set() which could be found at line 124 in Client_intf.c.
2.

```
/* Internal information about the state of the snmp session.*/
    struct snmp_internal_session {
        netsnmp_request_list *requests;         /* Info about outstanding requests */
        netsnmp_request_list *requestsEnd;    /* ptr to end of list */
        int                  (*hook_pre) (netsnmp_session *, netsnmp_transport *,
                                          void *, int);
        int                  (*hook_parse) (netsnmp_session *, netsnmp_pdu *,
                                            u_char *, size_t);
        int                  (*hook_post) (netsnmp_session *, netsnmp_pdu *, int);
        int                  (*hook_build) (netsnmp_session *, netsnmp_pdu *,
                                            u_char *, size_t *);
        int                  (*hook_realloc_build) (netsnmp_session *,
                                                    netsnmp_pdu *, u_char **,
                                                    size_t *, size_t *);
        int                  (*check_packet) (u_char *, size_t);
        netsnmp_pdu         *(*hook_create_pdu) (netsnmp_transport *,
                                                 void *, size_t);
        u_char              *packet;
        size_t               packet_len, packet_size;
    };
/* The list of active/open sessions. */
    struct session_list {
      struct session_list *next;
      netsnmp_session *session;
      netsnmp_transport *transport;
      struct snmp_internal_session *internal;
};
```

Open Source Implementation 6.6: Asterisk

Exercises

1. Find which .c file and lines where sip_request_call() is registered as a callback function.

2. Describe the sip_pvt structure and explain important variables in that structure.
3. Find which .c file and lines where the RTP/RTCP transport is establish for the SIP session.

Answer (1 hour):

1. In Chan_sip.c, line 2311:
   requester = **sip_request_call**, /* called with chan unlocked */

2. sip_pvt maintains information of a SIP connection, some of the important variables are:
   a. struct sip_pvt *next;
      Points to the next SIP dialog
   b. AST_STRING_FIELD(callid);
      Stores the caller id
   c. struct sip_socket socket;
      The socket of this SIP dialog.
3. In Rtp.c, line 2504:
   struct ast_rtp *ast_rtp_new_with_bindaddr()
   It establishes a RTP session by calling ast_rtp_new_init(rtp).

Open Source Implementation 6.8: BitTorrent

Exercises

1. Explore the locality by considering the round trip delay and changing the random selection code in the getNextOptimisticPeer() function accordingly. For example,

```
import com.aelitis.azureus.core.dht.speed.impl.DHTSpeedTesterImpl;
long[] RTT = new long[ optimistics.size() ];
ArrayList<PEPeer> RTTpeer = new ArrayList<PEPeer>( optimistics.size() );
//For each peer in the list of optimistic peers, get its RTT and sort the list
//based on RTT, put the result to RTTpeer
for (int i=0;i< optimistics.size();i++ ){
    PEPeer peer = all_peers.get( i );
    potentialPing pp = (potentialPing) optimistic.get(i);
    int newRTT = pp.getRTT()
    updateLargestValueFirstSort( newRTT, RTT, peer, RTTpeer, 0 );
}
//Sequentially output RTTpeer to the list of optimistic peers
for (int i=RTTpeer.size();i=RTTpeer.size()-num_needed;i-- ){

    result.add( RTTpeer.remove( i ));
}
```

2.

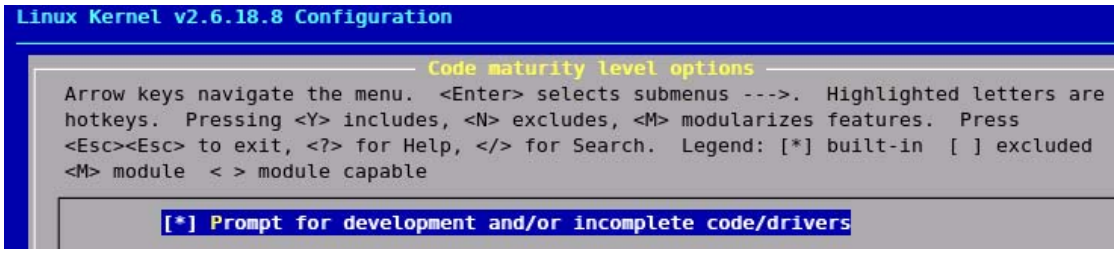
Two peers having shorter RTT implies that they are physically near to each other. When selecting an optimistic peer, we actually give the peer a chance to receive data from us. Since tit-for-tat is based on the amount of upload data from a neghbor peer, we in turn get a better chance to become a tit-for-tat peer of the selected optimistic peer. With goodwill, the selected optimistic peer will become our tit-for-tat peer later. Therefore, considering locality in choosing optimistic unchoked peer also results in better locality of tit-for-tat peers.

Open Source Implementation 7.1: Traffic Control Elements in Linux
Exercises

Could you re-configure your Linux kernel to install the TC modules and then figure out how to setup these modules? In the following open source implementations in this chapter, we shall detail several TC elements related to the text. Thus, it is a good time here to prepare yourself with this exercise. You can find useful references in Further Readings of this chapter.

Answer (1 hour):

Using make menuconfig->Code maturity level options->Prompt for development and /or incomplete code/drivers



Open Source Implementation 7.2: Traffic Estimator

Exercises

1. Could you explain how Line 6 or 10 performs the EWMA operation? What is the value of the historical *parameter w* used in the operation?
2. Could you read gen_estimator.c to find out how the gen_estimator of all flows are grouped? Do you know why the parameter idx is counted from 2?

Answer (1.5 hours):

1.

4: b**rat**e = (nbytes - e->last_bytes)<<(7 - idx);

5: e->last_bytes = nbytes;

6: e->avbps += ((s64)brate - e->avbps) >> e->ewma_log;

7: e->rate_est->bps = (e->avbps+0xF)>>5;

(1) To evaluate EWMA:

avrate = avrate*(1-W) + rate*W

where W is chosen as negative power of 2: W = 2^(-ewma_log)

The resulting time constant is:

T = A/(-ln(1-W))

(2) W is 2^(-ewma_log).

2.

(1) by linked list.

(2) We measure rate over A=(1<<interval) seconds. Minimal interval is HZ/4=250msec (it is the greatest common divisor for HZ=100 and HZ=1024 8)), maximal interval is (HZ*2^EST_MAX_INTERVAL)/4 = 8sec.

Open Source Implementation 7.3: Flow Identification

Exercises

1. Is there any reason that the destination IP address and port number are used in hashing before the source IP address and port number?
2. Could you find what hash function is used for the identification by reading the code in net/sched/cls_rsvp.h?

Answer (2 hours):

1.

Usually, the local host is a client side which connects to a server. Therefore, there could be several local ports that connect to the same server IP and its well known port. If hashing on the source IP and port number, then the result will be unique. As a consequence, there would be no feature of double-level hash. On the other hand, if hashing on the destination IP and port number, more than one session will hash to the same key (value) which is called the first-level hash. These sessions could be distinguished by hashing on the source IP and port number, so called the second-level hash. In this way, flow identification can be done using double-level hash.

2.

The two inline functions, hash_dst and hash_src in cls_rsvp.h, are main hash functions used for identification. Both of them use the variable h which is the bit string of the destination or source address as the key to the hash function. The hash function performs some shift and OR operations on the key and then takes part of the key value as the hash result. For example, in hash_dst, it first performs two shift and OR operations: h ^= h>>16 and h ^= h>>8, and then returns (h ^ protocol ^ tunnelid) & 0xFF) as the hash result.

Open Source Implementation 7.4: Token Bucket

Exercises

As mentioned in the beginning of the data structure, you can find another implementation of token bucket in act_police.c. Explain how the token bucket is implemented for that policer?

Answer (1 hour):

Flow control is done by the tcf_act_police() function. If the data rate of the flow is larger than the threshold, i.e., police->tcf_rate_est.bps >= police->tcfp_ewma_rat, it returns without sending packets. To send a packet, following three conditions must be met:

1. qdisc_pkt_len(skb) <= police->tcfp_mtu
2. police->tcfp_R_tab != NULL
3. (toks|ptoks) >= 0

The first condition requires that the packet length is less than the MTU; the second condition requires that the flow control table must exists; the third condition requires that toks or ptoks must be greater or equal to zero.

Following codes are key implementation of flow control:

```
now = psched_get_time();
toks = psched_tdiff_bounded(now, police->tcfp_t_c,
                                police->tcfp_burst);
if (police->tcfp_P_tab) {
        ptoks = toks + police->tcfp_ptoks;
        if (ptoks > (long)L2T_P(police,police->tcfp_mtu))
                ptoks = (long)L2T_P(police, police->tcfp_mtu);
        ptoks -= L2T_P(police, qdisc_pkt_len(skb));
}
```

"toks" records the accumulated amount of data that can be sent. When tcfp_P_tab is activated, flow can be sent at the peak rate(police->tcfp_ptoks) for a time period of ptoks. If tcfp_P_tab is not activated, it can sent data at mean rate(police->tcfp_burst). When all of the data in the buffer have been sent, the residual time and rate are stored back to toks and ptoks.

"act_police" is implemented similar to sch_tbf, it checks the size of packet and size of the bucket to determine if data can be sent. One of the differences is that act_police adopts spin lock to ensure that its variables will not be changed by other processes.

Open Source Implementation 7.5: Packet Scheduling

Exercises

1. Compared to the complicated PGPS, DRR is much easier both in its concept and implementation. You can find its implementation in sch_drr.c. Please read the code and explain how this simple yet useful algorithm is implemented.

2. There are several implementations of scheduling algorithms in the folder sched. For each implementation, could you find its differentiation from others? Do all of them belong to the fair-queuing scheduling?

Answer (3 hours):

1. DRR is able to schedule multiple queues, it uses drr_class to manage those queues. Each queue is set to different class with different time quantum. Queues are served in a round robin manner, but the amount of data can be served is determined by the time quantum. If the queue is empty, DRR uses drr_change_class() to change the class of the queue and drr_dequeue(struct Qdisc *sch) to remove the queue from service. DRR is able to serve queues with different size of packets. The time quantum can be accumulated if it is not used up at current round.

2. We give three scheduling examples as follows:

sch_fifo.c：FIFO (First In First Out) is the simplest scheduling rule. It does not provide any fairness guarantee to flows.

sch_tbf.c：this implementation adopts token bucket for scheduling. The data rate of each flow can be controlled by the token bucket such that the burst of one flow cannot overwhelm the transmission resource. Fairness could be achieved by proper setting of token bucket parameters, such as token rate and bucket size.

sch_prio.c：this implementation fulfils priority scheduling. Packets with higher priority are send before that of lower priority. Fairness is not considered among different priority queues.

Scheduling algorithms implemented under the sched directory do not all pursue fair-queueing. For example, priority scheduling (sch_prio.c) may give more bandwidth to high priority queues.

Open Source Implementation 7.6: Random Early Detection

Exercises

From /net/sched/ you can find a variant of RED, named generic RED (GRED), implemented in sch_gred.c. Figure out how it works and how it differs from RED?

Answer (2 hours):

GRED is a multi-level RED variant written by Jamal Hadi Salim. Instead of physical queue, it introduces the concept of ”Virtual Queue”(VQ). It can support up to 16 virtual queues. The RED algorithm is then implemented at each VQ. (It actually supports two modes, the “standard mode” has VQ have its own independent average queue estimate while the ‘RIO mode” couples average queue estimates from VQs.) The tc_index of a skb indetifies which VQ this packet belongs to. The prio variable in gred_sched_data does not represent the priority of the packet, it is a control parameter of VQ implementation.

Open Source Implementation 8.1: Hardware 3DES

Exercises

1. Point out which components in the design are likely to be inefficient if it were implemented in software.
2. Find out in the code how the initial 56-bit key is transformed into the 48-bit keys in each of the 16 iterations.

Answer (2.5 hours):

1. Bitwise permutation requires a significant number of cycles for copying and normalization. Components in the design of 3DES include: Bitwise permutations (P-boxes), substitutions (S-boxes), and linear mixing ((+) function). In software even a simple bitwise permutation is relatively tricky and therefore leads to

several lines of code (at least in C/C++).

2.

i. trunk\VHDL\key_schedule.vhd

ii. Def: Ki is the i-th subkey. Ki is transformed by some bit-permutation of the input key (key_input).

We know that the initial 64-bit key is transformed into a 56-bit key by discarding every 8^{th} bit of the initial key. Thus, for each round, a 56-bit key is available. From this 56-bit key, a different 48-bit sub-key is generated during each round using a process called key transformation. For this, the 56-bit key is divided into two halves, each of 28 bits. These halves are circularly shifted left by one or two positions, depending on the round. For example, in rounds 1, 2, 9 or 16, the shift is done by only one position. For other rounds, the circular shift is done by two positions. After an appropriate shift, 48 of the 56 bits are selected. Since the key transformation process involves permutation as well as selection of a 48-bit sub-set of the original 56-bit key, it is called compression permutation. Because of this compression permutation technique, a different subset of key bits gets used in each round which makes DES not so easy to crack.

Open Source Implementation 8.2: MD5

Exercises

1. Numerical values in a CPU may be represented in little endian or big endian. Explain how the md5.c program handles this disparity in representation for the computation.
2. Compared with sha1_generic.c in the same directory, find where and how the sha_tranform() function is implemented. What is the major difference between the implementations of md5_transform() and sha_tranform()?

Answer (2 hours):

1. md5.c uses 2 functions: le32_to_cpu_array and cpu_to_le32_array to handle the disparity in representation for the computation. Both the functions have 2 parameters: buf and words where buf is a buffer used to store a block and words represents the number of words in buf.
2. The sha_transform function is implemented in lib/sha1.c. Major differences between the implementations of *md5_transform* and *sha_tranform* are:

- The SHA-1 is an iterative algorithm that requires 80 transformation steps to generate the final hash value (Message Digest – MD). In each transformation step, a hash operation is performed that takes as inputs five 32-bit variables (*a*,*b*,*c*,*d*,*e*), and two extra 32-bit words (one is the message schedule, *W*t, which is provided by the Padding Unit, and the other word is a constant, *K*t, predefined by

the standard).

- As in SHA-1, MD5 focuses on the transformation of an initial input, through iterative operations. MD5 produces a 128-bit MD, instead of the 160-bit hash value of SHA-1. Additionally, there are still four rounds, consisting however of 16 operations each. There are four 32-bit (*a*,*b*,*c*,*d*) inputs and two extra 32-bit values (one is the message schedule, *M*t, which is provided by the Padding Unit, and the other word is a constant, *L*t, predefined by the standard) that are transformed iteratively to produce the final MD.

Open Source Implementation 8.3: AH and ESP in IPsec

Exercises

1. Find in xfrm_input.c how the xfrm_input function determines the protocol type and calls either the ah_input() or the esp_input() function.
2. Briefly describe how a specific open-source implementation of hash algorithm, eg., MD5 which consists of md5_init, md5_update and md5_final, is executed in *ah_mac_digest* function.

Answer (2.5 hours):

1. The major flow related to calling ah_input or esp_input in xfrm_input is like:

   while(…)

   x = xfrm_state_lookup(net, daddr, spi, nexthdr, family);

   …

   nexthdr = x->type->input(x, skb);

   }

   Here, xfrm_state_lookup returns a variable x, which contains the function pointer, i.e, x->type->input, pointing to either ah_input or esp_input. The return value is determined by the *nexthdr* parameter. Initially, nexthdr is indicated by the caller of xfrm_input. The caller decides the nexthdr by looking up the *protocol* field in an IP packet. For example, when this field has a protocol number 50 (or 51), it indicates the IP packet contains an ESP (or AH) payload. In case a nested IPSec packet is encountered then the x->type->input parses the payload, and returns the *nexthdr* variable matching the *next header* field in the payload.
2. There are three function pointers serving the INIT, UPDATE and FINAL functions of a specific hash algorithm. They are stored by the ahp->tfm variable, for e.g., when using MD5, ahp->tfm->input points to the md5_init function. In ah_mac_digest function, crypto_hash_init, crypto_hash_update and crypto_hash_final invoke the ahp->tfm->input, update and final respectively. This is how a specific hash algorithm is executed in ah_mac_digest function.

Open Source Implementation 8.4: Netfilter and iptables

Exercises

1. Indicate which function is eventually called to match the packet content in the IPT_MATCH_ITERATE macro.
2. Find out where the ipt_do_table() function is called from the hooks.

Answer (1.5 hours):

1. do_match() function
   IPT_MATCH_ITERATE macro matches the packet content
   http://lxr.linux.no/#linux+v2.6.32/net/ipv4/netfilter/ip_tables.c LINE 172
2. The ipv4 netfilter hooks in nf_nat_standalone.c consist of the four functions: nf_nat_in, nf_nat_out, nf_nat_local_fn and nf_nat_fn. They (except nf_nat_fn itself) in turn call nf_nat_fn. Then, nf_nat_fn calls nf_nat_rule_find (in nf_nat_rule.c) which finally calls ipt_do_table (in ip_tables.c).

Open Source Implementation 8.5: FireWall Toolkit (FWTK)

Exercises

1. Find out how the url_parse() and url_compare() functions are implemented in this package.
2. Do you think the approach of rule matching is efficient? What are possible ways to improve the efficiency?

Answer (2.5 hours):

1. url_parse(): parses to identify the scheme; three possibilities: (1) ":" followed by the scheme, (2) "http*:" followed by the scheme, (3) no scheme.
   url_compare(): returns 0 if an identical URL is found; compares pat_s and val_s according to the type of the scheme and checks for port, user name, password, etc. Returns 0 if all matched.
2. Defer the heaviest comparison, say on host name, to the last.

Open Source Implementation 8.6：ClamAV

Exercises

1. Find out how cli_filetype2()called by cli_magic_scandesc() identifies the file types.
2. Find out the number of signatures associated with each file type (or the generic type) in both scanning algorithms in your current version of ClamAV. (Hint: Use 'sigtool' to decompress the ClamAV Virus Databases files (*.cvd) and examine the resulted Extended Signature Format files (*.ndb).)

Answer (1.5 hours):

1. The type=cli_filetype2(*ctx->fmap, ctx->engine) function will call cli_filetype()

to examine the buf of fmap and ftypes of engine by using memcmp() function. After the examination, it can identify the file type of fmap.

2. 545,035 by sigtool –i *.cvd on version 0.95.3

   Where ”sigtool --unpack *.cvd” is used to decode the cvd format to retireve the individual *.ndb file, and ndb files are files that storing signatures.

Open Source Implementation 8.7：Snort

Exercises

1. List five preprocessors in Snort and study the execution flow of each one of them.
2. Find out what multiple-string matching algorithm is used for signature matching in Snort and where the algorithm is implemented.

Answer (1.5 hours):

1. Frag3, Stream5, RPC Decode, DNS, SSL/TLS
2. Aho-Corasick

Open Source Implementation 8.8：SpamAssassin

Exercises

1. Why is SpamAssassin implemented in Perl rather than in C or C++?
2. Discuss the pros and cons of using Bayesian filtering compared with the rule-based approaches.

Answer (0.5 hour):

1. Perl supports regular expression matching which is much needed.
2. Matching one keyword does not imply that the message is definitely a spam. It requires probabilistic calculation to assess the chance of being a spam.

   Remark:

   Bayesian filtering needs supervised learning. Use th (at fth (arai[(n data,rk:)]TJET1 g8828 464