TCP/IP ARCHITECTURE, DESIGN, AND IMPLEMENTATION IN LINUX
Press Operating Committee

Chair
Linda Shafer
former Director, Software Quality Institute
The University of Texas at Austin

Editor-in-Chief
Alan Clements
Professor
University of Teesside

Board Members
Mark J. Christensen, Independent Consultant
Phillip Laplante, Associate Professor Software Engineering, Penn State University
Richard Thayer, Professor Emeritus, California State University, Sacramento
Donald F. Shafer, Chief Technology Officer, Athens Group, Inc.
James Conrad, Associate Professor UNC Charlotte
Janet Wilson, Product Manager, CS Press

IEEE Computer Society Publications
The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available from most retail outlets. Visit the CS Store at http://computer.org/cspress for a list of products.

IEEE Computer Society / Wiley Partnership
The IEEE Computer Society and Wiley partnership allows the CS Press authored book program to produce a number of exciting new titles in areas of computer science, computing and networking with a special focus on software engineering. IEEE Computer Society members continue to receive a 15% discount on these titles when purchased through Wiley or at wiley.com/ieeeecs

To submit questions about the program or send proposals please e-mail jwilson@computer.org or write to Books, IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314. Telephone +1-714-821-8380.

Additional information regarding the Computer Society authored book program can also be accessed from our web site at http://computer.org/cspress.
CONTENTS

Preface xxvi
Acknowledgments xxvii

1 INTRODUCTION 1

1.1 Overview of TCP/IP Stack 2
1.1.1 Moving Down the Stack 3
1.1.2 Moving Up the Stack 5

1.2 Source Code Organization for Linux 2.4.20 5
1.2.1 Source Code Organization for Networking Code 7

1.3 TCP/IP Stack and Kernel Control Paths 7

1.4 Linux Kernel Until Version 2.4 Is Non-preemptible 11
1.4.1 System Call on Linux 14
1.4.2 Adding New System Call 16

1.5 Linux Process and Thread 17
1.5.1 fork() 17
1.5.2 Thread 18
1.5.3 Kernel Threads 19

1.6 Kernel Synchronization Mechanism 22
1.6.1 Semaphore 22
1.6.2 Atomic Operations 23
1.6.3 Spin Lock 23

1.7 Application Interfaces for TCP/IP Programming 24
1.7.1 Server Application 25
1.7.2 Client Application 27
1.7.3 Socket Options 29
1.7.4 Option Values 29

1.8 Shutdown 35
1.8.1 Kernel Shutdown Implementation 36
1.8.2 Send Shutdown 36
1.8.3 Receive Shutdown 36

1.9 I/O 38
1.9.1 read() 38
1.9.2 write() 38
1.9.3  recv()  38
1.9.4  send()  39
1.9.5  select()  39
1.10  TCP State  39
  1.10.1  Partial Close  45
  1.10.2  tcpdump Output for Partial Close  47
1.11  Summary  48

2  PROTOCOL FUNDAMENTALS  49
  2.1  TCP  50
    2.1.1  TCP Header  50
  2.2  TCP Options (RFC 1323)  54
    2.2.1  mss Option  55
    2.2.2  Window-Scaling Option  55
    2.2.3  Timestamp Option  56
    2.2.4  Selective Acknowledgment Option  57
  2.3  TCP Data Flow  58
    2.3.1  ACKing of Data Segments  58
  2.4  Delayed Acknowledgment  67
  2.5  Nagle’s Algorithm (RFC 896)  69
  2.6  TCP Sliding Window Protocol  72
  2.7  Maximizing TCP Throughput  79
  2.8  TCP Timers  82
    2.8.1  Retransmission Timer  82
    2.8.2  Persistent Timer  83
    2.8.3  Keepalive Timer  84
    2.8.4  TIME_WAIT Timer  85
  2.9  TCP Congestion Control  85
  2.10  TCP Performance and Reliability  86
    2.10.1  RTTD  86
    2.10.2  SACK/DSACK  86
    2.10.3  Window Scaling  87
  2.11  IP (Internet Protocol)  87
    2.11.1  IP Header  88
  2.12  Routing  90
  2.13  netstat  90
  2.14  traceroute  92
    2.14.1  traceroute Mechanism  93
  2.15  ICMP  93
  2.16  ping  95
  2.17  ARP/RARP  97
  2.18  Summary  99
4.3.8 State of the Connection Request when the Three-Way Handshake Is Still Pending 150
4.3.9 State of the Connection Request when the Three-Way Handshake Is Completed 151
4.4 Connection Request Handling by Kernel 151
4.4.1 SYN Queue Processing 155
4.4.2 Accept Queue Processing 155
4.4.3 Flow Control for Handling a New Connection Request 156
4.5 Accept 156
4.5.1 *inet_accept()* 159
4.5.2 Linking of Inode and Socket Data Structures when the Three-Way Handshake Has Completed and Is Accepted by Application 161
4.5.3 Linking of VFS and Socket Data Structures in the Kernel when a New Connection Is Established 162
4.5.4 File Table Entry of a New Accepted Connected Socket 162
4.5.5 Flow Control for Accepting New Established Connections 162
4.6 Client Side Setup 163
4.6.1 Client Side Operations 164
4.6.2 Connect 164
4.6.3 *tcp_v4_connect()* 167
4.6.4 *ip_route_connect()* 167
4.6.5 Flow Control for Generating a Connection Request 167
4.6.6 *tcp_v4_hash_connect()* 170
4.6.7 *__tcp_v4_check_established()* 171
4.6.8 *tcp_connect()* 174
4.6.9 *tcp_transmit_skb()* 176
4.7 Summary 178

5 *sk_buff* AND PROTOCOL HEADERS 181
5.1 struct *sk_buff* 182
5.2 struct *skb_shared_info* 186
5.3 *sk_buff* and DMA—SKB_FRAG_STRUCT 187
5.3.1 DMA and Fragmented *sk_buff* Containing Paged Data 188
5.3.2 *sk_buff* and IP Fragmentation 188
5.3.3 *sk_buff* and Fragmentation 190
5.4 Routines Operating on *sk_buff* 190
5.4.1 *alloc_skb()* 190
5.4.2 *skb_reserve()* 191
5.4.3 *skb_put()* 192
5.4.4 *skb_push()* 194
5.4.5 *skb_pull()* 195
5.5 *sk_buff* Builds Protocol Headers as It Traverses Down the Protocol Layers

5.5.1 Tcp Header Is Added to *sk_buff*

5.5.2 Ip Header Is Added to *sk_buff*

5.5.3 Link Layer Header Is Added to *sk_buff*

5.6 *sk_buff* Extracts Protocol Headers as It Traverses Up the Protocol Layers When a Packet Arrives

5.6.1 *sk_buff* Is Made to Point to a Datalink Layer Header Which Will Be Processed by a Datalink Driver

5.6.2 *sk_buff* Is Made to Point to an ip Layer Header Which Will Be Processed by an IP Layer

5.6.3 *sk_buff* Is Made to Point to a tcp Layer Header Which Will Be Processed by a tcp Layer

5.7 Summary

6 **MOVEMENT OF *sk_buff* ACROSS PROTOCOL LAYERS**

6.1 Packet Traversing Down the TCP/IP Stack

6.1.1 Path of Packet Traversal from Socket Layer to Device for Transmission

6.1.2 Kernel Path for TCP Packet Traversing Down the Stack

6.2 Routed Packet Ready for Transmission

6.3 Kernel Flow for a Packet Moving Down the Stack

6.4 Packet Traversing Up the TCP/IP Stack

6.4.1 Path of Packet Traversal from Device (Reception) to Socket Layer

6.4.2 Kernel Path for TCP Packet Traversing Up the Stack

6.5 Kernel Flow for a Packet Moving Up the Stack

6.6 Summary

7 **TCP SEND**

7.1 TCP Segmentation Unit for Sending Data

7.1.1 Functioning of Segmentation Unit without Scatter–Gather Support

7.1.2 Segmentation without Scatter–Gather Support

7.1.3 1 mss of Data Written over the Socket

7.2 Segmentation with Scatter–Gather Technique

7.2.1 Segmentation with Scatter–Gather Support

7.2.2 Application Writes Y Bytes over the Socket

7.2.3 *can_coalesce()*

7.2.4 *tcp_copy_to_page()*

7.2.5 *tcp_mark_push()*

7.2.6 *forced_push()*
7.2.7 tcp_push()
7.2.8 __tcp_push_pending_frames()
7.2.9 tcp_snd_test()
7.2.10 tcp_nagle_check()
7.2.11 tcp_minshall_check()
7.2.12 tcp_write_xmit()
7.2.13 update_send_head()
7.2.14 tcp_push_one()
7.2.15 skb_entail()

7.3 Sending OOB Data

7.4 Flow for TCP Segmentation Unit and Send Process

7.5 Functional Level Flow for Segmentation and Send Mechanism

7.6 Summary

8 TCP RECEIVE

8.1 Queuing Mechanism
8.1.1 Processing in tcp_rcv_established()
8.1.2 tcp_prequeue()
8.1.3 Processing of Queues
8.1.4 tcp_data_wait()
8.1.5 tcp_prequeue_process()
8.1.6 lock_sock()
8.1.7 __lock_sock()
8.1.8 release_sock()
8.1.9 __release_sock()

8.2 Processing of TCP Data from the Receive Queue
8.2.1 cleanup_rbuf()
8.2.2 skb_copy_datagram_iovec()
8.2.3 Reading Data from Receive Buffer without Paged Data Area
8.2.4 X Bytes Requested from the Application
8.2.5 1 mss = n Bytes Requested from the Application
8.2.6 n – X Bytes Requested from the Application
8.2.7 Consumption of Data from a Paged Buffer
8.2.8 n Bytes Requested by the Application
8.2.9 One Page of Data Requested by the Application

8.3 TCP Urgent Byte Processing
8.3.1 Urgent Byte Read as OOB Data
8.3.2 tcp_recv_urg()
8.3.3 Urgent Mode Processing and Reading an Urgent Byte as Inline Data
9 TCP MEMORY MANAGEMENT

9.1 Transmit Side TCP Memory Management
  9.1.1 select_size() 294
  9.1.2 tcp_alloc_skb() 295
  9.1.3 alloc_skb() 296
  9.1.4 tcp_alloc_page() 297
  9.1.5 skb_charge() 298
  9.1.6 tcp_mem_schedule() 298
  9.1.7 tcp_free_skb() 300
  9.1.8 sock_wfree() 300
  9.1.9 tcp_write_space() 301
  9.1.10 tcp_mem_reclaim() 302
  9.1.11 tcp_mem_reclaim() 302
  9.1.12 wait_for_tcp_memory() 303

9.2 Receive Side TCP Memory Management
  9.2.1 tcp_prune_queue() 308
  9.2.2 tcp_clamp_window() 309
  9.2.3 tcp_collapse_ofo_queue() 311
  9.2.4 tcp_collapse() 312
  9.2.5 skb_queue_purge() 317

9.3 Freeing of Memory Allocated to a Receive Buffer
9.4 System-Wide Control Parameters Are Worth Noticing When It Comes to TCP Memory Management

9.5 Summary 321

10 TCP TIMERS

10.1 Timers in Linux
  10.1.1 mod_timer() 324
  10.1.2 detach_timer() 325
  10.1.3 del_timer() 325
  10.1.4 When Are Timer Routines Executed? 326

10.2 TCP Retransmit Timer
  10.2.1 When Do We Set Retransmit Timer? 327
  10.2.2 When Do We Reset or Cancel Retransmit Timers? 327
  10.2.3 tcp_enter_loss() 330
  10.2.4 tcp_retransmit_skb() 333
  10.2.5 tcp_retrans_tryCollapse() 334
  10.2.6 skb_cloned() 336
10.3 Zero Window Probe Timer

10.3.1 When Is the First Time Probe Timer Installed? 337
10.3.2 When Is the Probe Timer Canceled for the Connection? 337
10.3.3 tcp_ack_probe() 338
10.3.4 How Does the Window Probe Timer Work? 338
10.3.5 tcp_probe_timer() 339
10.3.6 tcp_send_probe0() 339
10.3.7 tcp_write_wakeup() 339

10.4 Delay ACK Timer

10.4.1 When Is the ACK Scheduled? 342
10.4.2 How and When Is the ACK Segment Sent? 344
10.4.3 Quick ACK Mode 345
10.4.4 __tcp_ack_snd_check() 345
10.4.5 tcp_ack_snd_check() 346
10.4.6 tcp_send_delayed_ack() 347
10.4.7 tcp_delack_timer() 348
10.4.8 tcp_reset_xmit_timer() 349
10.4.9 tcp_write_timer() 351
10.4.10 tcp_clear_xmit_timer() 352

10.5 Keepalive Timer

10.5.1 When Is the Keepalive Timer Activated? 353
10.5.2 How Is the Timer Reset? 354
10.5.3 tcp_keepalive_timer() 354

10.6 SYN-ACK Timer

10.6.1 When Is the SYN-ACK Timer Activated? 356
10.6.2 When Is the SYN-ACK Timer Stopped? 357
10.6.3 tcp_synack_timer() 357

10.7 TIME_WAIT Timer

10.7.1 When Do We Trigger TIME_WAIT Timer? 361
10.7.2 tcp_time_wait() 362
10.7.3 tcp_tw_schedule() 362
10.7.4 Non-recycle Mode 363
10.7.5 Recycle Mode 365
10.7.6 tcp_twkill() 367
10.7.7 tcp_twcal_tick() 370
10.7.8 __tcp_tw_hashdance() 374

10.8 Summary 375

11 TCP CORE PROCESSING 377

11.1 TCP Incoming Segment Processing

11.1.1 Prediction Flags 378
11.1.2 Building Prediction Flags 379
11.1.3 Condition to Enable the Fast Path 380
11.1.4 When to Enable the Slow Path 382
11.1.5 When to Enable the Fast Path 382
11.1.6 Points to Remember about Prediction Flags 383
11.2 Fast Path Processing 384
11.3 Slow Path Processing 386
11.3.1 tcp_sequence() 387
11.3.2 tcp_replace_ts_recent() 387
11.3.3 tcp_event_data_recv() 390
11.3.4 tcp_incr_quickack() 391
11.3.5 tcp_grow_window() 392
11.3.6 __tcp_grow_window() 393
11.3.7 How Do We Calculate Window to Be Advertised? 394
11.3.8 tcp_receive_window() 395
11.3.9 __tcp_select_window() 395
11.3.10 tcp_space() 397
11.3.11 tcp_data_snd_check() 397
11.3.12 __tcp_data_snd_check() 398
11.3.13 tcp_paws_discard() 398
11.4 Processing of Incoming ACK 400
11.4.1 tcp-packets_in_flight() 403
11.4.2 tcp_ack_is_dubious() 404
11.4.3 tcp_cong_avoid() 405
11.4.4 tcp_ack_update_window() 406
11.4.5 tcp_may_update_window() 407
11.4.6 tcp_clean_rtx_queue() 408
11.5 Processing of SACK blocks 410
11.5.1 tcp_sacktag_write_queue() 410
11.6 Reordering Length 417
11.7 Processing TCP Urgent Pointer 421
11.7.1 tcp_check_urg() 422
11.8 Processing Data Segments in Slow Path 424
11.8.1 tcp_sack_new_ofo_skb() 433
11.8.2 tcp_sack_maybe_coalesce() 434
11.8.3 tcp_sack_extend() 435
11.8.4 tcp_ofo_queue() 436
11.8.5 tcp_sack_remove() 441
11.9 Overview of Core TCP Processing 442
11.10 Summary 442
12 TCP STATE PROCESSING

12.1 Overview of State Processing 446
12.2 TCP States 448
  12.2.1 TCP_CA_CWR 449
  12.2.2 Undoing from TCP_CA_CWR 449
12.3 Processing of Duplicate/Partial ACKs in Recovery State 449
  12.3.1 tcp_remove_reno_sacks() 450
  12.3.2 tcp_try_undo_partial() 451
12.4 Processing of Duplicate/Partial ACKs in Loss State 452
  12.4.1 tcp_try_undo_loss() 453
  12.4.2 tcp_check_sack_reneging() 455
12.5 Default Processing of TCP States 456
  12.5.1 tcp_time_to_recover() 459
  12.5.2 tcp_head_timedout() 460
  12.5.3 tcp_try_to_open() 461
  12.5.4 tcp_update_scoreboard() 462
  12.5.5 tcp_xmit_retransmit_queue() 464
  12.5.6 tcp_packet_delayed() 466
12.6 Processing of TCP Non-open States when ACKed Beyond tp -> high_seq 467
  12.6.1 TCP_CA_Loss 467
  12.6.2 TCP_CA_CWR 468
  12.6.3 TCP_CA_Disorder 470
  12.6.4 tcp_try_undo_dsack() 471
  12.6.5 TCP_CA_Recovery 471
  12.6.6 tcp_add_reno_sack() 472
  12.6.7 tcp_check_reno_reordering() 473
  12.6.8 tcp_may_undo() 473
  12.6.9 tcp_packet_delayed() 474
  12.6.10 tcp_undo_cwr() 475
  12.6.11 tcp_mark_head_lost() 475
  12.6.12 tcp_sync_left_out() 477
12.7 Summary 477

13 NETLINK SOCKETS

13.1 Introduction to Netlink Sockets 479
13.2 Netlink Socket Registration and Initialization at Boot Time 480
13.3 How Is the Kernel Netlink Socket Created? 481
13.4 How Is the User Netlink Socket Created? 482
13.5 Netlink Data Structures 485
  13.5.1 nl_table 485
  13.5.2 rtnetlink_link 486
13.6 Other Important Data Structures 488
  13.6.1 struct nlmsghdr 488
  13.6.2 struct msghdr 489
13.7 Netlink Packet Format 490
13.8 Netlink Socket Example—tc Command for Adding a qdisc 490
  13.8.1 tc Command Flow in User Space for Adding a qdisc 490
  13.8.2 tc Command in Kernel Space 491
    13.8.2.1 sys_sendmsg() 491
    13.8.2.2 sock_sendmsg() 492
    13.8.2.3 netlink_sendmsg() 492
    13.8.2.4 netlink_unicast() 493
    13.8.2.5 netlink_data_ready() 494
    13.8.2.6 rtnetlink_rcv() 494
    13.8.2.7 rtnetlink_rcv_skb() 494
    13.8.2.8 rtnetlink_rcv_msg() 495
13.9 Flow Diagram for tc Command in Kernel Space 496
13.10 Summary 496

14 IP ROUTING 499
14.1 Routing 501
14.2 Policy-Based Routing 503
14.3 Multipathing 505
14.4 Record Route Options (RFC 791) and Processing by Linux Stack 509
    14.4.1 Record Routing 510
14.5 Source Routing 510
    14.5.1 Strict Record Routing 510
    14.5.2 Loose Record Routing 511
    14.5.3 SRR Processing Implementation 511
14.6 Linux Kernel Implementation of Routing Table and Caches 517
14.7 Routing Cache Implementation Overview 517
    14.7.1 Routing Cache Data Structures 519
14.8 Managing Routing Cache 523
    14.8.1 Routing Cache for Local Connections 525
    14.8.2 __sk_dst_check() 526
    14.8.3 Link Failure and Reporting to Routing Subsystem 527
    14.8.4 dst_link_failure() 527
    14.8.5 ipv4_link_failure() 527
    14.8.6 dst_setExpires() 528
    14.8.7 Routing Cache for the Incoming Packets 529
    14.8.8 Routing Cache Timer 530
    14.8.9 rt_periodic_timer 530
14.8.10  \emph{rt\_may\_expire()}  \\
14.8.11  \emph{dst\_free()}  \\
14.8.12  \emph{\_\_dst\_free()}  \\
14.8.13  \emph{dst\_destroy()}  \\
14.8.14  \emph{dst\_run\_gc()}  \\
14.8.15  Interface down and \emph{rt\_flush\_timer}  \\
14.8.16  \emph{rt\_cache\_flush()}  \\

14.9  Implementation Overview of Forwarding Information Base (FIB)  \\
14.9.1  \emph{struct fib\_table}  \\
14.9.2  \emph{struct fn\_hash}  \\
14.9.3  \emph{struct fn\_zone}  \\
14.9.4  \emph{struct fib\_node}  \\
14.9.5  \emph{struct fib\_info}  \\
14.9.6  \emph{struct fib\_nh}  \\
14.9.7  \emph{struct fib\_rule}  \\

14.10  Adding New Entry in Routing Table Using ip Command (RT Netlink Interface)  \\
14.10.1  What Happens When the ip Command Is Run with a Route Option for Adding an Entry in Routing Table?  \\
14.10.2  \emph{inet\_rtm\_newroute()}  \\
14.10.3  \emph{struct rtmsg}  \\
14.10.4  \emph{struct kern\_rta}  \\
14.10.5  \emph{fn\_hash\_insert()}  \\
14.10.6  \emph{fn\_new\_zone()}  \\
14.10.7  \emph{fib\_create\_info()}  \\
14.10.8  \emph{fn\_hash\_insert()}  \\

14.11  What Happens When the ip Command Is Run with a Rule Option for Adding an Entry in the Routing Table?  \\
14.11.1  \emph{inet\_rtm\_newrule()}  \\
14.11.2  FIB Initialization  \\

14.12  FIB Traversal Flow Diagram  \\
14.12.1  \emph{ip\_route\_output()}  \\
14.12.2  \emph{ip\_route\_output\_key()}  \\
14.12.3  \emph{ip\_route\_output\_slow()}  \\
14.12.4  \emph{ip\_dev\_find()}  \\
14.12.5  \emph{\_\_in\_dev\_get()}  \\
14.12.6  \emph{inet\_select\_addr()}  \\
14.12.7  \emph{ROUTE\_SCOPES}  \\
14.12.8  \emph{fib\_lookup()}  \\

14.13  Summary
15  IP QUALITY OF SERVICE IN LINUX (IP QoS) 591

15.1 Introduction 591
15.2 Basic Components of Linux Traffic Control 592
15.3 Linux Implementation of pfifo_fast qdisc 593
15.4 Queueing Discipline Data Structure 596
  15.4.1 struct Qdisc 596
  15.4.2 struct Qdisc_ops 597
  15.4.3 struct Qdisc_class_ops 598
  15.4.4 struct cbq_class 599
15.5 tc User Program and Kernel Implementation Details 601
  15.5.1 tc_modify_qdisc() 601
  15.5.2 qdisc_create() 602
  15.5.3 cbq_init() 604
  15.5.4 qdisc_raft() 604
  15.5.5 dev_raft_qdisc() 605
15.6 The tc Commands for Creating Class Hierarchy for CBQ 605
  15.6.1 tc_ctl_tclass() 607
  15.6.2 cbq_change_class() 607
15.7 Filters 610
  15.7.1 tc_ctl_tfilter() 611
15.8 u32 Filter Implementation 614
  15.8.1 u32_change() 615
15.9 Route Filter Implementation 616
  15.9.1 route4_change() 618
15.10 Enqueue 619
  15.10.1 cbq_enqueue() 620
  15.10.2 cbq_classify() 621
  15.10.3 Overview of cbq_enqueue() 621
15.11 Overview of Linux Implementation of CBQ 622
15.12 cbq_dequeue() 622
  15.12.1 From net/dev/core.c 626
  15.12.2 qdisc_run() 626
  15.12.3 qdisc_restart() 626
  15.12.4 cbq_dequeue() 627
  15.12.5 cbq_dequeue_1() 629
  15.12.6 cbq_dequeue_prio() 630
15.13 Summary 633
16 IP FILTER AND FIREWALL

16.1 Netfilter Hook Framework 636
16.2 Netfilter Hooks on IP Stack 638
  16.2.1 Hooks for Outgoing Packets 638
  16.2.2 Hooks for Incoming Packets 639
16.3 Overview of Netfilter Hooks on Linux TCP-IP Stack 640
16.4 Registration of Netfilter Hooks 640
16.5 Processing of Netfilter Hooks 642
  16.5.1 nf_hook_slow() 642
  16.5.2 nf_iterate() 643
  16.5.3 struct nf_hook_ops 644
16.6 Compatibility Framework 644
  16.6.1 fw_in() 645
16.7 Ip Chains 647
  16.7.1 Filtering with Ipchains 648
  16.7.2 Ipchain Chain of Rules 649
  16.7.3 struct ip_chain 649
  16.7.4 struct ip_fwkernel 650
  16.7.5 struct ip_reent 651
  16.7.6 struct ip_fw 651
  16.7.7 Organization of Tables in Ipchains 652
16.8 How Is the Packet Filtered with Ipchains 653
  16.8.1 ip_fw_check() 653
  16.8.2 ip_rule_match() 655
16.9 Iptables 655
  16.9.1 Registration of Iptables Hooks 657
16.10 Iptables Filter Rules and Target Organization 657
  16.10.1 struct ipt_table 658
  16.10.2 struct ipt_table_info 658
  16.10.3 struct ipt_entry 661
  16.10.4 struct ipt_entry_match 662
  16.10.5 struct ipt_tcp 663
  16.10.6 struct ipt_entry_target 664
  16.10.7 struct ipt_standard_target 664
16.11 Organization of Filter Rules and Target for Iptables 664
16.12 Filtering Packets with Iptables 664
  16.12.1 ipt_do_table() 664
  16.12.2 IPT_MATCH_ITERATE 668
16.13 Summary 668
17 NET SoftIRQ

17.1 Why Net SoftIRQs, and How Do We Raise Them? 672
17.1.1 Transmission 672
17.1.2 Reception 672

17.2 How Are SoftIRQs Are Processed, and When? 675
17.3 Registration of SoftIRQs 678
17.4 Packet Reception and Delayed Processing by Rx SoftIRQ 679
17.5 Processing of Net Rx SoftIRQ 682
17.6 Packet Transmission and SoftIRQ 686
17.7 Summary 696

18 TRANSMISSION AND RECEPTION OF PACKETS 697

18.1 DMA Ring Buffers for Transmission and Reception of Packets 698
18.2 Packet Reception Process 698
18.2.1 Flow of Packet Reception with DMA 698
18.2.2 Reception Ring Buffer 698
18.3 Packet Transmission Process 700
18.3.1 Flow of Packet Transmission with DMA 702
18.3.2 Transmission Ring Buffer 702
18.3.3 Transmission Ring Buffer 703

18.4 Implementation of Reception and Transmission of Packets 704
18.4.1 \texttt{struct etrax_eth_descr} 705
18.4.2 \texttt{struct etrax_dma_descr} 706
18.4.3 Initialization of Device 707
18.4.5 Initialization of DMA Transmit Ring Buffers 707
18.4.6 Initialization of DMA Receive Ring Buffers 709

18.5 Rx Interrupt for Reception of Packets 709
18.5.1 Rx DMA Buffer Initialized 711
18.5.2 \texttt{e100_rx()} 711
18.5.3 Rx Descriptors After Reception of Three Packets in DMA Buffer Before Rx Interrupt Being Raised 713
18.5.4 Rx Descriptors After First Packet Is Pulled Out of DMA Buffer and Given to OS in Rx Interrupt Handler 713

18.6 Transmission of Packets 713
18.6.1 \texttt{e100_send_packet()} 713
18.6.2 Tx DMA Ring Buffer Descriptor After Initialization 717
18.6.3 \texttt{e100_hardware_send_packet()} 717
18.6.4 There Are Two Packets in Device’s DMA Tx Ring Buffer to Be Transmitted 717
18.6.5 \texttt{e100tx_interrupt()} 720
18.6.6 First Packet from the DMA Queue Is Transmitted and Second One Is yet to Be Transmitted; After Interrupt Is Generated, Transmitted Buffer Is Freed 721

18.7 Summary 721

19 *lkcd* AND DEBUGGING TCP/IP STACK 723

19.1 *lkcd* Source and Patches 724
19.2 Touching the Socket 724
19.3 Looking into the Receive Socket Buffer 726
   19.3.1 Route Information in *sk_buff* 727
19.4 Peep into Send Socket Buffer 727
19.5 TCP Segmentation Unit 729
19.6 Send Congestion Window and *ssthresh* 730
19.7 Retransmissions and Route 733
19.8 Peeping into Connection Queues and SYN Queues 733
19.9 Routing and IP Qos lcrash Steps 735
   19.9.1 lcrash Steps for Default Queueing Discipline in Linux *(pfifo_fast)* 735
19.10 CBQ (Class-Based) Queueing Discipline lcrash Steps 739
19.11 U32 Filters 739
19.12 Route Filters 743
19.13 FIB Table lcrash Output for Setting Up the Realm Using *ip* Command 745
19.14 lcrash Output for Setting Up Route Filter Using *tc* Command 749
19.15 Netlink Data Structure 755
   19.15.1 *nl_table* 755
   19.15.2 *rtnetlink_link* 755
19.16 Summary 757

20 NEXT EDITION 759

Bibliography 763

Index 765
For more than a decade, Linux has been the most popular choice for server technology, embedded systems, or research work in the networking domain. It slowly gained momentum beginning with the student community and slowly reaching researchers and the corporate world. Networking, when combined with Linux, gives birth to an innovative product line, be it in the high-end telecom sector, data centers, or embedded systems, and so on.

In 1996, I was introduced to Linux while doing my first assignment on TCP/IP socket programming. At that time, I had a very little knowledge about a server program using a unique port number to register itself with the system or a client program using the same port number to communicate with the server. I also had little knowledge of an IP address that is fed to the client program to identify the host. I then set myself to learn about how all that was made possible.

Much information needed to be explored at that time, such as system calls, protocols, Linux kernel, drivers, and kernel framework that supports the stack, and so on. Slowly, I explored the Linux kernel and user-land program interaction with that kernel by writing new system calls and kernel modules.

This learning process began with the *TCP/IP Illustrated, Volume 1* by the honorable Richard Stevens. But it continued to be really difficult to map the protocol with the implementation on Linux because there was so little documentation, and available books provided hardly any information. So, I decided to dive deep into the jungle of the huge source base to find out how the stack is implemented. Finally, I got hooked to the socket and VFS layer to understand how socket layer is linked to the VFS layer. Then slowly I was pointed to the TCP layer and the first routine that interfaces TCP protocol to send out data. Then the journey of documenting and experimenting with the TCP/IP stack began. When the documentation had grown big enough, the idea of making it available to the Linux community emerged. But writing a book was beyond my strength and it was too much work, requiring a lot of time and dedication. But I was determined to expose the complex topic to the Linux community to whatever extent I could even if it demanded many requirements. The absence of detailed, leveled documentation or a book that would have made the subject easier to understand, forced me to think about the topic. The idea of writing a book was supported when I received acceptance on the subject from IEEE Computer Society Press and John Wiley & Sons.

Working on the book along with office work became difficult so I searched for a co-author who would help cover some of the topics. After a long struggle, I convinced M. Ajaykumar Venkatesulu to be my co-author and work on a giant and most complex routing subsystem and QOS.
This text tries to cover almost all the aspects of TCP/IP stack and supporting kernel framework. The idea is to present the topic in a way that dilutes its complexity so that it can be easily understood. To understand TCP/IP implementation on any OS, we need to understand the kernel frameworks that support the stack. On Linux, these frameworks include VFS layer, socket framework, protocol layer, timers, memory management, interrupt handling, softIRQ, kernel threads, kernel synchronization mechanism, and so on. This is the kernel perspective of the stack. Apart from this, we also need to know the basics of the communication protocol and application interfaces (system calls) to open TCP communication sockets and the client–server program. This knowledge is helpful as a reference for experienced professionals and for students willing to learn the complex subject and contribute to the Linux community.

This book is written for the Linux kernel 2.4.20. The newest kernel version 2.6 does not have much variation as far as the TCP/IP stack is considered. Kernel version 2.4 is the most widely accepted kernel in the Linux world. Version 2.6 specific changes will be discussed in subsequent revisions of the book.

AUDIENCE

The book is targeted for large cross section of audience:

Researchers at Worldwide Premier Institutes. Researchers who work on various aspects of the TCP/IP stack find BSD the most suitable networking OS. But BSD is not a popular choice in the corporate world. So, the next most popular choice of researchers is the Linux OS and improvement of the TCP/IP stack performance on this OS. Networking is currently the most popular field for research because of growing usage and popularity of the Internet. Mostly, researchers prefer an OS with commercial viability that can run on cheap hardware.

Academia. Advanced academic degree projects, such as MS, M. Tech., B. Tech. and PG, are mostly done on Linux because it was the first UNIX-like OS available with fairly good documentation and stability. In the networking area, students usually choose Linux over TCP/IP for their project work. The project may require modifying the router or TCP performance, implementing some new TCP/IP RFC, network drivers, implementing secured IP layer, or improving scalability factor to handle network traffic.

Corporations. For the most part, the corporate world has widely accepted Linux as the base OS for networking products. Many companies are developing network products, such as IP security, QOS (class-based routing), developing routers, bandwidth management products, cluster servers and many more, which require modifying the TCP/IP stack or writing a new module altogether that fits into Linux TCP/IP stack somewhere. Linux is not only popular as an open system but is also a major choice for embedded products or real-time OS. These embedded products are mostly developed for networking domains such as routers, embedded web servers, web browsers, and so on.

Entrepreneurs. New ideas keep popping up which need to be turned into products. With the Internet gaining popularity, many ideas have been born to develop networking products. Linux is once again the most popular choice for development among entrepreneurs.
The Open Source Community. Because of the growing popularity of Linux and Internet technologies, many fresh college graduates or even software professionals want to contribute to Linux networking capabilities. Their goal is to make Linux more powerful, stable, secure, and full of network capabilities in order to meet corporate requirements in every possible way. Many professionals want to contribute to Linux networking capabilities but don’t find enough time to get acquainted with its networking stack and the kernel framework.

Defense Organizations. There is a growing popularity of Linux as network OS in defense organizations with increasing military adoption of Linux IP security with some modifications for secured military network transactions.

All these audiences require a thorough knowledge of Linux TCP/IP stack and kernel framework for networking stacks. To understand TCP, IP, BSD sockets, firewall, IP security, IP forwarding, router network driver, complete knowledge of how networking stack implementation and design work is needed. If IP security or firewall implementation is wanted, then knowledge of how the packet is implemented in Linux, how and where packet is passed to the IP layer, how the IP processes the packets and adds headers, and finally how the IP passes the packet to the device driver for final transmission is needed. Similarly, implementation of the QOS or some modifications in the existing implementation is needed, knowledge of Linux routing table implementation, packet structure, packet scheduling and all related kernel framework including network soft IRQs is required. So, anything and everything that requires modifying the Linux network stack or adding a new feature to the stack, requires complete knowledge of the design and implementation of Linux TCP/IP stack.

ORGANIZATION OF THIS BOOK

This book completely explains TCP/IP protocol, its design, and implementation in Linux. Basically, the book begins with simple client–server socket programs and ends with complex design and implementation of TCP/IP protocol in Linux. In between, we gradually explain the different aspects of socket programming and major TCP/IP-related algorithms. These are:

Linux Kernel and TCP/IP Application Interfaces: Chapter 1 covers the Linux kernel basics and we kick start with kernel interfaces (system calls) to use TCP/IP protocol stack for communication.

Protocols: Chapter 2 covers TCP/IP protocols and supporting protocols such as ARP and ICMP. We cover some of the major RFCs with illustrations to acquaint the reader with the protocols so that it will be easy to map Linux implementation on Linux in further chapters.

Sockets: Chapter 3 explains the implementation of BSD socket implementation in the Linux kernel. Here we discuss in detail how socket layer is hooked to VFS layer and how various protocols are hooked to BSD socket.

Kernel Implementation of Connection Setup: Chapter 4 explains the client–server application with the help of the C program. We explain the complete process of connection setup with the help of tcpdump output in different chapters. We cover kernel implementation of system calls used by application program to implement client–server interaction. We see how connections are accepted on the server side
and at the same time, learn how the server program registers with the kernel to bind to a specific listening port.

**Linux Implementation of Network Packet:** Chapter 5 explains sk_buff which represents network packet on Linux. We explain important routines that manipulate sk_buff.

**Movement of Packet Across the Layers:** Chapter 6 covers the complete TCP/IP stack framework, showing how the packet is generated and trickles down the network stack until it is out of the system. Similarly we explain the complete path taken by a packet received from the device to reach the owning socket, covering complete kernel framework that implements TCP/IP stack on Linux.

**TCP recv/send:** Chapters 7 and 8 address TCP receive/send implementation and cover all the aspects related to TCP receiving and sending data. We also explain the TCP segmentation unit when an ICMP error (mss change for the route) is received by the TCP. There is a small description of how urgent data are processed.

**TCP Socket Timers and Memory Management:** The kernel keeps track of memory consumed by a connection at the socket layer so that a single-socket connection is not able to hog all the system memory because of a misbehaving application. We also try to collapse sequential buffers in the receive queue when the application is not reading enough fast and socket has exhausted its quota. This aspect of memory management is covered in Chapter 9. TCP is an event-driven protocol. TCP implements timers to track loss of data, to send delayed ACKs, to send out zero window probes, and so on. Chapter 10 addresses all these aspects.

**TCP State Machine:** Chapter 11 covers TCP core processing, such as reception of packets, sending ACKs, sliding window protocol, Nagle’s algorithms, scheduling of delayed ACK’s, processing of out-of-order segments, processing SACK, D-SACK, and so on. The tcp_opt object represents state machine implementation on Linux. Chapter 12 covers TCP congestion control algorithms implementation.

**Netlink Sockets:** User–land applications, such as netstat and iproute, and routing protocol daemons use special netlink sockets to update/read routes and configure QOS in the kernel. We cover netlink sockets in Chapter 13.

**IP Layer and Routing Table Implementation:** Chapter 14 covers implementation of routing table (FIB) on Linux. We also explain different aspects associated with routing, such as multipathing, policy routing, and so on. This chapter also explains the different kernel control paths that update kernel routing tables and route cache management.

**IP QOS:** IP in today’s network is an advanced topic and is used for different services in the public network. Linux implements QOS very cleanly and we discuss PFIFO and CBQ queuing discipline implementation in Chapter 15.

**Netfilter Framework:** Linux provides extensions to the TCP/IP stack by way of the netfilter framework. These extensions can be firewall, masquerading, IP security, and so on. Chapter 16 covers netfilter hooks at different layers in the stack and also netfilter implementation.

**SoftIRQ Implementation for Scalability:** Network frames are received in the kernel memory in the interrupt handler code but complete processing of the packets can’t be done in the interrupt handler. Linux associates softIRQ, one each for reception and transmission of packets for processing of packets. Chapter 17 explains net softIRQ framework with the help of illustrations. This chapter completely explains the high scalability of Linux on SMP architecture in handling network traffic.
**Link Layer and DMA Ring Buffers:** Chapter 18 covers link layer (device driver) processing of packets. Design and working of DMA ring buffer for reception and transmission are also addressed and are explained with the help of a device driver and interrupt routines for a real device.

**Debug TCP/IP Stack:** Debugging the TCP/IP stack is discussed in Chapter 19. The lkcd (linux kernel crash dump) debugger is used to illustrate the debugging technique, peeking into different kernel data-structures associated with TCP/IP stack.

**LEVEL OF DISCRIPTION**

As outlined here, we have touched upon critical portions of the implementation that are required to understand core TCP/IP stack and kernel framework. Each chapter begins with a chapter outline and ends with a summary that highlights important points. Source-level explanations with diagrams are provided where ever required. Important routines are explained line-by-line. Code snippets are provided for all those routines with line numbers and files of code snippet. Sometimes routines are so big that they are split into different code snippets. Routines that are called from the main routines are explained in different sections. If the called routine is a couple of lines long, there is no separate section for those routines. Line number and code-snippet number (cs-) are provided with the explanation to assist understanding. When the routines are very big in size, notification is provided at the beginning of the section stating, see cs  ••. ••, unless mentioned; this means that where ever line numbers are mentioned, we need to see the code snippet mentioned at the start of the section.

In the explanation if we encounter some concept that is already explained in some other section, a cross reference to that section is provided, as see Section  ••. ••. Cross references are provided because the subject is interrelated, for example while explaining queuing of incoming TCP packet, we refer to sockets receive buffer. If we have exhausted the receive socket buffer, we need to call routines to collapse receive queue to make space for the new TCP data segment. For this we may need to refer to a section from the TCP memory management chapter. We have explained major data structures with significance separately. Where ever that has not been done, fields of those data-structures are explained as and when they appear in the routines.

Examples and illustrations are provided where ever it is required to make subject easier to understand. For example, diagrams to link various kernel data structures are drawn to illustrate connection requests in the SYN queue. Then we illustrate shifting of connection requests from SYN queue to accept queue when a three-way handshake is over with the help of diagrams. All these illustrations assist in visualizing the complex data structures and scenarios.

*Sameer Seth*

*Bangalore, India*

*September 2008*
For me, this is the heaviest section of the book that carries the most weight. First of all, I’m very thankful to my family for being so supportive and patient when I was working on the title, with little time left for them. My wife, Sumam, provided selfless support to the work right from day one. She provided me with confidence to convert my hard work into a book on the day she provided me with the list of publishers. When submitting my book proposal, only 20% of the work was done and that too was not organized.

I thank my co-author, M. Ajaykumar Venkatesulu, who agreed to join hands with me at the much-needed hour. His commitment eased the load on my shoulders and he worked very hard with all dedication to make this possible. He had a really tough time setting up QOS on Linux, with a couple of Linux boxes, and modifying the kernel for his illustrations.

I’d like to thank the very first person at the IEEE Computer Society with whom I interfaced, Deborah Plummer, who worked on the proposal until it was finished. She helped me in many ways to understand the publication process and was very patient all through, clarifying my doubts. IEEE Staffers, Janet Wilson and Dante David, were so nice and prompt throughout the review process. Even a small communication gap caused serious concerns because this was the first time I was working on such a big project. But Janet and Dante were patient and always prompt in their replies to make sure that all my concerns were addressed. I was introduced to Lisa Van Horn from Wiley much later, when the book had entered the production phase. It is a great experience working with her because she spent time educating me at every point. At times I would be very irritating to her by asking silly doubts but she tackled them all with grace. She has worked very hard editing the book because there were grammatical corrections in almost every line. Through the production process, she was very helpful, cooperative, and prompt in the same way.

There are a few names without which this book would look incomplete. I thank Richard McDougall, the respectable author of Solaris Internals, for time spent educating me on the publication process. His inputs helped me achieve the most from my hard work. The respectable senior engineer from SGI and owner of the dwarf extract utility for lkcd, Cliff Wickman, is owed thanks for without him this book would have looked quite dry. He provided a tool to generate a kernel-type database (kerntypes) because the basic lkcd utility does not come with all the stubs for kernel data-structures in kerntypes. Without this tool, the debug chapter would not have been possible. He not only provided the tool but also helped get the kernel-type database built for the kernel 2.4 when the tool was compatible only with kernel 2.6.

S. S.
Writing or co-authoring a book was never even in my wildest dreams. The opportunity came by chance and then it became my choice. God has been kind enough to give me such an amazing opportunity. I have a couple of people to thank with whom my words fall short. First of all I would like to thank the author of the book who had faith in me that I could write on this subject. He gave me a lot of trust when he gave me an opportunity to work on this book. It was solely his brainchild which he shared with me selflessly. He gave me guidance whenever I faced any difficulty in any subject matter. His valuable suggestions and most importantly his inspirations have made it possible for me to finish this assignment.

I thank my family for all their support: My father who stood beside me through all the odds and evens of life so that I could concentrate on this project; my newly wedded wife, Priyanka, who never complained when I had less or sometimes no time left for her; and lastly, my brother-in-law Balaji who has been a great source of inspiration in my life.

Last but not least, I thank Deborah Plummer, Janet Wilson, and Dante David from IEEE for being so cooperative and nice.

The book is not a result of any inspiration but the need of the day. When you have the strong desire to achieve something, then the whole of creation conspires to accomplish your goal.

M. A. V.