
TCP/IP ARCHITECTURE, DESIGN, AND IMPLEMENTATION IN LINUX

Sameer Seth
M. Ajaykumar Venkatesulu

IEEE
 computer
society

 **WILEY**

A JOHN WILEY & SONS, INC., PUBLICATION

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/ieeecss

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>.

TCP/IP ARCHITECTURE, DESIGN, AND IMPLEMENTATION IN LINUX

Sameer Seth
M. Ajaykumar Venkatesulu

IEEE
 computer
society

 **WILEY**

A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2008 by IEEE Computer Society.
Published by John Wiley & Sons, Inc., Hoboken, New Jersey. All rights reserved.

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permission>.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print, may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data is available.

ISBN 978-0470-14773-3

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

CONTENTS

Preface	xxi
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 <i>fork()</i>	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 <i>read()</i>	38
1.9.2 <i>write()</i>	38

1.9.3	<i>recv()</i>	38
1.9.4	<i>send()</i>	39
1.9.5	<i>select()</i>	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	<i>netstat</i>	90
2.14	<i>traceroute</i>	92
2.14.1	<i>traceroute</i> Mechanism	93
2.15	ICMP	93
2.16	<i>ping</i>	95
2.17	ARP/RARP	97
2.18	Summary	99

3	KERNEL IMPLEMENTATION OF SOCKETS	101
3.1	Socket Layer	102
3.2	VFS and Socket	103
3.3	Protocol Socket Registration	105
3.4	<i>struct inet_protosw</i>	107
3.5	Socket Organization in the Kernel	107
3.6	Socket	108
3.7	<i>inet_create</i>	110
3.7.1	Sock	112
3.8	Flow Diagram for Socket Call	118
3.9	Summary	118
4	KERNEL IMPLEMENTATION OF TCP CONNECTION SETUP	121
4.1	Connection Setup	122
4.1.1	Server Side Setup	122
4.1.2	Server Side Operations	124
4.2	Bind	124
4.2.1	Data Structures Related to Socket BIND	125
4.2.2	Hash Buckets for tcp Bind	125
4.2.3	<i>tcp_ehash</i>	125
4.2.4	<i>tcp_listening_hash</i>	125
4.2.5	<i>tcp_bhash</i>	125
4.2.6	<i>tcp_hashinfo</i>	126
4.2.7	<i>tcp_bind_hashbucket</i>	129
4.2.8	<i>tcp_bind_bucket</i>	129
4.2.9	<i>bind()</i>	130
4.2.10	<i>sys_bind()</i>	130
4.2.11	<i>sockfd_lookup()</i>	130
4.2.12	<i>fget()</i>	131
4.2.13	<i>inet_bind()</i>	131
4.2.14	<i>tcp_v4_get_port()</i>	133
4.2.15	<i>tcp_bind_conflict()</i>	135
4.3	Listen	137
4.3.1	<i>sys_listen()</i>	138
4.3.2	<i>inet_listen()</i>	139
4.3.3	<i>tcp_listen_start()</i>	139
4.3.4	Listen Flow	142
4.3.5	<i>struct open_request</i>	142
4.3.6	Accept Queue Is Full	147
4.3.7	Established Sockets Linked in <i>tcp_ehash</i> Hash Table	150

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	<i>inet_accept()</i>	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	<i>tcp_v4_connect()</i>	167
4.6.4	<i>ip_route_connect()</i>	167
4.6.5	Flow Control for Generating a Connection Request	167
4.6.6	<i>tcp_v4_hash_connect()</i>	170
4.6.7	<i>__tcp_v4_check_established()</i>	171
4.6.8	<i>tcp_connect()</i>	174
4.6.9	<i>tcp_transmit_skb()</i>	176
4.7	Summary	178
5	<i>sk_buff</i> AND PROTOCOL HEADERS	181
5.1	struct <i>sk_buff</i>	182
5.2	struct <i>skb_shared_info</i>	186
5.3	<i>sk_buff</i> and DMA—SKB_FRAG_STRUCT	187
5.3.1	DMA and Fragmented <i>sk_buff</i> Containing Paged Data	188
5.3.2	<i>sk_buff</i> and IP Fragmentation	188
5.3.3	<i>sk_buff</i> and Fragmentation	190
5.4	Routines Operating on <i>sk_buff</i>	190
5.4.1	<i>alloc_skb()</i>	190
5.4.2	<i>skb_reserve()</i>	191
5.4.3	<i>skb_put()</i>	192
5.4.4	<i>skb_push()</i>	194
5.4.5	<i>skb_pull()</i>	195

5.5	<i>sk_buff</i> Builds Protocol Headers as It Traverses Down the Protocol Layers	196
5.5.1	Tcp Header Is Added to <i>sk_buff</i>	196
5.5.2	Ip Header Is Added to <i>sk_buff</i>	197
5.5.3	Link Layer Header Is Added to <i>sk_buff</i>	198
5.6	<i>sk_buff</i> Extracts Protocol Headers as It Traverses Up the Protocol Layers When a Packet Arrives	199
5.6.1	<i>sk_buff</i> Is Made to Point to a Datalink Layer Header Which Will Be Processed by a Dalalink Driver	199
5.6.2	<i>sk_buff</i> Is Made to Point to an ip Layer Header Which Will Be Processed by an IP Layer	200
5.6.3	<i>sk_buff</i> Is Made to Point to a tcp Layer Header Which Will Be Processed by a tcp Layer	200
5.7	Summary	202
6	MOVEMENT OF <i>sk_buff</i> ACROSS PROTOCOL LAYERS	205
6.1	Packet Traversing Down the TCP/IP Stack	206
6.1.1	Path of Packet Traversal from Socket Layer to Device for Transmission	207
6.1.2	Kernel Path for TCP Packet Traversing Down the Stack	208
6.2	Routed Packet Ready for Transmission	214
6.3	Kernel Flow for a Packet Moving Down the Stack	214
6.4	Packet Traversing Up the TCP/IP Stack	214
6.4.1	Path of Packet Traversal from Device (Reception) to Socket Layer	219
6.4.2	Kernel Path for TCP Packet Traversing Up the Stack	219
6.5	Kernel Flow for a Packet Moving Up the Stack	225
6.6	Summary	225
7	TCP SEND	231
7.1	TCP Segmentation Unit for Sending Data	232
7.1.1	Functioning of Segmentation Unit without Scatter–Gather Support	232
7.1.2	Segmentation without Scatter–Gather Support	234
7.1.3	1 mss of Data Written over the Socket	235
7.2	Segmentation with Scatter–Gather Technique	235
7.2.1	Segmentation with Scatter–Gather Support	239
7.2.2	Application Writes Y Bytes over the Socket	239
7.2.3	<i>can_coalesce()</i>	239
7.2.4	<i>tcp_copy_to_page()</i>	240
7.2.5	<i>tcp_mark_push()</i>	241
7.2.6	<i>forced_push()</i>	241

7.2.7	<i>tcp_push()</i>	242
7.2.8	<i>__tcp_push_pending_frames()</i>	243
7.2.9	<i>tcp_snd_test()</i>	243
7.2.10	<i>tcp_nagle_check()</i>	244
7.2.11	<i>tcp_minshall_ckeck()</i>	245
7.2.12	<i>tcp_write_xmit()</i>	245
7.2.13	<i>update_send_head()</i>	247
7.2.14	<i>tcp_push_one()</i>	247
7.2.15	<i>skb_entail()</i>	248
7.3	Sending OOB Data	249
7.4	Flow for TCP Segmentation Unit and Send Process	250
7.5	Functional Level Flow for Segmentation and Send Mechanism	250
7.6	Summary	251

8	TCP RECEIVE	255
8.1	Queuing Mechanism	256
8.1.1	Processing in <i>tcp_rcv_established()</i>	256
8.1.2	<i>tcp_prequeue()</i>	258
8.1.3	Processing of Queues	259
8.1.4	<i>tcp_data_wait()</i>	263
8.1.5	<i>tcp_prequeue_process()</i>	264
8.1.6	<i>lock_sock()</i>	265
8.1.7	<i>__lock_sock()</i>	265
8.1.8	<i>release_sock()</i>	266
8.1.9	<i>__release_sock()</i>	266
8.2	Processing of TCP Data from the Receive Queue	267
8.2.1	<i>cleanup_rbuf()</i>	268
8.2.2	<i>skb_copy_datagram_iovec()</i>	271
8.2.3	Reading Data from Receive Buffer without Paged Data Area	273
8.2.4	X Bytes Requested from the Application	273
8.2.5	1 mss = n Bytes Requested from the Application	275
8.2.6	n – X Bytes Requested from the Application	275
8.2.7	Consumption of Data from a Paged Buffer	275
8.2.8	n Bytes Requested by the Application	276
8.2.9	One Page of Data Requested by the Application	276
8.3	TCP Urgent Byte Processing	276
8.3.1	Urgent Byte Read as OOB Data	277
8.3.2	<i>tcp_recv_urg()</i>	278
8.3.3	Urgent Mode Processing and Reading an Urgent Byte as Inline Data	280

8.4	DATA Flow Diagram for Receiving Data over the TCP Socket	284
8.5	Summary	290

9 TCP MEMORY MANAGEMENT 291

9.1	Transmit Side TCP Memory Management	291
9.1.1	<i>select_size()</i>	294
9.1.2	<i>tcp_alloc_pskb()</i>	295
9.1.3	<i>alloc_skb()</i>	296
9.1.4	<i>tcp_alloc_page()</i>	297
9.1.5	<i>skb_charge()</i>	298
9.1.6	<i>tcp_mem_schedule()</i>	298
9.1.7	<i>tcp_free_skb()</i>	300
9.1.8	<i>sock_wfree()</i>	300
9.1.9	<i>tcp_write_space()</i>	301
9.1.10	<i>tcp_mem_reclaim()</i>	302
9.1.11	<i>__tcp_mem_reclaim()</i>	302
9.1.12	<i>wait_for_tcp_memory()</i>	303
9.2	Receive Side TCP Memory Management	305
9.2.1	<i>tcp_prune_queue()</i>	308
9.2.2	<i>tcp_clamp_window()</i>	309
9.2.3	<i>tcp_collapse_ofo_queue()</i>	311
9.2.4	<i>tcp_collapse()</i>	312
9.2.5	<i>__skb_queue_purge()</i>	317
9.3	Freeing of Memory Allocated to a Receive Buffer	319
9.4	System-Wide Control Parameters Are Worth Noticing When It Comes to TCP Memory Management	319
9.5	Summary	321

10 TCP TIMERS 323

10.1	Timers in Linux	324
10.1.1	<i>mod_timer()</i>	324
10.1.2	<i>detach_timer()</i>	325
10.1.3	<i>del_timer()</i>	325
10.1.4	When Are Timer Routines Executed?	326
10.2	TCP Retransmit Timer	326
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	<i>tcp_enter_loss()</i>	330
10.2.4	<i>tcp_retransmit_skb()</i>	333
10.2.5	<i>tcp_retrans_try_collapse()</i>	334
10.2.6	<i>skb_cloned()</i>	336

10.3	Zero Window Probe Timer	336
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	<i>tcp_ack_probe()</i>	338
10.3.4	How Does the Window Probe Timer Work?	338
10.3.5	<i>tcp_probe_timer()</i>	339
10.3.6	<i>tcp_send_probe0()</i>	339
10.3.7	<i>tcp_write_wakeup()</i>	339
10.4	Delay ACK Timer	342
10.4.1	When Is the ACK Scheduled?	344
10.4.2	How and When Is the ACK Segment Sent?	344
10.4.3	Quick ACK Mode	345
10.4.4	<i>__tcp_ack_snd_check()</i>	345
10.4.5	<i>tcp_ack_snd_check()</i>	346
10.4.6	<i>tcp_send_delayed_ack()</i>	347
10.4.7	<i>tcp_delack_timer()</i>	348
10.4.8	<i>tcp_reset_xmit_timer()</i>	349
10.4.9	<i>tcp_write_timer()</i>	351
10.4.10	<i>tcp_clear_xmit_timer()</i>	352
10.5	Keepalive Timer	353
10.5.1	When Is the Keepalive Timer Activated?	353
10.5.2	How Is the Timer Reset?	354
10.5.3	<i>tcp_keepalive_timer()</i>	354
10.6	SYN-ACK Timer	356
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	<i>tcp_synack_timer()</i>	357
10.7	TIME_WAIT Timer	361
10.7.1	When Do We Trigger <i>TIME_WAIT</i> Timer?	361
10.7.2	<i>tcp_time_wait()</i>	362
10.7.3	<i>tcp_tw_schedule()</i>	362
10.7.4	Non-recycle Mode	363
10.7.5	Recycle Mode	365
10.7.6	<i>tcp_twkill()</i>	367
10.7.7	<i>tcp_twcal_tick()</i>	370
10.7.8	<i>__tcp_tw_hashdance()</i>	374
10.8	Summary	375

11 TCP CORE PROCESSING 377

11.1	TCP Incoming Segment Processing	378
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	<i>tcp_sequence()</i>	387
11.3.2	<i>tcp_replace_ts_recent()</i>	387
11.3.3	<i>tcp_event_data_rcv()</i>	390
11.3.4	<i>tcp_incr_quickack()</i>	391
11.3.5	<i>tcp_grow_window()</i>	392
11.3.6	<i>__tcp_grow_window()</i>	393
11.3.7	How Do We Calculate Window to Be Advertised?	394
11.3.8	<i>tcp_receive_window()</i>	395
11.3.9	<i>__tcp_select_window()</i>	395
11.3.10	<i>tcp_space()</i>	397
11.3.11	<i>tcp_data_snd_check()</i>	397
11.3.12	<i>__tcp_data_snd_check()</i>	398
11.3.13	<i>tcp_paws_discard()</i>	398
11.4	Processing of Incoming ACK	400
11.4.1	<i>tcp_packets_in_flight()</i>	403
11.4.2	<i>tcp_ack_is_dubious()</i>	404
11.4.3	<i>tcp_cong_avoid()</i>	405
11.4.4	<i>tcp_ack_update_window()</i>	406
11.4.5	<i>tcp_may_update_window()</i>	407
11.4.6	<i>tcp_clean_rtx_queue()</i>	408
11.5	Processing of SACK blocks	410
11.5.1	<i>tcp_sacktag_write_queue()</i>	410
11.6	Reordering Length	417
11.7	Processing TCP Urgent Pointer	421
11.7.1	<i>tcp_check_urg()</i>	422
11.8	Processing Data Segments in Slow Path	424
11.8.1	<i>tcp_sack_new_ofo_skb()</i>	433
11.8.2	<i>tcp_sack_maybe_coalesce()</i>	434
11.8.3	<i>tcp_sack_extend()</i>	435
11.8.4	<i>tcp_ofo_queue()</i>	436
11.8.5	<i>tcp_sack_remove()</i>	441
11.9	Overview of Core TCP Processing	442
11.10	Summary	442

12	TCP STATE PROCESSING	445
12.1	Overview of State Processing	446
12.2	TCP States	448
12.2.1	<i>TCP_CA_CWR</i>	449
12.2.2	Undoing from <i>TCP_CA_CWR</i>	449
12.3	Processing of Duplicate/Partial ACKs in Recovery State	449
12.3.1	<i>tcp_remove_reno_sacks()</i>	450
12.3.2	<i>tcp_try_undo_partial()</i>	451
12.4	Processing of Duplicate/Partial ACKs in Loss State	452
12.4.1	<i>tcp_try_undo_loss()</i>	453
12.4.2	<i>tcp_check_sack_reneging()</i>	455
12.5	Default Processing of TCP States	456
12.5.1	<i>tcp_time_to_recover()</i>	459
12.5.2	<i>tcp_head_timedout()</i>	460
12.5.3	<i>tcp_try_to_open()</i>	461
12.5.4	<i>tcp_update_scoreboard()</i>	462
12.5.5	<i>tcp_xmit_retransmit_queue()</i>	464
12.5.6	<i>tcp_packet_delayed()</i>	466
12.6	Processing of TCP Non-open States when ACKed Beyond <i>tp</i> \rightarrow <i>high_seq</i>	467
12.6.1	<i>TCP_CA_Loss</i>	467
12.6.2	<i>TCP_CA_CWR</i>	468
12.6.3	<i>TCP_CA_Disorder</i>	470
12.6.4	<i>tcp_try_undo_dsack()</i>	471
12.6.5	<i>TCP_CA_Recovery</i>	471
12.6.6	<i>tcp_add_reno_sack()</i>	472
12.6.7	<i>tcp_check_reno_reordering()</i>	473
12.6.8	<i>tcp_may_undo()</i>	473
12.6.9	<i>tcp_packet_delayed()</i>	474
12.6.10	<i>tcp_undo_cwr()</i>	475
12.6.11	<i>tcp_mark_head_lost()</i>	475
12.6.12	<i>tcp_sync_left_out()</i>	477
12.7	Summary	477
13	NETLINK SOCKETS	479
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	<i>nl_table</i>	485
13.5.2	<i>rtnetlink_link</i>	486

13.6	Other Important Data Structures	488
13.6.1	<i>struct nlmsg_hdr</i>	488
13.6.2	<i>struct msghdr</i>	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	<i>sys_sendmsg()</i>	491
13.8.2.2	<i>sock_sendmsg()</i>	492
13.8.2.3	<i>netlink_sendmsg()</i>	492
13.8.2.4	<i>netlink_unicast()</i>	493
13.8.2.5	<i>netlink_data_ready()</i>	494
13.8.2.6	<i>rtnetlink_rcv()</i>	494
13.8.2.7	<i>rtnetlink_rcv_skb()</i>	494
13.8.2.8	<i>rtnetlink_rcv_msg()</i>	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	<i>__sk_dst_check()</i>	526
14.8.3	Link Failure and Reporting to Routing Subsystem	527
14.8.4	<i>dst_link_failure()</i>	527
14.8.5	<i>ipv4_link_failure()</i>	527
14.8.6	<i>dst_set_expires()</i>	528
14.8.7	Routing Cache for the Incoming Packets	529
14.8.8	Routing Cache Timer	530
14.8.9	<i>rt_periodic_timer</i>	530

14.8.10	<i>rt_may_expire()</i>	533
14.8.11	<i>dst_free()</i>	534
14.8.12	<i>__dst_free()</i>	535
14.8.13	<i>dst_destroy()</i>	535
14.8.14	<i>dst_run_gc()</i>	536
14.8.15	Interface down and <i>rt_flush_timer</i>	537
14.8.16	<i>rt_cache_flush()</i>	538
14.9	Implementation Overview of Forwarding Information Base (FIB)	540
14.9.1	<i>struct fib_table</i>	540
14.9.2	<i>struct fn_hash</i>	543
14.9.3	<i>struct fn_zone</i>	543
14.9.4	<i>struct fib_node</i>	544
14.9.5	<i>struct fib_info</i>	546
14.9.6	<i>struct fib_nh</i>	547
14.9.7	<i>struct fib_rule</i>	548
14.10	Adding New Entry in Routing Table Using ip Command (RT Netlink Interface)	549
14.10.1	What Happens When the ip Command Is Run with a Route Option for Adding an Entry in Routing Table?	550
14.10.2	<i>inet_rtm_newroute()</i>	550
14.10.3	<i>struct rtmsg</i>	551
14.10.4	<i>struct kern_rta</i>	552
14.10.5	<i>fn_hash_insert()</i>	553
14.10.6	<i>fn_new_zone()</i>	554
14.10.7	<i>fib_create_info()</i>	557
14.10.8	<i>fn_hash_insert()</i>	558
14.11	What Happens When the ip Command Is Run with a Rule Option for Adding an Entry in the Routing Table?	558
14.11.1	<i>inet_rtm_newrule()</i>	559
14.11.2	FIB Initialization	561
14.12	FIB Traversal Flow Diagram	563
14.12.1	<i>ip_route_output()</i>	563
14.12.2	<i>ip_route_output_key()</i>	564
14.12.3	<i>ip_route_output_slow()</i>	566
14.12.4	<i>ip_dev_find()</i>	576
14.12.5	<i>__in_dev_get()</i>	577
14.12.6	<i>inet_select_addr()</i>	578
14.12.7	<i>ROUTE__SCOPES</i>	580
14.12.8	<i>fib_lookup()</i>	581
14.13	Summary	589

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

16	IP FILTER AND FIREWALL	635
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	<i>nf_hook_slow()</i>	642
16.5.2	<i>nf_iterate()</i>	643
16.5.3	<i>struct nf_hook_ops</i>	644
16.6	Compatibility Framework	644
16.6.1	<i>fw_in()</i>	645
16.7	Ip Chains	647
16.7.1	Filtering with Ipchains	648
16.7.2	Ipchain Chain of Rules	649
16.7.3	<i>struct ip_chain</i>	649
16.7.4	<i>struct ip_fwkernel</i>	650
16.7.5	<i>struct ip_reent</i>	651
16.7.6	<i>struct ip_fw</i>	651
16.7.7	Organization of Tables in Ipchains	652
16.8	How Is the Packet Filtered with Ipchains	653
16.8.1	<i>ip_fw_check()</i>	653
16.8.2	<i>ip_rule_match()</i>	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	<i>struct ipt_table</i>	658
16.10.2	<i>struct ipt_table_info</i>	658
16.10.3	<i>struct ipt_entry</i>	661
16.10.4	<i>struct ipt_entry_match</i>	662
16.10.5	<i>struct ipt_tcp</i>	663
16.10.6	<i>struct ipt_entry_target</i>	664
16.10.7	<i>struct ipt_standard_target</i>	664
16.11	Organization of Filter Rules and Target for Iptables	664
16.12	Filtering Packets with Iptables	664
16.12.1	<i>ipt_do_table()</i>	664
16.12.2	<i>IPT_MATCH_ITERATE</i>	668
16.13	Summary	668

17	NET SOFTIRQ	671
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	<i>struct etrax_eth_descr</i>	705
18.4.2	<i>struct etrax_dma_descr</i>	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	<i>e100_rx()</i>	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	<i>e100_send_packet()</i>	713
18.6.2	Tx DMA Ring Buffer Descriptor After Initialization	717
18.6.3	<i>e100_hardware_send_packet()</i>	717
18.6.4	There Are Two Packets in Device's DMA Tx Ring Buffer to Be Transmitted	717
18.6.5	<i>e100tx_interrupt()</i>	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	<i>lkcd</i> AND DEBUGGING TCP/IP STACK	723
19.1	<i>lkcd</i> 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 <i>sk_buff</i>	727
19.4	Peep into Send Socket Buffer	727
19.5	TCP Segmentation Unit	729
19.6	Send Congestion Window and <i>ssthresh</i>	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 (<i>pfifo_fast</i>)	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	<i>nl_table</i>	755
19.15.2	<i>rtnetlink_link</i>	755
19.16	Summary	757
20	NEXT EDITION	759
	Bibliography	763
	Index	765

PREFACE

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 tcp dump 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 DISCRPTION

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

ACKNOWLEDGMENTS

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.