



Contents

1	Introduction And Overview	1
1.1	TCP/IP Protocols	1
1.2	The Need To Understand Details.....	1
1.3	Complexity Of Interactions Among Protocols	2
1.4	The Approach In This Text.....	2
1.5	The Importance Of Studying Code.....	3
1.6	The Xinu Operating System	3
1.7	Organization Of The Remainder Of The Book	4
1.8	Summary	4
1.9	FOR FURTHER STUDY	5
2	The Structure Of TCP/IP Software In An Operating System	6
2.1	Introduction	6
2.2	The Process Concept	7
2.3	Process Priority.....	8
2.4	Communicating Processes.....	8
2.5	Interprocess Communication.....	10
2.5.1	Ports	10
2.5.2	Message Passing.....	11
2.6	Device Drivers, Input, And Output.....	12
2.7	Network Input and Interrupts	13
2.8	Passing Packets To Higher Level Protocols	14
2.9	Passing Datagrams From IP To Transport Protocols	14
2.9.1	Passing Incoming Datagrams to TCP	15
2.9.2	Passing Incoming Datagrams to UDP	15
2.10	Delivery To Application Programs	16
2.11	Information Flow On Output.....	17
2.12	From TCP Through IP To Network Output	18
2.13	UDP Output	19
2.14	Summary	19
2.15	FOR FURTHER STUDY	23
2.16	EXERCISES.....	23
3	Network Interface Layer	24
3.1	Introduction	24
3.2	The Network Interface Abstraction	25
3.2.1	Interface Structure	25



3.2.2	Statistics About Use.....	27
3.3	Logical State Of An Interface	28
3.4	Local Host Interface	28
3.5	Buffer Management.....	29
3.5.1	Large Buffer Solution.....	30
3.5.2	Linked List Solutions (mbufs)	30
3.5.3	Our Example Solution	30
3.5.4	Other Suffer Issues	31
3.6	Demultiplexing Incoming Packets	32
3.7	Summary	34
3.8	FOR FURTHER STUDY	35
3.9	EXERCISES.....	35
4	Address Discovery And Binding (ARP).....	36
4.1	Introduction	36
4.2	Conceptual Organization Of ARP Software	36
4.3	Example ARP Design	37
4.4	Data Structures For The ARP Cache	38
4.5	ARP Output Processing	41
4.5.1	Searching The ARP Cache.....	41
4.5.2	Broadcasting An ARP Request	42
4.5.3	Output Procedure.....	43
4.6	ARP Input Processing.....	46
4.6.1	Adding Resolved Entries To The Table	46
4.6.2	Sending Waiting Packets	47
4.6.3	ARP Input Procedure.....	47
4.7	ARP Cache Management.....	50
4.7.1	Allocating A Cache Entry	50
4.7.2	Periodic Cache Maintenance	52
4.7.3	Deallocating Queued Packets	53
4.8	ARP Initialization.....	54
4.9	ARP Configuration Parameters	55
4.10	Summary	55
4.11	FOR FURTHER STUDY	56
4.12	EXERCISES.....	56
5	IP: Global Software Organization	57
5.1	Introduction	57
5.2	The Central Switch.....	57



5.3	IP Software Design.....	58
5.4	IP Software Organization And Datagram Flow	59
5.4.1	A Policy For Selecting Incoming Datagram.....	59
5.4.2	Allowing The IP Process To Block.....	61
5.4.3	Definitions Of Constants Used By IP	65
5.4.4	Checksum Computation	68
5.4.5	Handling Directed Broadcasts	68
5.4.6	Recognizing A Broadcast Address.....	71
5.5	Byte-Ordering In The IP Header	72
5.6	Sending A Datagram To IP	73
5.6.1	Sending Locally-Generated Datagrams.....	73
5.6.2	Sending Incoming Datagrams	75
5.7	Table Maintenance.....	76
5.8	Summary	77
5.9	FOR FURTHER STUDY	78
5.10	EXERCISES.....	78
6	IP: Routing Table And Routing Algorithm.....	80
6.1	Introduction	80
6.2	Route Maintenance And Lookup.....	80
6.3	Routing Table Organization.....	81
6.4	Routing Table Data Structures.....	81
6.5	Origin Of Routes And Persistence.....	83
6.6	Routing A Datagram.....	84
6.6.1	Utility Procedures.....	84
6.6.2	Obtaining A Route	88
6.6.3	Data Structure Initialization.....	89
6.7	Periodic Route Table Maintenance	90
6.7.1	Adding A Route	92
6.7.2	Deleting A Route	96
6.8	IP Options Processing.....	98
6.9	Summary	99
6.10	FOR FURTHER STUDY	100
6.11	EXERCISES.....	100
7	IP: Fragmentation And Reassembly	102
7.1	Introduction	102
7.2	Fragmenting Datagrams	102
7.2.1	Fragmenting Fragments.....	103



7.3	Implementation Of Fragmentation	103
7.3.1	Sending One Fragment	105
7.3.2	Copying A Datagram Header.....	106
7.4	Datagram Reassembly	108
7.4.1	Data Structures	108
7.4.2	Mutual Exclusion.....	109
7.4.3	Adding A Fragment To A List.....	109
7.4.4	Discarding During Overflow	112
7.4.5	Testing For A Complete Datagram	113
7.4.6	Building A Datagram From Fragments	115
7.5	Maintenance Of Fragment Lists	116
7.6	Initialization	118
7.7	Summary	119
7.8	FOR FURTHER STUDY	119
7.9	EXERCISES.....	119
8	IP: Error Processing (ICMP)	121
8.1	Introduction	121
8.2	ICMP Message Formats	121
8.3	Implementation Of ICMP Messages	121
8.4	Handling Incoming ICMP Messages.....	124
8.5	Handling An ICMP Redirect Message	126
8.6	Setting A Subnet Mask	128
8.7	Choosing A Source Address For An ICMP Packet	129
8.8	Generating ICMP Error Messages.....	130
8.9	Avoiding Errors About Errors.....	133
8.10	Allocating A Buffer For ICMP	134
8.11	The Data Portion Of An ICMP Message	136
8.12	Generating An ICMP Redirect Message.....	138
8.13	Summary	140
8.14	FOR FURTHER STUDY	140
8.15	EXERCISES.....	140
9	IP: Multicast Processing (IGMP)	141
9.1	Introduction	141
9.2	Maintaining Multicast Group Membership Information	141
9.3	A Host Group Table.....	142
9.4	Searching For A Host Group	144
9.5	Adding A Host Group Entry To The Table	145



9.6	Configuring The Network Interface For A Multicast Address.....	146
9.7	Translation Between IP and Hardware Multicast Addresses	148
9.8	Removing A Multicast Address From The Host Group Table	150
9.9	Joining A Host Group	151
9.10	Maintaining Contact With A Multicast Router	153
9.11	Implementing IGMP Membership Reports	154
9.12	Computing A Random Delay.....	155
9.13	A Process To Send IGMP Reports	157
9.14	Handling Incoming IGMP Messages.....	158
9.15	Leaving A Host Group.....	159
9.16	Initialization Of IGMP Data Structures	161
9.17	Summary	162
9.18	FOR FURTHER STUDY	162
9.19	EXERCISES.....	163
10	UDP: User Datagram.....	164
10.1	Introduction	164
10.2	UDP Ports And Demultiplexing	164
	10.2.1 Ports Used For Pairwise Communication.....	165
	10.2.2 Ports Used For Many-One Communication	165
	10.2.3 Modes Of Operation.....	166
	10.2.4 The Subtle Issue Of Demultiplexing	166
10.3	UDP	168
	10.3.1 UDP Declarations	168
	10.3.2 Incoming Datagram Queue Declarations.....	170
	10.3.3 Mapping UDP port numbers To Queues.....	172
	10.3.4 Allocating A Free Queue	172
	10.3.5 Converting To And From Network Byte Order	173
	10.3.6 Processing An Arriving Datagram.....	174
	10.3.7 UDP Checksum Computation.....	176
10.4	UDP Output Processing.....	178
	10.4.1 Sending A UDP Datagram.....	179
10.5	Summary	181
10.6	FOR FURTHER STUDY	181
10.7	EXERCISES.....	181
11	TCP: Data Structures And Input Processing.....	183
11.1	Introduction	183
11.2	Overview Of TCP Software.....	183



11.3	Transmission Control Blocks	184
11.4	TCP Segment Format	188
11.5	Sequence Space Comparison.....	190
11.6	TCP Finite State Machine.....	191
11.7	Example State Transition.....	193
11.8	Declaration Of The Finite State Machine	193
11.9	TCB Allocation And Initialization.....	195
11.9.1	Allocating A TCB	195
11.9.2	Deallocating A TCB.....	196
11.10	Implementation Of The Finite State Machine	197
11.11	Handling An Input Segment	198
11.11.1	Converting A TCP Header To Local Byte Order	200
11.11.2	Computing The TCP Checksum	201
11.11.3	Finding The TCB For A Segment.....	202
11.11.4	Checking Segment Validity	204
11.11.5	Choosing A Procedure For the Current State.....	205
11.12	Summary	207
11.13	FOR FURTHER STUDY	207
11.14	EXERCISES.....	207
12	TCP: Finite State Machine Implementation	209
12.1	Introduction	209
12.2	CLOSED State Processing	209
12.3	Graceful Shutdown.....	210
12.4	Timed Delay After Closing.....	210
12.5	TIME-WAIT State Processing.....	211
12.6	CLOSING State Processing	212
12.7	FIN-WAIT-2 State Processing	214
12.8	FIN-WAIT-1 State Processing	215
12.9	CLOSE-WAIT State Processing	217
12.10	LAST-ACK State Processing	218
12.11	ESTABLISHED State Processing	219
12.12	Processing Urgent Data In A Segment	220
12.13	Processing Other Data In A Segment	222
12.14	Keeping Track Of Received Octets	224
12.15	Aborting A TCP Connection.....	227
12.16	Establishing A TCP Connection	228
12.17	Initializing A TCB	228



12.18	SYN-SENT State Processing.....	230
12.19	SYN-RECEIVED State Processing.....	231
12.20	LISTEN State Processing	234
12.21	Initializing Window Variables For A New TCB	235
12.22	Summary	237
12.23	FOR FURTHER STUDY	237
12.24	EXERCISES.....	238
13	TCP: Output Processing	239
13.1	Introduction	239
13.2	Controlling TCP Output Complexity.....	239
13.3	The Four TCP Output States.....	240
13.4	TCP Output As A Process	240
13.5	TCP Output Messages	241
13.6	Encoding Output States And TCB Numbers	241
13.7	Implementation Of The TCP Output Process	242
13.8	Mutual Exclusion	243
13.9	Implementation Of The IDLE State	243
13.10	Implementation Of The PERSIST State	244
13.11	Implementation Of The TRANSMIT State	245
13.12	Implementation Of The RETRANSMIT State	247
13.13	Sending A Segment	247
13.14	Computing The TCP Data Length.....	251
13.15	Computing Sequence Counts	252
13.16	Other TCP Procedures	252
13.16.1	Sending A Reset.....	252
13.16.2	Converting To Network Byte Order	254
13.16.3	Waiting For Space In The Output Buffer.....	255
13.16.4	Awakening Processes Waiting For A TCB	256
13.16.5	Choosing An Initial Sequence Number	258
13.17	Summary	259
13.18	FOR FURTHER STUDY	259
13.19	EXERCISES.....	259
14	TCP: Timer Management	261
14.1	Introduction	261
14.2	A General Data Structure For Timed Events	261
14.3	A Data Structure For TCP Events.....	262
14.4	Timers, Events, And Messages.....	263



14.5	The TCP Timer Process	263
14.6	Deleting A TCP Timer Event	266
14.7	Deleting All Events For A TCB	267
14.8	Determining The Time Remaining For An Event	268
14.9	Inserting A TCP Timer Event	269
14.10	Starting TCP Output Without Delay	271
14.11	Summary	272
14.12	FOR FURTHER STUDY	272
14.13	EXERCISES.....	272
15	TCP: Flow Control And Adaptive Retransmission.....	274
15.1	Introduction	274
15.2	The Difficulties With Adaptive Retransmission	275
15.3	Tuning Adaptive Retransmission.....	275
15.4	Retransmission Timer And Backoff	275
15.4.1	Karn's Algorithm	275
15.4.2	Retransmit Output State Processing	276
15.5	Window-Based Flow Control	277
15.5.1	Silly Window Syndrome.....	278
15.5.2	Receiver-Side Silly Window Avoidance.....	278
15.5.3	Optimizing Performance After A Zero Window	279
15.5.4	Adjusting The Sender's Window	280
15.6	Maximum Segment Size Computation.....	282
15.6.1	The Sender's Maximum Segment Size	282
15.6.2	Option Processing.....	284
15.6.3	Advertising An Input Maximum Segment Size	285
15.7	Congestion Avoidance And Control	286
15.7.1	Multiplicative Decrease	286
15.8	Slow-Start And Congestion Avoidance	287
15.8.1	Slow-start.....	287
15.8.2	Slower Increase After Threshold	287
15.8.3	Implementation Of Congestion Window Increase	288
15.9	Round Trip Estimation And Timeout	290
15.9.1	A Fast Mean Update Algorithm.....	290
15.9.2	Handling Incoming Acknowledgements.....	291
15.9.3	Generating Acknowledgments For Data Outside The Window.....	294
15.9.4	Changing Output State After Receiving An Acknowledgement.....	295
15.10	Miscellaneous Notes And Techniques	296



15.11	Summary	296
15.12	FOR FURTHER STUDY	297
15.13	EXERCISES.....	297
16	TCP: Urgent Data Processing And The Push Function	299
16.1	Introduction	299
16.2	Out-Of-Band Signaling	299
16.3	Urgent Data	300
16.4	Interpreting The Standard.....	300
16.4.1	The Out-Of-Band Data Interpretation	300
16.4.2	The Data Marie Interpretation	302
16.5	Configuration For Berkeley Urgent Pointer Interpretation.....	303
16.6	Informing An Application.....	303
16.6.1	Multiple Concurrent Application Programs.....	304
16.7	Reading Data From TCP	304
16.8	Sending Urgent Data	307
16.9	TCP Push Function.....	308
16.10	Interpreting Push With Out-Of-Order Delivery.....	308
16.11	Implementation Of Push On Input	309
16.12	Summary	310
16.13	FOR FURTHER STUDY	311
16.14	EXERCISES.....	311
17	Socket-Level Interface	312
17.1	Introduction	312
17.2	Interfacing Through A Device	312
17.2.1	Single Byte I/O	313
17.2.2	Extensions For Non-Transfer Functions.....	313
17.3	TCP Connections As Devices	314
17.4	An Example TCP Client Program	314
17.5	An Example TCP Server Program.....	316
17.6	Implementation Of The TCP Master Device	318
17.6.1	TCP Master Device Open Function.....	318
17.6.2	Forming A Passive TCP Connection	319
17.6.3	Forming An Active TCP Connection.....	320
17.6.4	Allocating An Unused Local Port.....	322
17.6.5	Completing An Active Connection.....	323
17.6.6	Control For The TCP Master Device.....	325
17.7	Implementation Of A TCP Slave Device.....	326



17.7.1	Input From A TCP Slave Device	326
17.7.2	Single Byte Input From A TCP Slave Device.....	328
17.7.3	Output Through A TCP Slave Device.....	329
17.7.4	Closing A TCP Connection.....	331
17.7.5	Control Operations For A TCP Slave Device	333
17.7.6	Accepting Connections From A Passive Device	335
17.7.7	Changing The Size Of A Listen Queue.....	335
17.7.8	Acquiring Statistics From A Slave Device	336
17.7.9	Setting Or Clearing TCP Options	338
17.8	Initialization Of A Slave Device.....	339
17.9	Summary	340
17.10	FOR FURTHER STUDY	341
17.11	EXERCISES.....	341
18	RIP: Active Route Propagation And Passive Acquisition.....	343
18.1	Introduction	343
18.2	Active And Passive Mode Participants.....	344
18.3	Basic RIP Algorithm And Cost Metric	344
18.4	Instabilities And Solutions.....	345
18.4.1	Count To Infinity	345
18.4.2	Gateway Crashes And Route Timeout.....	345
18.4.3	Split Horizon	346
18.4.4	Poison Reverse	347
18.4.5	Route Timeout With Poison Reverse.....	348
18.4.6	Triggered Updates	348
18.4.7	Randomization To Prevent Broadcast Storms	348
18.5	Message Types.....	349
18.6	Protocol Characterization	349
18.7	Implementation Of RIP	350
18.7.1	The Two Styles Of Implementation.....	350
18.7.2	Declarations	351
18.7.3	Conceptual Organization For Output.....	353
18.8	The Principle RIP Process	353
18.8.1	Must Be Zero Field Must Be Zero.....	355
18.8.2	Processing An Incoming Response.....	356
18.8.3	Locking During Update	358
18.8.4	Verifying An Address	358
18.9	Responding To An Incoming Request	359



18.10	Generating Update Messages	360
18.11	Initializing Copies Of An Update Message	361
18.11.1	Adding Routes to Copies Of An Update Message.....	363
18.11.2	Computing A Metric To Advertise.....	364
18.11.3	Allocating A Datagram For A RIP Message	365
18.12	Generating Periodic RIP Output.....	367
18.13	Limitations Of RIP	368
18.14	Summary	368
18.15	FOR FURTHER STUDY	368
18.16	EXERCISES.....	369
19	OSPF: Route Propagation With An SPF Algorithm	370
19.1	Introduction	370
19.2	OSPF Configuration And Options.....	370
19.3	OSPF's Graph-Theoretic Model	371
19.4	OSPF Declarations	374
19.4.1	OSPF Packet Format Declarations	375
19.4.2	OSPF Interlace Declarations	376
19.4.3	Global Constant And Data Structure Declarations	378
19.5	Adjacency And Link State Propagation.....	380
19.6	Discovering Neighboring Gateways With Hello	381
19.7	Sending Hello Packets.....	383
19.7.1	A Template For Hello Packets	385
19.7.2	The Hello Output Process.....	386
19.8	Designated Router Concept.....	388
19.9	Electing A Designated Router	389
19.10	Reforming Adjacencies After A Change.....	393
19.11	Handling Arriving Hello Packets.....	395
19.12	Adding A Gateway To The Neighbor List	397
19.13	Neighbor State Transitions	399
19.14	OSPF Timer Events And Retransmissions	400
19.15	Determining Whether Adjacency Is Permitted	402
19.16	Handling OSPF input	403
19.17	Declarations And Procedures For Link State Processing	406
19.18	Generating Database Description Packets	409
19.19	Creating A Template	411
19.20	Transmitting A Database Description Packet	412
19.21	Handling An Arriving Database Description Packet	414



19.21.1	Handling A Packet In The EXSTART State.....	415
19.21.2	Handling A Packet In The EXCHNG State	417
19.21.3	Handling A. Packet In The FULL State.....	418
19.22	Handling Link State Request Packets.....	419
19.23	Building A Link State Summary.....	421
19.24	OSPF Utility Procedures	423
19.25	Summary	426
19.26	FOR FURTHER STUDY	427
19.27	EXERCISES.....	427
20	SNMP: MIB Variables, Representations, And Bindings	428
20.1	Introduction	428
20.2	Server Organization And Name Mapping	429
20.3	MIB Variables.....	429
20.3.1	Fields Within tables	430
20.4	MIB Variable Names	430
20.4.1	Numeric Representation Of Names	431
20.5	Lexicographic Ordering Among Names.....	431
20.6	Prefix Removal.....	432
20.7	Operations Applied To MIB Variables	433
20.8	Names For Tables	433
20.9	Conceptual Threading Of The Name Hierarchy.....	434
20.10	Data Structure For MIB Variables	435
20.10.1	Using Separate Functions To Perform Operations.....	437
20.11	A Data Structure For Fast Lookup.....	437
20.12	Implementation Of The Hash Table	439
20.13	Specification Of MIB Bindings.....	439
20.14	Internal Variables Used In Bindings	444
20.15	Hash Table Lookup.....	445
20.16	SNMP Structures And Constants	448
20.17	ASN.1 Representation Manipulation.....	451
20.17.1	Representation Of Length.....	452
20.17.2	Converting Integers To ASN.1 Form.....	454
20.17.3	Converting Object Ids To ASN.1 Form	456
20.17.4	A Generic Routine For Converting Values	459
20.18	Summary	461
20.19	FOR FURTHER STUDY	462
20.20	EXERCISES.....	462



21	SNMP: Client And Server	463
21.1	Introduction	463
21.2	Data Representation In The Server.....	463
21.3	Server Implementation	464
21.4	Parsing An SNMP Message.....	466
21.5	Converting ASN.1 Names In The Binding List.....	470
21.6	Resolving A Query	471
21.7	Interpreting The Get-Next Operation	473
21.8	Indirect Application Of Operations	474
21.9	Indirection For Tables.....	477
21.10	Generating A Reply Message Backward	478
21.11	Converting From Internal Form to ASN.1.....	481
21.12	Utility Functions Used By The Server	482
21.13	Implementation Of An SNMP Client	483
21.14	Initialization Of Variables.....	485
21.15	Summary	487
21.16	FOR FURTHER STUDY	487
21.17	EXERCISES.....	487
22	SNMP: Table Access Functions	489
22.1	Introduction	489
22.2	Table Access	489
22.3	Object Identifiers For Tables	490
22.4	Address Entry Table Functions.....	490
22.4.1	Get Operation For The Address Entry Table	492
22.4.2	Get-First Operation For The Address Entry Table.....	493
22.4.3	Get-Next Operation For The Address Entry Table	494
22.4.4	Incremental Search In The Address Entry Table	496
22.4.5	Set Operation For The Address Entry Table	497
22.5	Address Translation Table Functions.....	497
22.5.1	Get Operation For The Address Translation Table	499
22.5.2	Get-First Operation For The Address Translation Table.....	500
22.5.3	Get-Next Operation For The Address Translation Table	502
22.5.4	Incremental Search In The Address Entry Table	503
22.5.5	Order From Chaos	504
22.5.6	Set Operation For The Address Translation Table	505
22.6	Network Interface Table Functions	506
22.6.1	Interface Table ID Matching.....	506



22.6.2	Get Operation For The Network Interface Table	507
22.6.3	Get-First Operation For The Network Interface Table	510
22.6.4	Get-Next Operation For The Network Interface Table	511
22.6.5	Set Operation For The Network Interface Table.....	513
22.7	Routing Table Functions.....	514
22.7.1	Get Operation For The Routing Table	515
22.7.2	Get-First Operation For The Routing Table.....	517
22.7.3	Get-Next Operation For The Routing Table	518
22.7.4	Incremental Search In The Routing Table	520
22.7.5	Set Operation For The Routing Table	521
22.8	TCP Connection Table Functions	523
22.8.1	Get Operation For The TCP Connection Table.....	525
22.8.2	Get-First Operation For The TCP Connection Table	526
22.8.3	Get-Next Operation For The TCP Connection Table.....	527
22.8.4	Incremental Search In The TCP Connection Table.....	529
22.8.5	Set Operation For The TCP Connection Table	530
22.9	Summary	531
22.10	FOR FURTHER STUDY	531
22.11	EXERCISES.....	531
23	Implementation In Retrospect	532
23.1	Introduction	532
23.2	Statistical Analysis Of The Code.....	532
23.3	Lines Of Code For Each Protocol	532
23.4	Functions And Procedures For Each Protocol	534
23.5	Summary	535
23.6	EXERCISES.....	536
24	Appendix 1Cross Reference Of Procedure Calls.....	537
24.1	Introduction	537
25	Appendix 2 Xinu Functions And Constants Used In The Code	559
25.1	Introduction	559
25.2	Alphabetical Listing	559
25.3	Xinu System Include Files.....	567
26	Bibliography.....	571
27	Index.....	580



1 *Introduction And Overview*

1.1 TCP/IP Protocols

The TCP/IP Internet Protocol Suite has become, de facto, the standard for open system interconnection in the computer industry. Computer systems worldwide use TCP/IP Internet protocols to communicate because TCP/IP provides the highest degree of interoperability, encompasses the widest set of vendors' systems, and runs over more network technologies than any other protocol suite. Research and education institutions use TCP/IP as their primary platform for data communication. In addition, industries that use TCP/IP include aerospace, automotive, electronics, hotel, petroleum, printing, pharmaceutical, and many others.

Besides conventional use on private industrial networks, many academic, government, and military sites use TCP/IP protocols to communicate over the connected Internet. Schools with TCP/IP connections to the Internet exchange information and research results more quickly than those that are not connected, giving researchers at such institutions a competitive advantage.

1.2 The Need To Understand Details

Despite its popularity and widespread use, the details of TCP/IP protocols and the structure of software that implements them remain a mystery to most computer professionals. While it may seem that understanding the internal details is not important, programmers who use TCP/IP learn that they can produce more robust code if they understand how the protocols operate. For example, programmers who understand TCP urgent data processing can add functionality to their applications that is impossible otherwise.

Understanding even simple ideas such as how TCP buffers data can help programmers design, implement, and debug applications. For example, some programs that use TCP fail because programmers misunderstand the relationships between output buffering, segment transmission, input buffering, and the TCP push operation. Studying



the details of TCP input and output allows programmers to form a conceptual model that explains how the pieces interact, and helps them understand how to use the underlying mechanisms.

1.3 Complexity Of Interactions Among Protocols

The main reason the TCP/IP technology remains so elusive is that documentation often discusses each protocol independently, without considering how multiple protocols operate together. A protocol standard document, for example, usually describes how a single protocol should operate; it discusses the action of the protocol and its response to messages in isolation from the rest of the system. The most difficult aspect of protocols to understand, however, lies in their interaction. When one considers the operation of all protocols together, the interactions produce complicated, and sometimes unexpected, effects. Minor details that may seem unimportant suddenly become essential. Heuristics to handle problems and nuances in protocol design can make important differences in overall operation or performance.

As many programmers have found, the interactions among protocols often dictate how they must be implemented. Data structures must be chosen with all protocols in mind. For example, IP uses a routing table to make decisions about how to forward datagrams. However, the routing table data structures cannot be chosen without considering protocols such as the Routing Information Protocol, the Internet Control Message Protocol, and the Exterior Gateway Protocol, because all may need to update routes in the table. More important, the routing table update policies must be chosen carefully to accommodate all protocols or the interaction among them can lead to unexpected results. We can summarize:

The TCP/IP technology comprises many protocols that all interact. To fully understand the details and implementation of a protocol, one must consider its interaction with other protocols in the suite.

1.4 The Approach In This Text

This book explores TCP/IP protocols in great detail. It reviews concepts and explains nuances in each protocol. It discusses abstractions that underlie TCP/IP software, and describes the data structures and procedures that implement the protocols. Finally, it reviews design choices, and discusses the consequence of design alternatives.

To provide a concrete example of protocol implementation, and to help the reader understand the relationships among protocols, the text takes an integrated view — it focuses on a complete working system. It shows data structures and source code, and explains the principles underlying each.



Code from the example system helps answer many questions and explain many subtleties that could not be understood otherwise. It fills in details and provides the reader with an understanding of the relative difficulty of implementing each part. It shows how the judicious choice of data representation can make some protocols easier to implement (and conversely how a poor choice of representation can make the implementation tedious and difficult). The example code allows the reader to understand ideas like urgent data processing and network management that spread across many parts of the code. More to the point, the example system clearly shows the reader how protocols interact and how the implementation of individual protocols can be integrated. To summarize;

To explain the details, internal organization, and implementation of TCP/IP protocols, this text focuses on an example working system. Source code for the example system allows the reader to understand how the protocols interact and how the software can be integrated into a simple and efficient system.

1.5 The Importance Of Studying Code

The example TCP/IP system is the centerpiece of the text. To understand the data structures, the interaction among procedures, and the subtleties of the protocol internals, it is necessary to read and study the source code.^① Thus,

The example programs should be considered part of the text, and not merely a supplement to it.

1.6 The Xinu Operating System

On most machines, TCP/IP protocol software resides in the operating system kernel. A single copy of the TCP/IP software is shared by all application programs. The software presented in this text is part of the Xinu operating system.^② We have chosen to use Xinu for several reasons. First, Xinu has been documented in two textbooks, so source code for the entire system is completely available for study. Second, because Xinu does not have cost accounting or other administrative overhead, the TCP/IP code in Xinu is free from unnecessary details and, therefore, much easier to understand. Third, because the text concentrates on explaining abstractions underlying the code, most of the

^① To make it easy to use computer tools to explore parts of the system, the publisher has made machine readable copies of the code from the text available.

^② Xinu is a small, elegant operating system that has many features similar to UNIX. Several vendors have used versions of Xinu as an embedded system in commercial products.



ideas presented apply directly to other implementations. Fourth, using Xinu and TCP/IP software designed by the authors completely avoids the problem of commercial licensing, and allows us to sell the text freely. While the Xinu system and the TCP/IP code presented have resulted from a research project, readers will find that they are surprisingly complete and, in many cases, provide more functionality than their commercial counterparts. Finally, because we have attempted to follow the RFC specifications rigorously, readers may be surprised to learn that the Xinu implementation of TCP/IP obeys the protocols standards more strictly than many popular implementations.

1.7 Organization Of The Remainder Of The Book

This text is organized around the TCP/IP protocol stack in approximately the same order as Volume I. It begins with a review of the operating system functions that TCP uses, followed by a brief description of the device interface layer. Remaining chapters describe the TCP/IP protocols, and show example code to illustrate the implementation of each.

Some chapters describe entire protocols, while others concentrate on specific aspects of the design. For example, Chapter 15 discusses heuristics for round trip estimation, retransmission, and exponential backoff. The code appears in the chapter that is most pertinent; references appear in other chapters.

Appendix 1 contains a cross reference of the procedures that comprise the TCP/IP protocol software discussed throughout the text. For each procedure, function, or inline macro, the cross reference tells the file in which it is defined, the page on which that file appears in the text, the list of procedures called in that file, and the list of procedures that call it. The cross reference is especially helpful in finding the context in which a given procedure is called, something that is not immediately obvious from the code.

Appendix 2 provides a list of those functions and procedures used in the code that are not contained in the text. Most of the procedures listed come from the C run-time support libraries or the underlying operating system, including the Xinu system calls that appear in the TCP/IP code. For each procedure or function, Appendix 2 lists the name and arguments, and gives a brief description of the operation it performs.

1.8 Summary

This text explores the subtleties of TCP/IP protocols, details of their implementation, and the internal structure of software that implements them. It focuses on an example implementation from the Xinu operating system, including the source code that forms a working system. Although the Xinu implementation was not designed



as a commercial product, it obeys the protocol standards. To fully understand the protocols, the reader must study the example programs. The appendices help the reader understand the code. They provide a cross reference of the TCP/IP routines and a list of the operating system routines used.

1.9 FOR FURTHER STUDY

Volume I [Comer 1991] presents the concepts underlying the TCP/IP Internet Protocol Suite, a synopsis of each protocol, and a summary of Internet architecture. We assume the reader is already familiar with most of the material in volume I. Corner [1984] and Comer [1987] describe the structure of the Xinu operating system, including an early version of ARP, UDP, and IP code. Leffler, McKusick, Karels, and Quarterman [1989] describes the Berkeley UNIX system. Stevens [1990] provides examples of using the TCP/IP interface in various operating systems.



2 The Structure Of TCP/IP Software In An Operating System

2.1 Introduction

Most TCP/IP software runs on computers that use an operating system to manage resources, like peripheral devices. Operating systems provide support for concurrent processing. Even on machines with a single processor they give the illusion that multiple programs can execute simultaneously by switching the CPU among them rapidly. In addition, operating systems manage main memory that contains executing programs, as well as secondary (nonvolatile) storage, where file systems reside.

TCP/IP software usually resides in the operating system, where it can be shared by all application programs running on the machine. That is, the operating system contains a single copy of the code for a protocol like TCP, even though multiple programs can invoke that code. As we will see, code that can be used by multiple, concurrently executing programs is significantly more complex than code that is part of a single program.

This chapter provides a brief overview of operating system concepts that we will use throughout the text. It shows the general structure of protocol software and explains in general terms how the software fits into the operating system. Later chapters review individual pieces of protocol software and present extensive detail.

The examples in this chapter come from Xinu, the operating system used throughout the text. Although the examples refer to system calls and argument that are only available in Xinu, the concepts apply across a wide variety of operating systems, including the popular UNIX timesharing system.



2.2 The Process Concept

Operating systems provide several abstractions that are needed for understanding the implementation of TCP/IP protocols. Perhaps the most important is that of a process (sometimes called a task or thread of control). Conceptually, a process is a computation that proceeds independent of other computations. An operating system provides mechanisms to create new processes and to terminate existing processes. In the example system we will use, a program calls function `create` to form a new process. `Create` returns an integer process identifier used to reference the process when performing operations on it.

```
procid = create (arguments) ; /* create a new process */
```

Once created, a process proceeds independent of its creator. To terminate an existing process, a program calls `kill`, passing as an argument the process identifier that `create` returned.

```
kill(procid) ; /* terminate a process */
```

Unlike conventional (sequential) programs in which a single thread of control steps through the code belonging to a program, processes are not bound to any particular code or data. The operating system can allow two or more processes to execute a single piece of code. For example, two processes can execute operating system code concurrently, even though only one copy of the operating system code exists. In fact, it is possible for two or more processes to execute code in a single procedure concurrently.

Because processes execute independently, they can proceed at different rates. In particular, processes sometimes perform operations that cause them to be blocked or suspended. For example, if a process attempts to read a character from a keyboard, it may need to wait for the user to press a key. To avoid having the process use the CPU while waiting, the operating system blocks the process but allows others to continue executing. Later, when the operating system receives a keystroke event, it will allow the process waiting for that keystroke to resume execution.

The implementation of TCP/IP software we will examine uses multiple, concurrently executing processes. Instead of trying to write a single program that handles all possible sequences of events, the code uses processes to help partition the software into smaller, more manageable pieces. As we will see, using processes simplifies the design and keeps the code easy to understand and modify.

Processes are especially useful in handling the timeout and retransmission algorithms found in many protocols. Using a single program to implement timeout for multiple protocols makes the program complex, because the timeouts can overlap. For example, consider trying to write a single program to manage timers for all TCP/IP protocols. A high-level protocol like TCP may create a segment, encapsulate it in a datagram, send the datagram, and start a timer. Meanwhile, IP must route the datagram and pass it to the network interface. Eventually a low-level protocol like ARP may be



invoked and it may go through several cycles of transmitting a request, setting a timer, having the timer expire, and retransmitting the request independent of the TCP timer. In a single program, it can be difficult to handle events when a timer for one protocol expires while the program is executing code for another protocol. If the system uses a separate process to implement each protocol that requires a timeout, the process only needs to handle timeout events related to its protocol. Thus, the code in each process is easier to understand and less prone to errors.

2.3 Process Priority

We said that all processes execute concurrently, but that is an oversimplification. In fact, each process is assigned a priority by the programmer who designs the software. The operating system honors priorities when granting processes the use of the CPU. The priority scheme we will use is simple and easy to understand: the CPU is granted to the highest priority process that is not blocked; if multiple processes share the same high priority, the system will switch the CPU among them rapidly.

The priority scheme is valuable in protocol software because it allows a programmer to give one process precedence over another. For example, compare an ordinary application program to the protocol software that must accept packets from the hardware as they arrive. The designer can assign higher priority to the process that implements protocol software, forcing it to take precedence over application processes. Because the operating system handles all the details of process scheduling, the processes themselves need not contain any code to handle scheduling,

2.4 Communicating Processes



If each process is an independent computation, how can data flow from one to another? The answer is that the operating system must provide mechanisms that permit processes to communicate. We will use three such mechanisms: counting semaphores, ports, and message passing.

A counting semaphore is a general purpose process synchronization mechanism. The operating system provides a function, screate, that can be called to create a semaphore when one is needed. Screate returns a semaphore identifier that must be used in subsequent operations on the semaphore.

```
semid = screate (initcount) ; /* create semaphore, specifying count */
```

Each semaphore contains an integer used for counting; the caller gives an initial value for the integer when creating the semaphore. Once a semaphore has been created, processes can use the operating system functions wait and signal to manipulate the count. When a process calls wait, the operating system decrements the semaphore's count by 1,



and blocks the process if the count becomes negative. When a process calls signal, the operating system increments the semaphore count, and unblocks one process if any process happens to be blocked on that semaphore.

Although the semantics of wait and signal may seem confusing, they can be used to solve several important process synchronization problems. Of most importance, they can be used to provide mutual exclusion. Mutual exclusion means allowing only one process to execute a given piece of code at a given time; it is important because multiple processes can execute the same piece of code. To understand why mutual exclusion is essential, consider what might happen if two processes concurrently execute code that adds a new item to a linked list. If the two processes execute concurrently, they might each start at the same point in the list and try to insert their new item. Depending on how much CPU time the processes receive, one of them could execute for a short time, then the other, then the first, and so on. As a result, one could override the other (leaving one of the new items out altogether), or they could produce a malformed list that contained incorrect pointers.

To prevent processes from interfering with one another, all the protocol software that can be executed by multiple processes must use semaphores to implement mutual exclusion. To do so, the programmer creates a semaphore with initial count of 1 for every piece of code that must be protected.

```
s = screate(1); /* create mutual exclusion semaphore */
```

Then, the programmer places calls to wait and signal around the critical piece of code as the following illustrates.

```
wait(s); /* before code to be protected */
...
...critical code...
signal(s); /* after code to be protected */
```

The first process that executes wait(s) decrements the count of semaphore s to zero and continues execution (because the count remains nonnegative). If that process finishes and executes signal(s), the count of s returns to 1. However, if the first process is still using the critical code when a second process calls wait(s), the count becomes negative and the second process will be blocked. Similarly, if a third happens to execute wait(s) during this time, the count remains negative and the third process will also be blocked. When the first process finally finishes using the critical code, it will execute signal(s), incrementing the count and unblocking the second process. The second process will begin executing the critical code while the third waits. When the second process finishes and executes signal(s), the third can begin using the critical code. The point is that at any time only one process can execute the critical code; all others that try will be blocked by the semaphore.

In addition to providing mutual exclusion, examples in this text use semaphores to provide synchronization for queue access. Synchronization is needed because queues have finite capacity. Assume that a queue contains space for N items, and that some set



of concurrent processes is generating items to be placed in the queue. Also assume that some other set of processes is extracting items and processing them (typically many processes insert items and one process extracts them). A process that inserts items in the queue is called a producer, and a process that extracts items is called a consumer. For example, the items might be IP datagrams generated by a set of user applications, and a single IP process might extract the datagrams and route each to its destination. If the application programs producing datagrams generate them faster than the IP process can consume and route them, the queue eventually becomes full. Any producer that attempts to insert an item when the queue is full must be blocked until the consumer removes an item and makes space available. Similarly, if the consumer executes quickly, it may extract all the items from the queue and must be blocked until another item arrives. Two semaphores are required for coordination of producers and consumers as they access a queue of N items. The semaphores are initialized as follows.

```
s1 = screate(N); /* counts space in queue */  
s2 = screate(0); /* counts items in queue */
```

After the semaphores have been initialized, producers and consumers use them to synchronize. A producer executes the following

```
wait(s1); /* wait for space */  
...insert item in next available slot...  
signal(s2); /* signal item available */
```

And the consumer executes

```
wait(s2); /* wait for item in queue */  
...extract oldest item from queue...  
signal(s1); /* signal space available */
```

The semaphores guarantee that a producer process will be blocked if the queue is full, and a consumer will be blocked if the queue is empty. At all other times both producers and consumers can proceed.

2.5 Interprocess Communication

2.5.1 Ports

The port abstraction provides a rendezvous point through which processes can pass data. We think of a port as a finite queue of messages plus two semaphores that control access. A program creates a port by calling function pcreate and specifying the size of the queue as an argument. Pcreate returns an identifier used to reference the port.

```
portid = pcreate(size); /* create a port specifying size */
```

Once a port has been created, processes call procedures psend and preceive to deposit or remove items. Psend sends a message to a port.

```
psend(portid, message); /* send a message to a port */
```



It takes two arguments: a port identifier and a one-word message to send (in the TCP/IP code, the message will usually consist of a pointer to a packet).

Preceive extracts a message from a port.

```
message = preceive(port); /* extract next message from port */
```

As we suggested, the implementation uses semaphores so that psend will block the calling process if the port is full, and preceive will block the calling process if the port is empty. Once a process blocks in psend it remains blocked until another process calls preceive, and vice versa. Thus, when designing systems of processes that use ports, the programmer must be careful to guarantee that the system will not block processes forever (this is the equivalent of saying that programmers must be careful to avoid endless loops in sequential programs).

In addition to prohibiting interactions that block processes indefinitely, some designs add even more stringent requirements. They specify that a select group of processes may not block under any circumstances, even for short times. If the processes do block, the system may not operate correctly. For example, a network design may require that the network input process never block to guarantee that the entire system will not halt when application programs stop accepting incoming packets. In such cases, the process needs to check whether a call to psend will block and, if so, take alternative action (e.g., discard a packet).

To allow processes to determine whether psend will block, the system provides a function, pcount, that allows a process to find out whether a port is full.

```
n = pcount(portid); /* find out whether a port is full */
```

The process calls pcount, supplying the identifier of a port to check; pcount returns the current count of items in the port. If the count is zero no items remain in the port. If the count equals the size of the port, the port is full.

2.5.2 Message Passing

We said that processes also communicate and synchronize through message passing. Message passing allows one process to send a message directly to another. A process calls send to send a message to another process. Send takes a process identifier and a message as arguments; it sends the specified message to the specified process.

```
send(msg, pid); /* send integer msg to process pid */
```

A process calls receive to wait for a message to arrive.

```
message = receive(); /* wait for msg and return it */
```

In our system, receive blocks the caller until a message arrives, but send always proceeds. If the receiving process does not execute receive between two successive calls of send, the second call to send will return SYSERR, and the message will not be sent. It is the programmer's responsibility to construct the system in such a way that messages are not lost. To help synchronize message exchange, a program can call recvclr, a



function that removes any waiting message but does not block.

```
message = recvclr(); /* clear message buffer */
```

Because protocols often specify a maximum time to wait for acknowledgements, they often use the message passing function recvtim, a version of receive that allows the caller to specify a maximum time to wait. If a message arrives within the specified time, recvtim returns it to the caller. Otherwise, recvtim returns a special value, TIMEOUT.

```
message = recvtim(50); /* wait 5 seconds (50 tenths of a second) for a mesg and return it */
```

2.6 Device Drivers, Input, And Output

Network interface hardware transfers incoming packets from the network to the computer's memory and informs the operating system that a packet has arrived. Usually, the network interface uses the interrupt mechanism to do so. An interrupt causes the CPU to temporarily suspend normal processing and jump to code called a device driver. The device driver software takes care of minor details. For example, it resets the hardware interrupt mechanism and (possibly) restarts the network interface hardware so it can accept another packet. The device driver also informs protocol software that a packet has arrived and must be processed. Once the device driver completes its chores, it returns from the interrupt to the place where the CPU was executing when the interrupt occurred. Thus, we can think of an interrupt as temporarily "borrowing" the CPU to handle an I/O activity.

Like most operating systems, the XINU system arranges to have network interface devices interrupt the processor when a packet arrives. The device driver code handles the interrupt and restarts the device so it can accept the next packet.

The device driver also provides a convenient interface for programs that send or receive packets. In particular, it allows a process to block (wait) for an incoming packet. From the process' point of view, the device driver is hidden beneath a general-purpose I/O interface, making it easy to capture incoming packets. For example, to send a frame (packet) on an Ethernet interface, a program invokes the following:

```
write(device, buff, len); /* write one Ethernet packet */
```

where device is a device descriptor that identifies a particular Ethernet interface device, buff gives the address of a buffer that contains the frame to be sent, and len is the length of the frame measured in octets^①.

^① An octet is an 8-bit unit of data, called a byte on many systems.

2.7 Network Input and Interrupts

Now that we understand the facilities the operating system supplies, we can examine the general structure of the example TCP/IP software. Recall that the operating system contains device driver software that communicates with hardware I/O devices and handles interrupts. The code is hidden in an abstraction called a device; the system contains one such device for each network to which it attaches (most hosts have only one network interface but gateways have multiple network interfaces).

To accommodate random packet arrivals, the system needs the ability to read packets from any network interface. It is possible to solve the problem of waiting for a random interface in several ways. Some operating systems use the computer's software interrupt mechanism. When a packet arrives, a hardware interrupt occurs and the device driver performs its usual duties of accepting the packet and restarting the device. Before returning from the interrupt, the device driver tells the hardware to schedule a second, lower priority interrupt. As soon as the hardware interrupt completes, the low priority interrupt occurs exactly as if another hardware device had interrupted. This "software interrupt" suspends processing and causes the CPU to jump to code that will handle it. Thus, in some systems, all input processing occurs as a series of interrupts. The idea has been formalized in a UNIX System V mechanism known as STREAMS.

Software interrupts are efficient, but require hardware not available on all computers. To make the protocol software portable, we chose to avoid software interrupts and design code that relies only on a conventional interrupt mechanism.

Even operating systems that use conventional hardware interrupts have a variety of ways to handle multiple interfaces. Some have mechanisms that allow a single process to block on a set of input devices and be informed as soon as a packet arrives on one of them. Others use a process per interface, allowing that process to block until a packet arrives on its interface. To make the design efficient, we use the organization that Figure 2.1 illustrates.

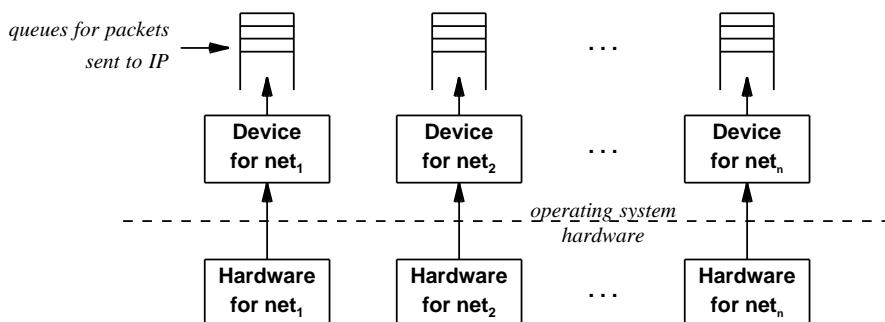


Figure 2.1 The flow of packets from the network interface hardware through the device driver in

the operating system to an input queue associated with the device.

The Ethernet interrupt routine uses the packet type field of arriving packets to determine which protocol was used in the packet. For example, if the packet type of an Ethernet packet is 0800_{16} , the packet carries an IP datagram. On networks that do not have self-identifying frames, the system designer must either choose to use a link-level protocol that identifies the packet contents, or choose the packet type a priori. The IEEE 802.2 link-level protocol is an example of the former, and Serial Line IP (SLIP) is an example of the latter.

2.8 Passing Packets To Higher Level Protocols

Because input occurs at interrupt time, the device driver code cannot call arbitrary procedures to process the packet; it must return from the interrupt quickly. Therefore, the interrupt procedure does not call IP directly. Furthermore, because the system uses a separate process to implement IP, the device driver cannot call IP directly. Instead, the system uses a queue along with the message passing primitives described earlier in this chapter to synchronize communication. When a packet that carries an IP datagram arrives, the interrupt software must enqueue the packet and invoke send to notify the IP process that a datagram has arrived. When the IP process has no packets to handle, it calls receive to wait for the arrival of another datagram. There is an input queue associated with each network device; a single IP process extracts datagrams from all queues and processes them. Figure 2.2 illustrates the concept.

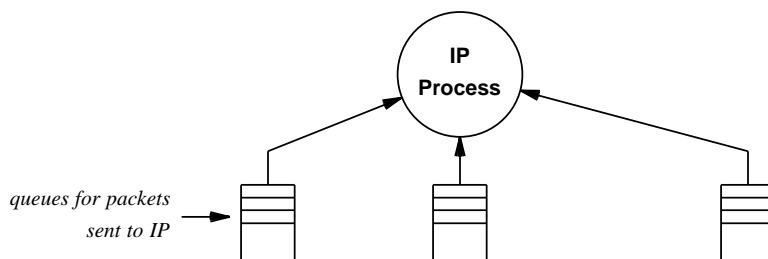


Figure 2.2 Communication between the network device drivers and the process that implements IP uses a set of queues. When a datagram arrives, the network input process enqueues it and sends a message to the IP process.

2.9 Passing Datagrams From IP To Transport Protocols

Once the IP process accepts an incoming datagram, it must decide where to send it for further processing. If the datagram carries a TCP segment, it must go to the TCP



module; if it carries a UDP datagram, it must go to the UDP module, and so on. We will examine the internals of each module later; at this point, only the process structure is important.

2.9.1 Passing Incoming Datagrams to TCP

Because TCP is complex, most designs use a separate process to handle incoming TCP segments. Because they execute as separate processes, IP and TCP must use an interprocess communication mechanism to communicate. They use the port mechanism described earlier. IP calls `psend` to deposit segments in the port, and TCP calls `preceive` to retrieve them. As we will see later, other processes send messages to TCP using this port as well.

Once TCP receives a segment, it uses the TCP protocol port numbers to find the connection to which the segment belongs. If the segment contains data, TCP will add the data to a buffer associated with the connection and return an acknowledgement to the sender. If the incoming segment carries an acknowledgement for outbound data, the TCP input process must also communicate with the TCP timer process to cancel the pending retransmission.

2.9.2 Passing Incoming Datagrams to UDP

The process structure used to handle incoming UDP datagrams is quite different from that used for TCP. Because UDP is much simpler than TCP, the UDP software module does not execute as a separate process. Instead, it consists of conventional procedures that the IP process executes to handle an incoming UDP datagram. These procedures examine the destination UDP protocol port number and use it to select an operating system queue (port) for the user datagram. The IP process deposits the UDP datagram on the appropriate port, where an application program can extract it. Figure 2.3 illustrates the difference.

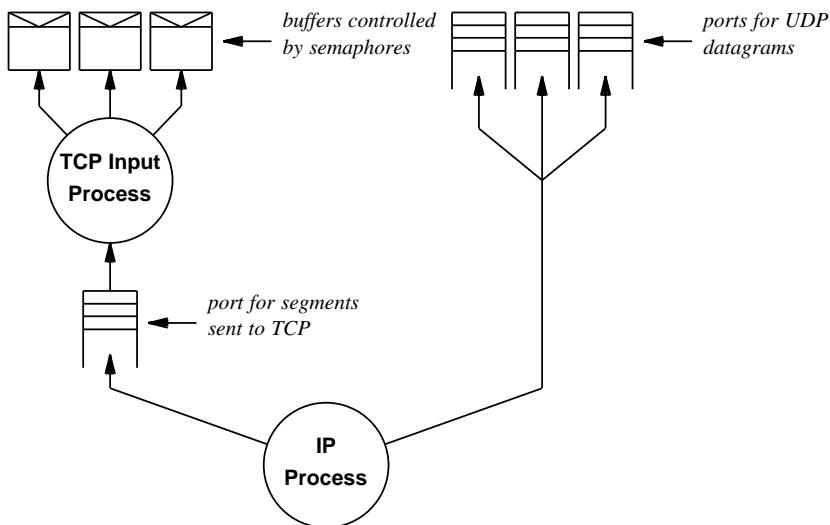


Figure 2.3 The flow of datagrams through higher layers of software. The IP process sends incoming segments to the TCP process, but places incoming UDP datagrams directly in separate ports where they can be accessed by application programs.

2.10 Delivery To Application Programs

As Figure 1.3 shows, UDP demultiplexer incoming user datagrams based on protocol port number and places them in operating system queues. Meanwhile, TCP separates incoming data streams and places the data in buffers. When an application program needs to receive either a UDP datagram or data from a TCP stream, it must access the UDP port or TCP buffer. While the details are complex, the reader should understand a simple idea at this point:

Because each application program executes as a separate process, it must use system communication primitives to coordinate with the processes that implement protocols.

For example, an application program calls the operating system function `preceive` to retrieve a UDP datagram. Of course, the interaction is much more complex when an application program interacts with a process in the operating system than when two processes inside the operating system interact.

For incoming TCP data, application programs do not use `preceive`. Instead, the system uses semaphores to control access to the data in a TCP buffer. An application



program that wishes to read incoming data from the stream calls wait on the semaphore that controls the buffer: the TCP process calls signal when it adds data to the buffer.

2.11 Information Flow On Output

Outgoing packers originate for one of two reasons. Either (1) an application program passes data to one of the high-level protocols which, in turn, sends a message (or datagram) to a lower-level protocol and eventually causes transmission on a network, or (2) protocol software in the operating system transmits information (e.g., an acknowledgement or a response to an echo request). In either case, a hardware frame must be sent out over a particular network interface.

To help isolate the transmission of packets from the execution of processes that implement application programs and protocols, the system has a separate output queue for each network interface. Figure 2.4 illustrates the design.

The queues associated with output devices provide an important piece of the design. They allow processes to generate a packet, enqueue it for output, and continue execution without waiting for the packet to be sent. Meanwhile, the hardware can continue transmitting packets simultaneously. If the hardware is idle when a packet arrives (i.e., there are no packets in the queue), the process performing output enqueues its packet and calls a device driver routine to start the hardware. When the output operation completes, the hardware interrupts the CPU. The interrupt handler, which is part of the device driver, dequeues the packet that was just sent. If any additional packets remain in the queue, the interrupt handler restarts the hardware to send the next packet. The interrupt handler then returns from the interrupt, allowing normal processing to continue.

Thus, from the point of view of the IP process, transmission of packets occurs automatically in the background. As long as packets remain on a queue, the hardware continues to transmit them. The hardware only needs to be started when IP deposits a packet on an empty queue.

Of course, each output queue has finite capacity and can become full if the system generates packets faster than the network hardware can transmit them. We assume that such cases are rare, but if they do occur, processes that generate packets must make a choice: discard the packet or block until the hardware finishes transmitting a packet and makes more space available.

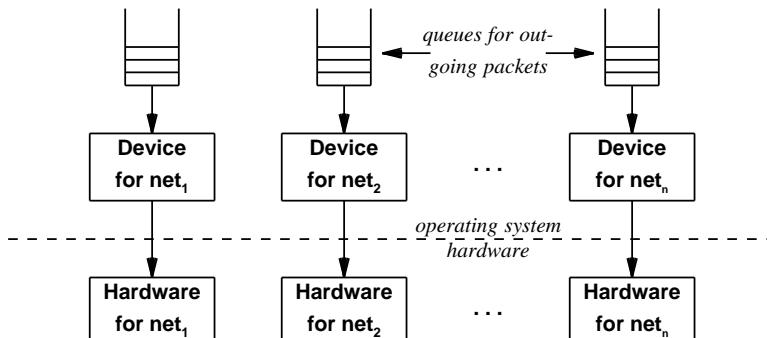


Figure 2.4 Network output and the queues that buffer output packets. Using queues isolates processing from network transmission

2.12 From TCP Through IP To Network Output

Like TCP input, TCP output is complex. Connections must be established, data must be placed in segments, and the segments must be retransmitted until acknowledgements arrive. Once a segment has been placed in a datagram, it can be passed to IP for routing and delivery. The software uses two TCP processes to handle the complexity. The first, called `tcpout`, handles most of the segmentation and data transmission details. The second, called `tcptimer`, manages a timer, schedules retransmission timeouts, and prompts `tcpout` when a segment must be retransmitted.

The `tcpout` process uses a port to synchronize input from multiple processes. Because TCP is stream oriented, allowing application programs to send a few bytes of data at a time, items in the port do not correspond to individual packets or segments. Instead, a process that emits data places the data in an output buffer and places a single message in the port informing TCP that more data has been written. The timer process deposits a message in the port whenever a timer expires and TCP needs to retransmit a segment. Thus, we can think of the port as a queue of events for TCP to process — each event can cause transmission or retransmission of a segment. Alternatively, an event may not cause an action (e.g., if data arrives while the receiver's window is closed). A later chapter reviews the exact details of events and TCP's responses.

Once TCP produces a datagram, it passes the datagram to IP for delivery. Although it is possible for two applications on a given machine to communicate, in most cases, the destination of a datagram is another machine. IP chooses a network interface over which the datagram must be sent and passes the datagram to the corresponding network output process. Figure 2.5 illustrates the path of outgoing TCP data.

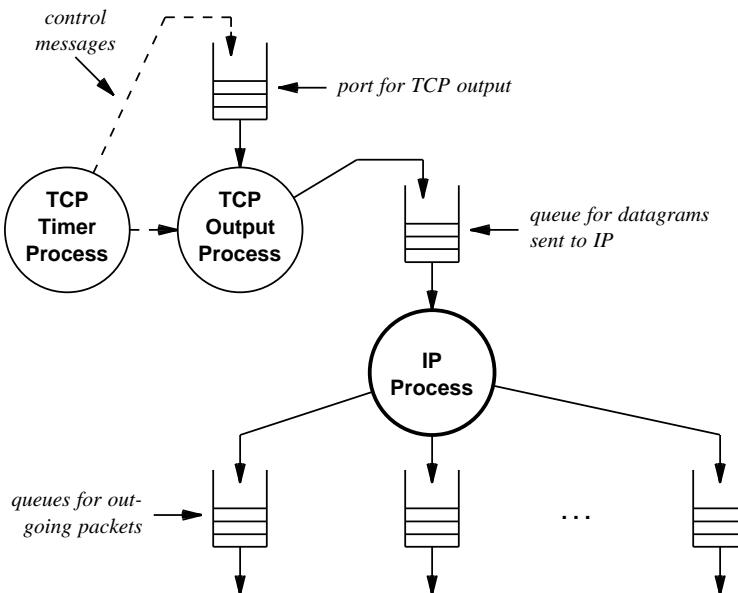


Figure 2.5 The TCP output and timer processes use the IP process to send data.

2.13 UDP Output

The path for outgoing UDP traffic is much simpler. Because UDP does not guarantee reliable delivery, the sending machine does not keep a copy of the datagram nor does it need to time retransmissions. Once the datagram has been created, it can be transmitted and the sender can discard its copy.

Any process that sends a UDP datagram must execute the UDP procedures needed to format it, as well as the procedures needed to encapsulate it and pass the resulting IP datagram to the IP process.

2.14 Summary

TCP/IP protocol software is part of the computer operating system. It uses the process abstraction to isolate pieces of protocol software, making each easier to design, understand, and modify. Each process executes independently, providing apparent parallelism. The system has a process for IP, TCP input, TCP output, and TCP timer management, as well as a process for each application program.

The operating system provides a semaphore mechanism that processes used to



synchronize their execution. The example code uses semaphores for mutual exclusion (i.e., to guarantee that only one process accesses a piece of code at a given time), and for producer-consumer relationships (i.e., when a set of processes produces data items that another set of processes consumes). The operating system also provides a port mechanism that allows processes to send messages to one another through a finite queue. The port mechanism uses semaphores to coordinate the processes that use the queue. If a process attempts to send a message to a port that is full, it will be blocked until another process extracts a message. Similarly, if a process attempts to extract a message from an empty port, it will be blocked until some other process deposits a message in the port.

Processes implementing protocols use both conventional queues and ports to pass packets among themselves. For example, the IP input process sends TCP segments to a port from which the TCP process extracts them, while the network input processes place arriving datagrams in a queue from which IP extracts them. When data is passed through conventional queues, the system must use message passing or semaphores to synchronize the actions of independent processes.

Figure 2.6 summarizes the flow of information between an application program and the network hardware during output. An application program, executing as a separate process, calls system routines to pass stream data to TCP or datagrams to UDP. For UDP output, the process executing the application program transfers into the operating system (through a system call), where it executes UDP procedures that allocate an IP datagram, fill in the appropriate destination address, encapsulate the UDP datagram in it, and send the IP datagram to the IP process for delivery.

For TCP output, the process executing an application program calls a system routine to transfer data across the operating system boundary and place it in a buffer. The application process then informs the TCP output process that new data is waiting to be sent. When the TCP output process executes, it divides the data stream into segments and encapsulates each segment in an IP datagram for delivery. Finally, the TCP output process enqueues the IP datagram on the port where IP will extract and send it.

Figure 2.7 summarizes the flow on input. The network device drivers enqueue all incoming packets that carry IP datagrams on queues for the IP process. IP extracts packets from the queues and demultiplexes them, delivering each packet to the appropriate high-level protocol software. When IP finds a datagram carrying UDP, it invokes UDP procedures that deposit the incoming datagram on the appropriate port, from which application programs read them. When IP finds a datagram carrying a TCP segment, it passes the datagram to a port from which the TCP input process extracts it. Note that the IP process is a central part of the design — a single IP process handles both input and output.

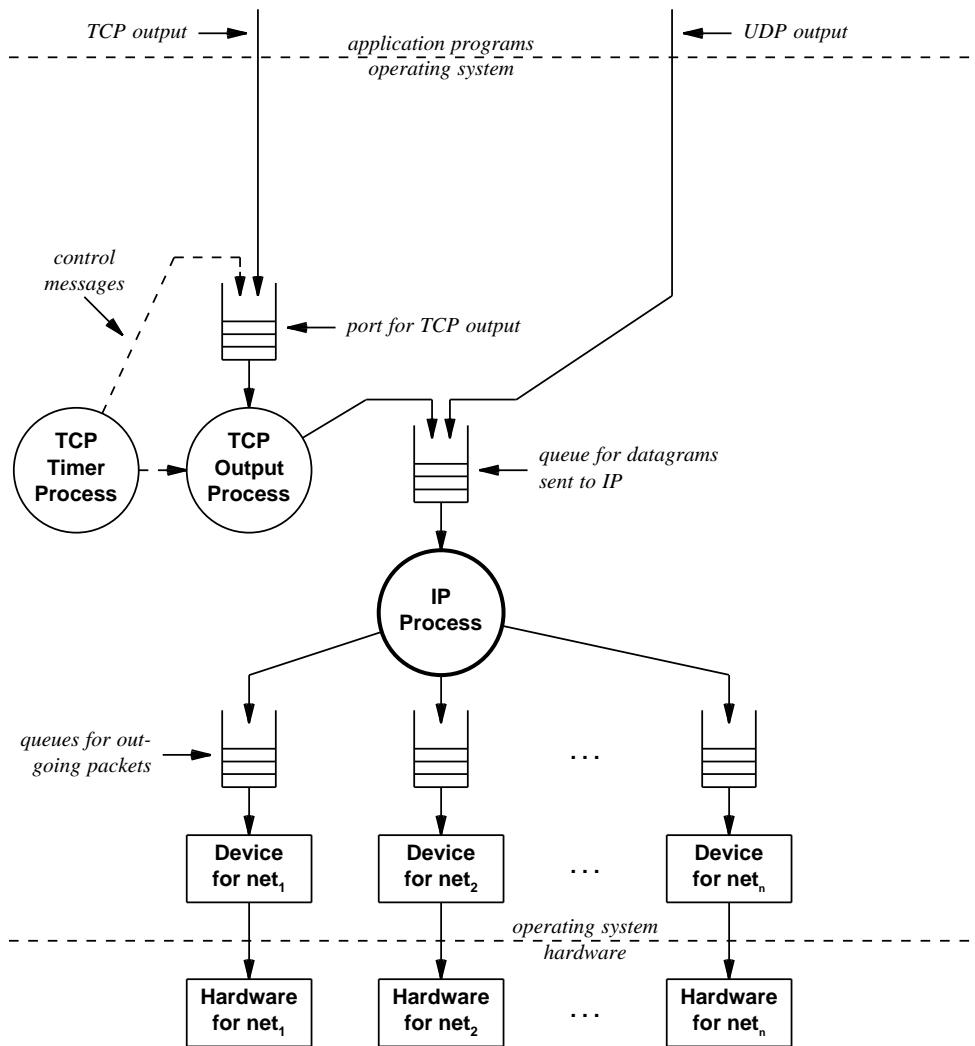


Figure 2.6 Output process structure showing the path of data between an application program and the network hardware. Output from the device queues is started at interrupt time. IP is a central part of the design — the software for input and output both share a single IP process.

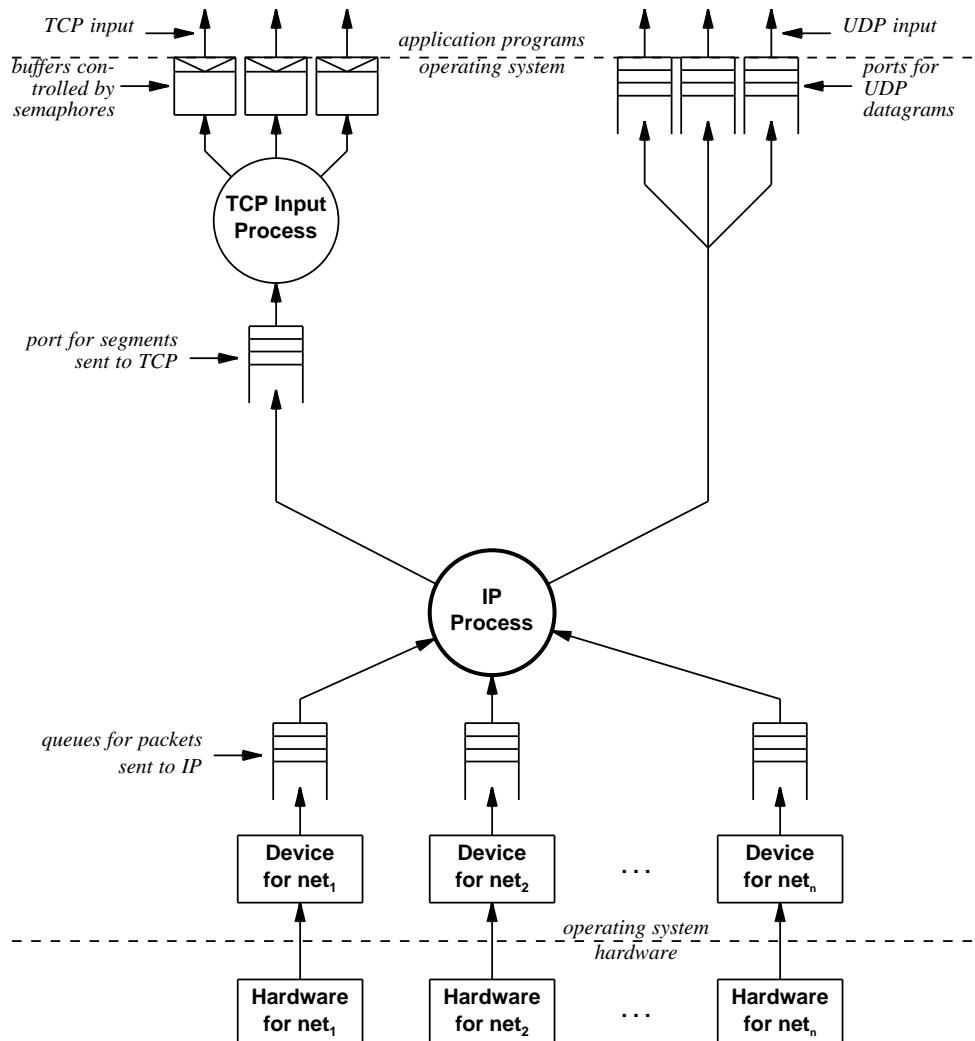


Figure 2.7 Input process structure showing the path of data between the network hardware and an application program. Input to the device queues occurs asynchronously with processing. IP is a central part of the design - the software for input and output share a single IP process.



2.15 FOR FURTHER STUDY

Our examples use the Xinu operating system. Comer [1984] provides a detailed description of the system, including the process and port abstractions. Comer [1987] shows how processes and ports can be used for simple protocols like UDP. Ritchie [1984] describes Stream I/O in System V UNIX, and Romkey [RFC 1055] contains the specification for SLIP.

2.16 EXERCISES

1. Why do protocol implementors try to minimize the number of processes that protocols use?
2. If the system described in this chapter executes on a computer in which the CPU is slow compared to the speed at which the network hardware can deliver packets, what will happen?
3. Read more about software interrupts and sketch the design of a protocol implementation that uses software interrupts instead of processes.
4. Read about the UNIX STREAMS facility and compare it to the process-oriented implementation described in this chapter. What are the advantages and disadvantages of each?
5. Compare two designs: one in which each application program that sends a UDP datagram executes all the UDP and IP code directly, and an alternative in which a separate UDP process accepts outgoing datagrams from all application programs. What are the two main advantages and disadvantages of each?
6. Consider a protocol software design that uses a large number of processes to handle packets. Assume that the system assigns a process to each datagram that arrives or each datagram that local applications generate. Also assume that the process follows the datagram through the protocol software until it can be sent or delivered. What is the chief advantage of such a design? The chief disadvantage?



3 Network Interface Layer

3.1 Introduction

TCP/IP Internet Protocol software is organized into five conceptual layers, as Figure 3.1 shows.

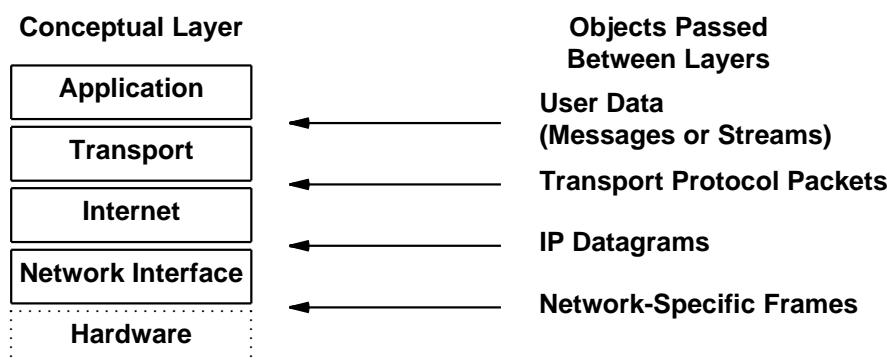


Figure 3.1 The conceptual organization of TCP/IP protocol software into layers.

This chapter examines the lowest layer, known as the network interface layer. Conceptually, the network interface layer controls the network hardware, performs mappings from IP addresses to hardware addresses, encapsulates and transmits outgoing packets, and accepts and demultiplexes incoming packets. This chapter shows how device driver and interface software can be organized to allow higher layers of protocol software to recognize and control multiple network hardware interfaces attached to a single machine. It also considers buffer management and packet demultiplexing. Chapter 4 discusses address resolution and encapsulation.

We have chosen to omit the network device driver code because it contains many low-level details that can only be understood completely by someone intimately familiar with the particular network hardware devices. Instead, this chapter concentrates on the elements of the network interface layer that are fundamental to an understanding of high-level protocol software.



3.2 The Network Interface Abstraction

Software in the network interface layer provides a network interface abstraction that is used throughout the rest of the system. The idea is simple:

The network interface abstraction defines the interface between protocol software in the operating system and the underlying hardware. It hides hardware details and allows protocol software to interact with a variety of network hardware using the same data structures.

3.2.1 Interface Structure

To achieve hardware independence, we define a data structure that holds all hardware-independent information about an interface (e.g., whether the hardware is up or down), and arrange protocol software to interact with the hardware primarily through this data structure. In our example code, the network interface consists of an array, nif, with one element for each hardware interface attached to the machine. Items in the interface array are known throughout the system by their index in the array. Thus, we can talk about "network interface number zero" or "the first network interface." File netif.h contains the pertinent declarations.

```
/* netif.h - NIGET */

#define NI_MAXHWA 14           /* max size of any hardware */
                             /* (physical) net address */
struct hwa {               /* a hardware address */
    int ha_len;             /* length of this address */
    char ha_addr[NI_MAXHWA]; /* actual bytes of the address */
};

#define NI_INQSZ 30           /* interface input queue size */
#define NETNLEN     30           /* length of network name */

#define NI_LOCAL 0            /* index of local interface */
#define NI_PRIMARY 1           /* index of primary interface */

#define NI_MADD 0              /* add multicast (ni_mcast) */
#define NI_MDEL 1              /* delete multicast (ni_mcast)

/* interface states */
```



```
#define NIS_UP 0x1
#define NIS_DOWN 0x2
#define NIS_TESTING 0x3

/* Definitions of network interface structure (one per interface) */

struct netif { /* info about one net interface */
    char ni_name[NETNLEN]; /* domain name of this interface*/
    char ni_state; /* interface states: NIS_ above */
    IPEndPoint ni_ip; /* IP address for this interface*/
    IPEndPoint ni_net; /* network IP address */
    IPEndPoint ni_subnet; /* subnetwork IP address */
    IPEndPoint ni_mask; /* IP subnet mask for interface */
    IPEndPoint ni_brc; /* IP broadcast address */
    IPEndPoint ni_nbrc; /* IP net broadcast address */
    int ni_mtu; /* max transfer unit (bytes) */
    int ni_hwtype; /* hardware type (for ARP) */
    struct hwa ni_hwa; /* hardware address of interface*/
    struct hwa ni_hwb; /* hardware broadcast address */
    int (*ni_mcast)(int op,int dev,Eaddr hwa,IPEndPoint ipa);
    Bool ni_invalid; /* is ni_ip valid? */
    Bool ni_nvalid; /* is ni_name valid? */
    Bool ni_svalid; /* is ni_subnet valid? */
    int ni_dev; /* the Xinu device descriptor */
    int ni_ipinq; /* IP input queue */
    int ni_outq; /* (device) output queue */
    /* Interface MIB */
    char *ni_descr; /* text description of hardware */
    int ni_mtype; /* MIB interface type */
    long ni_speed; /* bits per second */
    char ni_admstate; /* administrative status (NIS_*)*/
    long ni_lastchange; /* last state change (1/100 sec)*/
    long ni_ioctets; /* # of octets received */
    long ni_iucast; /* # of unicast received */
    long ni_inucast; /* # of non-unicast received */
    long ni_idiscard; /* # dropped - output queue full*/
    long ni_ierrors; /* # input packet errors */
    long ni_iunkproto; /* # in packets for unk. protos */
    long ni_ooctets; /* # of octets sent */
    long ni_oucast; /* # of unicast sent */
    long ni_onucast; /* # of non-unicast sent */
}
```



```
long ni_odiscard;          /* # output packets discarded      */
long ni_oerrors;           /* # output packet errors       */
long ni_oqlen;             /* output queue length        */
long     ni_maxreasmb;    /* max datagram can reassemble */
};

#define  NIGET(ifn)    ((struct ep *)deq(nif[ifn].ni_ipinq))

#define  NIF  Neth+Noth+1      /* # of interfaces, +1 for local*/

extern struct netif      nif[];
```

Structure `netif` defines the contents of each element in `nif`. Fields in `netif` define all the data items that protocol software needs as well as variables used to collect statistics. For example, field `ni_ip` contains the IP address assigned to the interface, and field `ni_mtu` contains the maximum transfer unit, the maximum size in octets of the data that can be sent in one packet on the network. Fields with names that end in `valid` contain Boolean variables that tell whether other fields are valid; initialization software sets them to `TRUE` once the fields have been assigned values. For example, `ni_invalid` is `TRUE` when `ni_ip` contains a valid IP address.

The device driver software places arriving datagrams for the IP process in a queue. Field `ni_ipinq` contains a pointer to that queue. To extract the next datagram, programs use the macro `NIGET`, which takes an interface number as an argument, dequeues the next packet from the interface queue, and returns a pointer to it.

3.2.2 Statistics About Use

Keeping statistics about an interface is important for debugging and for network management. For example, field `ni_iucast` holds a count of incoming unicast (non-broadcast) packets, while fields `ni_idiscard` and `ni_odiscard` count input and output packets that must be discarded due to errors.

The interface structure holds the physical (hardware) address in field `ni_hwa` and the physical (hardware) broadcast address in field `ni_hwb`. Because the length of a physical address depends on the underlying hardware, the software uses structure `hwa` to represent such addresses. Each hardware address begins with an integer length field followed by the address. Thus, high-level software can manipulate hardware addresses without understanding the hardware details.



3.3 Logical State Of An Interface

When debugging, managers often need to disable one or more of the interfaces on a given machine. Field `ni_state` provides a mechanism to control the logical state of an interface, independent of the underlying hardware. For example, a network manager can assign `ni_state` the value `NIS_DOWN` to stop input and output completely. Later, the manager can assign `ni_state` the value `NIS_UP` to restart I/O.

It is important to separate the logical state of an interface from the status of the physical hardware because it allows a manager freedom to control its operation. Of course, a manager can declare an interface down if the hardware fails. However, declaring an interface down does not disconnect the physical hardware, nor does it mean the hardware cannot work correctly. Instead, the declaration merely causes software to stop accepting incoming packets and to block outgoing packets. For example, a manager can declare an interface down when the network to which it attaches is overloaded.

3.4 Local Host Interface

In addition to routing datagrams among network interfaces, IP must also route datagrams to and from higher-level protocol software on the local computer. The interaction between IP and the local machine can either be implemented as:

- Explicit tests in the IP code, or
- An additional network interface for the local machine.

Our design uses a pseudo-network interface. The pseudo-network interface does not have associated device driver routines, nor does it correspond to real hardware, as Figure 3.2 shows. Instead, a datagram sent to the pseudo-net work will be delivered to protocol software on the local machine. Similarly, when protocol software generates an outgoing datagram, it sends the datagram to IP through the pseudo-network interface.

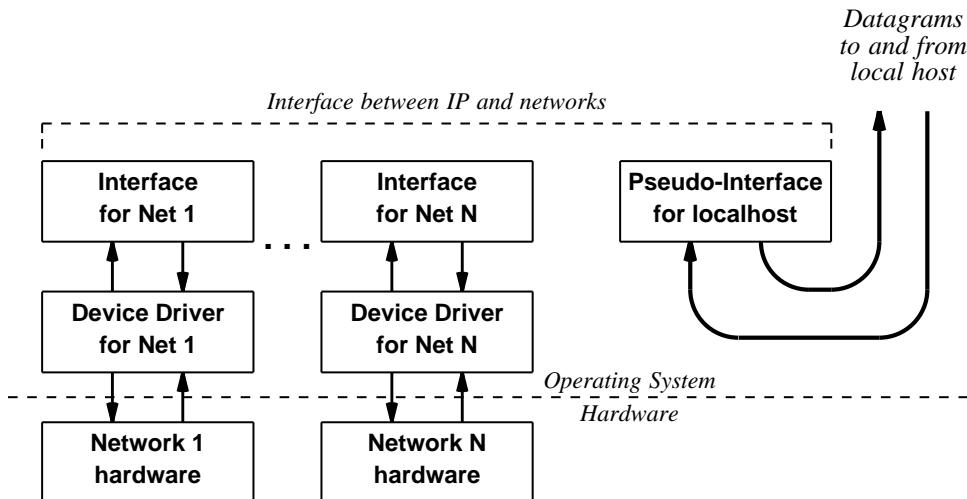


Figure 3.2 The pseudo-network interface used for communication with the local host.

Using a pseudo-net work for the local machine has several advantages. First, it eliminates special cases, simplifying the IP code. Second, it allows the local machine to be represented in the routing table exactly like other destinations. Third, it allows a network manager to interrogate the local interface as easily as other interfaces (e.g., to obtain a count of packets generated by local applications).

3.5 Buffer Management

Incoming packets must be placed in memory and passed to the appropriate protocol software for processing. Meanwhile, when an application program generates output, it must be stored in packets in memory and passed to a network hardware device for transmission. Thus, the network interface layer accepts outgoing data in memory and passes incoming data to higher-level protocol software in memory. The ultimate efficiency of protocol software depends on how it manages the memory used to hold packets. A good design allocates space quickly and avoids copying data as packets move between layers of protocol software.

Ideally, a system could make memory allocation efficient by dividing memory into fixed-size buffers, where each buffer is sufficient to hold a packet. In practice, however, choosing an optimum buffer size is complex for several reasons. First, a computer may connect to several networks, each of which has its own notion of maximum packet size. Furthermore, it should be possible to add connections to new types of networks without changing the system's buffer size. Second, IP may need to store datagrams larger than the underlying network packet sizes (e.g., to reassemble a large datagram). Third, an



application program may choose to send or receive arbitrary size messages.

3.5.1 Large Buffer Solution

It may seem that the ideal solution is to allocate buffers that are capable of storing the largest possible message or packet. However, because an IP datagram can be 64K octets long, allocating buffers large enough for arbitrary datagrams quickly expends all available memory on only a few buffers. Furthermore, small packets are the norm; large datagrams are rare. Thus, using large buffers can result in a situation where memory utilization remains low even though the system does not have sufficient buffers to accommodate traffic.

In practice, designers who use the large buffer approach usually choose an upper bound on the size of datagrams the system will handle, D, and make buffers large enough to hold a datagram of size D plus the physical network frame header. The choice of D is a tradeoff between allowing large datagrams and having sufficient buffers for the expected traffic. Thus, D depends on the expected size of buffer memory as well as the expected use of the system. Typically, timesharing systems choose values of D between 4K and 8K bytes.

3.5.2 Linked List Solutions (mbufs)

The chief alternative to large buffers uses linked lists of smaller buffers to handle arbitrary datagram sizes. In linked list designs, the individual buffers on the list can be fixed or variable size. Most systems allocate fixed size buffers because doing so prevents fragmentation and guarantees high memory utilization. Usually, each buffer is small (e.g., between 128 and 1K bytes), so many buffers must be linked together to represent a complete datagram. For example, Berkeley UNIX uses a linked structure known as the mbuf, where each mbuf is 128 bytes long. Individual mbufs need not be completely full; a short header specifies where data starts in the mbuf and how many bytes are present. Permitting buffers on the linked list to contain partial data has another advantage: it allows quick encapsulation without copying. When a layer of software receives a message from a higher layer, it allocates a new buffer, fills in its header information, and prepends the new buffer to the linked list that represents the message. Thus, additional bytes can be inserted at the front of a message without moving the existing data.

3.5.3 Our Example Solution

Our example system chooses a compromise between having large buffers sufficient to store arbitrary datagrams and linked lists of small buffers: it allocates many network buffers large enough to hold a single packet and allocates a few buffers large enough to hold large datagrams. The system performs packet-level I/O using the small buffers, and



only resorts to using large buffers when generating or reassembling large datagrams. This design was chosen because we expect most datagrams to be smaller than a conventional network MTU, but want to be able to reassemble larger datagrams as well. Thus, in most instances, it will be possible to pass an entire buffer to IP after reading a packet into it; the system will only need to copy data when reassembling a large datagram.

To make buffer processing uniform, our system uses a self-identifying buffer scheme provided by the operating system. To allocate a buffer, the system calls function getbuf and specifies whether it needs a large buffer or a small one. However, once the buffer has been allocated, only the pointer to it need be saved. To return the buffer to the free list, the system call freebuf, passing it a pointer to the buffer being released; freebuf deduces the size of the buffer automatically. The advantage of having the buffer be self-identifying is that protocol software can pass along a pointer to the buffer without having to remember whether it was allocated from the large or small group. Thus, outgoing packets can be kept in a simple list that identifies them by address. Once a device has transmitted a packet, the driver software can call freebuf to dispose of the buffer without having to know the buffer type.

3.5.4 Other Buffer Issues

DMA Memory. Hardware requirements often complicate buffer management. For example, some devices can only perform I/O in an area of memory reserved for direct memory access (DMA). In such systems, the operating system may choose to allocate two sets of buffers: those used by protocol software and those used for device transfer. The system must copy outgoing data from conventional buffers to the DMA area before transmission, and must copy incoming data from the DMA area to conventional buffers.

Gather-write, scatter-read. Some devices can transmit or receive packets in noncontiguous memory locations. On output, such devices accept a list of buffer addresses and lengths. They gather pieces of the packet from buffers on the list, and transmit the resulting sequence of bytes without requiring the system to assemble the packet in contiguous memory locations. The technique is known as gather-write. Similarly, the hardware may also support scatter-read in which the hardware deposits the packet in noncontiguous memory locations according to a list of buffer addresses specified by the device driver. Obviously, gather-write and scatter-read make linked buffer allocation easy and efficient because they allow the hardware to pick up pieces of the packet from the buffers on the linked list without requiring the processor to assemble a complete packet in memory. These techniques can also be used with fixed-size buffers because they allow the driver to encapsulate a datagram without copying it. To do so, the driver places the frame header in one part of memory and passes to the hardware the address of the header along with the address of the datagram, which becomes the data portion of the physical packet.



Page alignment. In a computer system that supports paged virtual memory, protocol software can attempt to allocate buffers on page boundaries, making it possible to pass the buffer to other processes by exchanging page table entries instead of copying. The technique is especially useful on machines with small page sizes (e.g., a Digital Equipment Corporation. VAX architecture, which has 512 byte pages), but it does not work well on computers with large page sizes (e.g., Sun Microsystems Sun 3 architecture. which has 8K byte pages). Furthermore, swapping page table entries improves efficiency most when moving data between the operating system and an application program. However, incoming packets contain a set of headers that make the exact offset of user data difficult or impossible to determine before a packet has been read. Therefore, few implementations try to align data on page boundaries.

3.6 Demultiplexing Incoming Packets

When a packet arrives, the device driver software in the network interface layer examines the packet type field to determine which protocol software will handle the packet. In general, designers take one of two basic approaches when building interface software: either they encode the demultiplexing in a procedure or use a table that maps the packet type to an appropriate procedure. Using code is often more efficient, but it means the software must be recompiled when new protocols are added. Using a table makes experimentation easier. In our implementation, we have chosen to demultiplex packets in a procedure. Procedure ni_in contains the demultiplexing code.

```
/* ni_in.c - ni_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#include <ospf.h>

int arp_in(struct netif *, struct ep *);
int rarp_in(struct netif *, struct ep *);
int ip_in(struct netif *, struct ep *);

/*
 * ni_in - network interface input function
 */
int
ni_in(struct netif *pni, struct ep *pep, unsigned len)
{
```



```
int rv;

switch (pep->ep_type) {
    case EPT_ARP: rv = arp_in(pni, pep); break;
    case EPT_RARP: rv = rarp_in(pni, pep); break;
    case EPT_IP:   rv = ip_in(pni, pep);  break;
    default:
        pni->ni_iunkproto++;
        freebuf(pep);
        rv = OK;
}
return rv;
}

/* ni_in.c - ni_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 * ni_in - network interface input function
 *-----
 */
int ni_in(pni, pep, len)
struct netif *pni;          /* the interface */
struct ep *pep;             /* the packet */
int len;                   /* length, in octets */
{
    int rv;

    pep->ep_ifn = pni - &nif[0]; /* record originating intf # */
    switch (pep->ep_type) {
        case EPT_ARP: rv = arp_in(pni, pep); break;
        case EPT_RARP: rv = rarp_in(pni, pep); break;
        case EPT_IP:
#ifdef DEBUG
{
            struct ip *pip = (struct ip *)pep->ep_data;
            if (pip->ip_proto == IPT_OSPF) {
                struct ospf *po = (struct ospf *)pip->ip_data;
                if (po->ospf_type == T_DATADESC) {

```



```
        kprintf("ni_in(pep %x, len %d)\n", pep, len);
        pdump(pep);
    }
}

#endif /* DEBUG */

        rv = ip_in(pni, pep);  break;

default:
    pni->ni_iunkproto++;
    freebuf(pep);
    return OK;
}

pni->ni_ioctets += len;
if (blkequ(pni->ni_hwa.ha_addr, pep->ep_dst, EP_ALEN))
    pni->ni_iucast++;
else
    pni->ni_inucast++;
return rv;
}
```

In our implementation, the device driver calls `ni_in` whenever an interrupt occurs to signal that a new packet has arrived. `Ni_in` handles four cases. If the packet carries an ARP message, RARP message, or IP datagram, `ni_in` passes the packet to the appropriate protocol routine and returns the result. Otherwise, it discards the packet by returning the buffer to the buffer pool. If the packet is accepted, `ni_in` increments appropriate counters to record the arrival of either a broadcast packet or a unicast packet. We will examine the procedures that `ni_in` calls in later chapters.

3.7 Summary

The network interface layer contains software that communicates between other protocol software and the network hardware devices. It includes buffer management routines, low-level device driver code, and contains many hardware-dependent details. Most important, it provides an abstraction known as the network interface that isolates higher-level protocols from the details of the hardware.

The `netif` structure defines the information kept for each network interface. It contains all information pertinent to the interface, making it possible for higher-level protocols to access the information without understanding the details of the specific hardware interface. Among the fields in `netif`, some contain information about the hardware (e.g., the hardware address), while others contain information used by protocol software (e.g., the subnet mask).



3.8 FOR FURTHER STUDY

Comer [1984] presents more details on the buffer pool scheme used in the example code. Comer [1987] describes an Ethernet hardware interface, shows the details of a device driver, and explains how the device driver code fits into an operating system. Leffler, McKusick, Karels, and Quarterman [1989] describes the use of mbufs in 4BSD UNIX

3.9 EXERCISES

1. Examine the MIB used with SMMP (RFC 1213). What statistics does it specify keeping for each network interface? Does the interface structure contain a field for each of them?
2. Read the BSD UNIX source code to see how mbufs are structured. Why does the header contain two pointers to other mbuf nodes?
3. Experiment with the 4BSD UNIX ping program (i.e., ICMP echo request/reply) to determine the largest datagram size that machines in your local environment can send and receive. How does it compare to the network MTU?
4. Find a hardware description of the Lance Ethernet interface device. Is it possible to enqueue multiple packets for transmission? If so, does this provide any advantages for the software designer?
5. Find a hardware architecture manual that describes DMA memory. How does a device driver use DMA memory for buffers?



4 Address Discovery And Binding (ARP)

4.1 Introduction

The previous chapter showed the organization of a network interface layer that contains device drivers for network hardware, as well as the associated software that sends outgoing packets and accepts incoming packets. Device drivers communicate directly with the network hardware and use only physical network addresses when transmitting and receiving packets.

This chapter examines ARP software that also resides in the network interface layer. ARP binds high-level, IP addresses to low-level, physical addresses. Address binding software forms a boundary between higher layers of protocol software, which use only IP addresses, and the lower layers of device driver software, which use only hardware addresses. Later chapters that discuss higher-layer protocols illustrate clearly how ARP insulates those layers from hardware addresses.

We said that address binding is part of the network interface layer, and our implementation reflects this idea. Although the ARP software maintains an address mapping that binds IP addresses to hardware addresses, higher layers of protocol software do not access the table directly. Instead, the ARP software encapsulates the mapping table and handles both table lookup as well as table update.

4.2 Conceptual Organization Of ARP Software

Conceptually, the ARP software can be divided into three parts: an output module, an input module, and a cache manager. When sending a datagram, the network interface software calls a procedure in the output module to bind a high-level protocol address (e.g., an IP address) to its corresponding hardware address. The output procedure returns a binding, which the network interface routines use to encapsulate and transmit the packet. The input module handles ARP packets that arrive from the network; it updates



the ARP cache by adding new bindings. The cache manager implements the cache replacement policy; it examines entries in the cache and removes them when they reach a specified age.

Before reviewing the procedures that implement ARP, we need to understand the basic design and the data structures used for the ARP address binding cache. The next sections discuss the design and the data structures used to implement it.

4.3 Example ARP Design

Although the ARP protocol seems simple, details can complicate the software. Many implementations fail to interpret the protocol specification correctly. Other implementations supply incorrect bindings because they eliminate cache timeout in an attempt to improve efficiency. It is important to consider the design of ARP software carefully and to include all aspects of the protocol.

Our example ARP software follows a few simple design rules:

- **Single Cache.** A single physical cache holds entries for all networks. Each entry in the cache contains a field that specifies the network from which the binding was obtained. The alternative is a multiple cache scheme that keeps a separate ARP cache for each network interface. The choice between using a single cache and multiple caches only makes a difference for gateways or multi-homed hosts that have multiple network connections.
- **Global Replacement Policy.** Our cache policy specifies that if a new binding must be added to the cache after it is already full, an existing item in the cache can be removed, independent of whether the new binding comes from the same network. The alternative is a local replacement policy in which a new binding can only replace a binding from the same network. In essence, a local replacement policy requires preallocation of cache space to each network interface and achieves the same effect as using separate caches.
- **Cache Timeout and Removal.** It is important to revalidate entries after they remain in the ARP cache for a fixed time. In our design, each cache entry has a time-to-live field associated with it. When an entry is added to the cache (or whenever an entry is validated), ARP software initializes the time-to-live field on the entry. As time proceeds, the cache manager decrements the value in the time-to-live field, and discards the entry when the value reaches zero. Removal from the cache is independent of the frequency with which the entry is used. Discarding an entry forces the ARP software to use the network to obtain a new binding from the destination machine. ARP does not automatically revalidate entries removed from the cache — the software waits



until an outgoing packet needs the binding before obtaining it again.

- **Multiple Queues of Waiting Packets.** Our design allows multiple outstanding packets to be enqueued waiting for an address to be resolved. Each entry in the ARP cache has a queue of outgoing packets destined for the address in that entry. When an ARP reply arrives that contains the needed hardware address, the software removes packets from the queue and transmits them.
- **Exclusive Access.** Our software disables interrupts and avoids context switching to guarantee that only one process accesses the ARP cache at any time. Procedures that operate on the cache (e.g., search it) require exclusive access, but do not contain code for mutual exclusion; responsibility to insure mutual exclusion falls to the caller.

In general, using a separate physical cache for each interface or using a local replacement policy provides some isolation between network interfaces. In the worst case, if the traffic on one network interface involves substantially more destinations than the traffic on others, bindings from the heavily-used interface may dominate the cache by replacing bindings from other networks. The symptom is the same as for any poorly-tuned cache: the cache remains 100% full at all times, but the probability of finding an entry in the cache is low. Our design assumes that the manager will monitor performance problems and allocate additional cache space when they occur.

While our design can behave poorly in the worst case, it provides more flexibility in the expected case because it allows cache allocation to vary dynamically with network load. If most of the traffic during a given time interval involves only a few networks, bindings for hosts on those networks will dominate the cache. If the traffic later shifts to a different set of networks, entries for hosts on the new networks will eventually dominate the cache,

4.4 Data Structures For The ARP Cache

File arp.h contains the declaration of the data structures for the ARP packet format, the internal data structures for the ARP cache, and the definitions for symbolic constants used throughout the ARP code.

```
/* arp.h - SHA, SPA, THA, TPA */

/* Internet Address Resolution Protocol (see RFCs 826, 920) */

#define AR_HARDWARE 1 /* Ethernet hardware type code */

/* Definitions of codes used in operation field of ARP packet */
```



```
#define AR_REQUEST 1 /* ARP request to resolve address */
#define AR_REPLY 2 /* reply to a resolve request */

#define RA_REQUEST 3 /* reverse ARP request (RARP packets) */
#define RA_REPLY 4 /* reply to a reverse request (RARP *) */

struct arp {
    u_short ar_hwtype; /* hardware type */
    u_short ar_prtype; /* protocol type */
    u_char ar_hwlen; /* hardware address length */
    u_char ar_prlen; /* protocol address length */
    u_short ar_op; /* ARP operation (see list above) */
    u_char ar_addrs[1]; /* sender and target hw & proto addrs */
/* char ar_sha[???]; - sender's physical hardware address */
/* char ar_spa[???]; - sender's protocol address (IP addr.) */
/* char ar_tha[???]; - target's physical hardware address */
/* char ar_tpa[???]; - target's protocol address (IP) */
};

#define SHA(p) (&p->ar_addrs[0])
#define SPA(p) (&p->ar_addrs[p->ar_hwlen])
#define THA(p) (&p->ar_addrs[p->ar_hwlen + p->ar_prlen])
#define TPA(p) (&p->ar_addrs[(p->ar_hwlen*2) + p->ar_prlen])

#define MAXHWALEN EP_ALEN /* Ethernet */
#define MAXPRALEN IP_ALEN /* IP */

#define ARP_TSIZE 50 /* ARP cache size */
#define ARP_QSIZE 10 /* ARP port queue size */

/* cache timeouts */

#define ARP_TIMEOUT 600 /* 10 minutes */
#define ARP_INF 0xffffffff /* "infinite" timeout value */
#define ARP_RESEND 1 /* resend if no reply in 1 sec */
#define ARP_MAXRETRY 4 /* give up after ~30 seconds */

struct arpentry { /* format of entry in ARP cache */
    short ae_state; /* state of this entry (see below) */
    short ae_hwtype; /* hardware type */
}
```



```
short      ae_prtype;      /* protocol type          */
char ae_hwlen; /* hardware address length      */
char ae_prlen; /* protocol address length      */
struct netif *ae_pni; /* pointer to interface structure */
int ae_queue; /* queue of packets for this address */
int ae_attempts; /* number of retries so far      */
int ae_ttl; /* time to live      */
u_char    ae_hwa[MAXHWALEN]; /* Hardware address      */
u_char    ae_pra[MAXPRALEN]; /* Protocol address      */
};

#define AS_FREE      0      /* Entry is unused (initial value) */
#define AS_PENDING    1      /* Entry is used but incomplete */
#define AS_RESOLVED   2      /* Entry has been resolved */

/* RARP variables */

extern int    rarppid; /* id of process waiting for RARP reply */
extern int    rarpsem; /* semaphore for access to RARP service */

/* ARP variables */

extern struct arpentry arptable[ARP_TSIZE];
```

Array arptable forms the global ARP cache. Each entry in the array corresponds to a single binding between a protocol (IP) address (field ae_pra), and a hardware address (ae_hwa). Field ae_state gives the state of the entry, which must be one of AS_FREE (entry is currently unused), AS_PENDING (entry is being used but binding has not yet been found), or AS_RESOLVED (entry is being used and the binding is correct). Each entry also contains fields that give the hardware and protocol types (ae_hwtype and ae_prtype), and the hardware and protocol address lengths (ae_hwlen and ae_prlen). Field ae_pni points to the network interface structure corresponding to the network from which the binding was obtained. For entries that have not yet been resolved, field dequeue points to a queue of packets that can be sent when an answer arrives. For entries in state AS_PENDING, field ae_attempts specifies the number of times a request for this entry has been broadcast. Finally, field ae_ttl specifies the time (in seconds) an entry can remain in the cache before the timer expires and it must be removed.

Structure arp defines the format of an ARP packet. Fields ar_hwtype and ar_prtype specify the hardware and protocol types, and fields ar_hwlen and ar_prlen contain integers that specify the sizes of the hardware address and the protocol address, respectively. Field ar_op specifies whether the packet contains a request or a reply.



Because the size of addresses carried in an ARP packet depends on the type of hardware and type of protocol address being mapped, the arp structure cannot specify the size of all fields in a packet. Instead, the structure only specifies the fixed-size fields at the beginning of the packet, and uses field name ar_addrs to mark the remainder of the packet. Conceptually, the bytes starting at field ar_addrs comprise four fields: the hardware and protocol address pairs for the sender and target, as the comments in the declaration illustrate. Because the size of each address field can be determined from information in the fixed fields of the header, the location of each address field can be computed efficiently. In-line functions SHA, SPA, THA, and TPA perform the computations. Each function takes a single argument that gives the address of an ARP packet, and returns the location of the field in that packet that corresponds to the function name.

4.5 ARP Output Processing

4.5.1 Searching The ARP Cache

The network interface code that handles output uses ARP to resolve IP addresses into the corresponding hardware addresses. In particular, the network output process calls procedure arpfnd to search the ARP cache and find an entry that matches a given protocol address.

```
/* arpfnd.c - sendarp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * arpfnd - find an ARP entry given a protocol address and interface
 */
struct arpentry *arpfnd(pra, prtype, pni)
char      *pra;
int       prtype;
struct netif  *pni;
{
    struct arpentry   *pae;
    int      i;

    for (i=0; i<ARP_TSIZE; ++i) {
        pae = &arptable[i];
```



```
    if (pae->ae_state == AS_FREE)
        continue;

    if (pae->ae_prtype == prtype &&
        pae->ae_pni == pni &&
        blkequ(pae->ae_pra, pra, pae->ae_prlen))
        return pae;

    }

    return 0;
}
```

Argument pra points to a high-level (protocol) address that must be resolved, argument prtype gives the type of the address (using the standard ARP values for protocol types), and argument pni points to a network interface structure. Arpfind searches the ARP cache sequentially until it finds an entry that matches the specified address. It returns a pointer to the entry.

Recall that our design places all ARP bindings in a single table. For technologies like Ethernet, where hardware addresses are globally unique, a single table does not present a problem. However, some technologies allow reuse of hardware addresses on given hardware address (e.g., address 5) in its cache. Argument pni insures that arpfind will select bindings that correspond to the correct network interface. Conceptually, our implementation uses the combination of a network interface number and hardware address to uniquely identify an entry in the table.

4.5.2 Broadcasting An ARP Request

Once an ARP cache entry has been allocated for a given IP address, the network interface software calls procedure arpsend to format and broadcast an ARP request for the corresponding hardware address.

```
/* arpsend.c - arpsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * arpsend - broadcast an ARP request
 *   N.B. Assumes interrupts disabled
 */
int arpsend(pae)
    struct    arpentry *pae;
```



```
{  
    struct netif *pni = pae->ae_pni;  
    struct ep *pep;  
    struct arp *parp;  
    int arplen;  
  
    pep = (struct ep *) getbuf(Net.netpool);  
    if (pep == SYSERR)  
        return SYSERR;  
    blkcopy(pep->ep_dst, pni->ni_hwb.ha_addr, pae->ae_hwlen);  
    pep->ep_type = EPT_ARP;  
    parp = (struct arp *) pep->ep_data;  
    parp->ar_hwtype = hs2net(pae->ae_hwtype);  
    parp->ar_prtype = hs2net(pae->ae_prtype);  
    parp->ar_hwlen = pae->ae_hwlen;  
    parp->ar_prlen = pae->ae_prlen;  
    parp->ar_op = hs2net(AR_REQUEST);  
    blkcopy(SHA(parp), pni->ni_hwa.ha_addr, pae->ae_hwlen);  
    blkcopy(SPA(parp), pni->ni_ip, pae->ae_prlen);  
    bzero(THA(parp), pae->ae_hwlen);  
    blkcopy(TPA(parp), pae->ae_pra, pae->ae_prlen);  
    arrlen = sizeof(struct arp) + 2*(parp->ar_hwlen + parp->ar_prlen);  
    write(pni->ni_dev, pep, arrlen);  
    return OK;  
}
```

Arpsend takes a pointer to an entry in the cache as an argument, forms an ARP request for the IP address in that entry, and transmits the request. The code is much simpler than it appears. After allocating a buffer to hold the packed, arpsend fills in each field, obtaining most of the needed information from the arp cache entry given by argument pae. It uses the hardware broadcast for the packet destination address and specifies that the packet is an ARP request (AR_REQUEST). After the hardware and protocol address length fields have been assigned, arpsend can use in-line procedures SHA, SPA, THA, and TPA to compute the locations in the ARP packet of the variable-length address fields.

After arpsend creates the ARP request packet, it invokes system call write to send it.

4.5.3 Output Procedure

Procedure netwrite accepts packets for transmission on a given network interface.



```
/* netwrite.c - netwrite */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <q.h>

struct    arpentry *arpalloc(), *arpfind();

/*-----
 * netwrite - write a packet on an interface, using ARP if needed
 *-----
 */

int netwrite(pni, pep, len)
struct    netif      *pni;
struct    ep      *pep;
int        len;
{
    struct    arpentry *pae;
    STATWORD      ps;
    int            i;

    if (pni->ni_state != NIS_UP) {
        freebuf(pep);
        return SYSERR;
    }
    pep->ep_len = len;
    if (pni == &nif[NI_LOCAL])
        return local_out(pep);
    else if (isbrc(pep->ep_nexthop)) {
        blkcopy(pep->ep_dst, pni->ni_hwb.ha_addr, EP_ALEN);
        write(pni->ni_dev, pep, len);
        return OK;
    }
    /* else, look up the protocol address... */

    disable(ps);
    pae = arpfind(pep->ep_nexthop, pep->ep_type, pni);
    if (pae && pae->ae_state == AS_RESOLVED) {
        blkcopy(pep->ep_dst, pae->ae_hwa, pae->ae_hwlen);
        restore(ps);
    }
}
```



```
        return write(pni->ni_dev, pep, len);
    }

    if (IP_CLASSD(pep->ep_nexthop)) {
        restore(ps);
        return SYSERR;
    }

    if (pae == 0) {
        pae = arpalloc();
        pae->ae_hwtype = AR_HARDWARE;
        pae->ae_prtype = EPT_IP;
        pae->ae_hwlen = EP_ALEN;
        pae->ae_prlen = IP_ALEN;
        pae->ae_pni = pni;
        pae->ae_queue = EMPTY;
        blkcopy(pae->ae_pra, pep->ep_nexthop, pae->ae_prlen);
        pae->ae_attempts = 0;
        pae->ae_ttl = ARP_RESEND;
        arpsend(pae);
    }

    if (pae->ae_queue == EMPTY)
        pae->ae_queue = newq(ARP_QSIZE, QF_NOWAIT);
    if (enq(pae->ae_queue, pep, 0) < 0)
        freebuf(pep);
    restore(ps);
    return OK;
}
```

Netwrite calls arpfind to look up an entry in the cache for the destination address. If the entry has been resolved, netwrite copies the hardware address into the packet and calls write to transmit the packet. If the entry has not been resolved and is not pending, netwrite calls arpalloc to allocate an ARP request. It then fills in fields in the ARP entry, and calls arpsend to broadcast the request.

Because netwrite must return to its caller without delay, it leaves packets awaiting address resolution on the queue of packets associated with the ARP cache entry for that address. It first checks to see if a queue exists. If one is needed, it calls newq to create a queue. Finally, netwrite calls enq to enqueue the packet for transmission later, after the address has been resolved. Each output queue has a finite size. If the queue is full when netwrite needs to enqueue a packet, netwrite discards the packet.



4.6 ARP Input Processing

4.6.1 Adding Resolved Entries To The Table

ARP input processing uses two utility procedures, arpadd and arpqsnd. Arpadd takes information from an ARP packet that has arrived over the network, allocates an entry in the cache, and fills the entry with information from the packet. Because it fills in both the hardware and protocol address fields, arpadd assigns AS_RESOLVED to the entry's state field. It also assigns the entry's time-to-live field and the maximum timeout value, ARP_TIMEOUT.

```
/* arpadd.c - arpadd */

#include <conf.h>
#include <kernel.h>
#include <network.h>

struct arpentry *arpalloc();

/*
 * arpadd - Add a RESOLVED entry to the ARP cache
 * N.B. Assumes interrupts disabled
 */
struct arpentry *arpadd(pni, parp)
struct netif *pni;
struct arp *parp;
{
    struct arpentry *pae;

    pae = arpalloc();

    pae->ae_hwtype = parp->ar_hwtype;
    pae->ae_prtype = parp->ar_prtype;
    pae->ae_hwlen = parp->ar_hwlen;
    pae->ae_prlen = parp->ar_prlen;
    pae->ae_pni = pni;
    pae->ae_queue = EMPTY;
    blkcopy(pae->ae_hwa, SHA(parp), parp->ar_hwlen);
    blkcopy(pae->ae_pra, SPA(parp), parp->ar_prlen);
    pae->ae_ttl = ARP_TIMEOUT;
    pae->ae_state = AS_RESOLVED;
    return pae;
}
```



}

4.6.2 Sending Waiting Packets

We have seen that the ARP output procedures enqueue packets that are waiting for address resolution. When an ARP packet arrives that contains information needed to resolve an entry, the ARP input procedure calls arpqsnd to transmit the waiting packets.

```
/* arpqsnd.c - arpqsnd */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 * arpqsnd - write packets queued waiting for an ARP resolution
 *-----
 */
void arpqsnd(pae)
struct arpentry *pae;
{
    struct ep *pep;
    struct netif *pni;

    if (pae->ae_queue == EMPTY)
        return;

    pni = pae->ae_pni;
    while (pep = (struct ep *)deq(pae->ae_queue))
        netwrite(pni, pep, pep->ep_len);
    freeq(pae->ae_queue);
    pae->ae_queue = EMPTY;
}
```

Arpqsnd does not transmit waiting packets directly. Instead, it iterates through the queue extracting packets and calling netwite to place each packet on the network output queue (where the network device will extract and transmit it). Once it has removed all packets, arpqsnd calls freeq to deallocate the queue itself.

4.6.3 ARP Input Procedure

As we have seen, when an ARP packet arrives, the network device driver passes it



to procedure arp_in for processing.

```
/* arp_in.c - arp_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>

struct    arpentry *arpfind(), *arpadd();

/*-----
 * arp_in - handle ARP packet coming in from Ethernet network
 * N.B. - Called by ni_in-- SHOULD NOT BLOCK
 *-----
 */
int arp_in(pni, pep)
struct    netif      *pni;
struct    ep       *pep;
{
    struct    arp      *parp = (struct arp *)pep->ep_data;
    struct    arpentry *pae;
    int         arplen;

    parp->ar_hwtype = net2hs(parp->ar_hwtype);
    parp->ar_prtype = net2hs(parp->ar_prtype);
    parp->ar_op = net2hs(parp->ar_op);

    if (parp->ar_hwtype != pni->ni_hwtype ||
        parp->ar_prtype != EPT_IP) {
        freebuf(pep);
        return OK;
    }

    if (pae = arpfind(SPA(parp), parp->ar_prtype, pni)) {
        blkcopy(pae->ae_hwa, SHA(parp), pae->ae_hwlen);
        pae->ae_ttl = ARP_TIMEOUT;
    }
    if (!blkque(TPA(parp), pni->ni_ip, IP_ALEN)) {
        freebuf(pep);
        return OK;
    }
    if (pae == 0)
```



```
    pae = arpadd(pni, parp);
    if (pae->ae_state == AS_PENDING) {
        pae->ae_state = AS_RESOLVED;
        arpqsend(pae);
    }
    if (parp->ar_op == AR_REQUEST) {
        parp->ar_op = AR_REPLY;
        blkcopy(TPA(parp), SPA(parp), parp->ar_prlen);
        blkcopy(THA(parp), SHA(parp), parp->ar_hwlen);
        blkcopy(pep->ep_dst, THA(parp), EP_ALEN);
        blkcopy(SHA(parp), pni->ni_hwa.ha_addr,
               pni->ni_hwa.ha_len);
        blkcopy(SPA(parp), pni->ni_ip, IP_ALEN);

        parp->ar_hwtype = hs2net(parp->ar_hwtype);
        parp->ar_prtype = hs2net(parp->ar_prtype);
        parp->ar_op = hs2net(parp->ar_op);

        arplen = sizeof(struct arp) +
                  2*(parp->ar_prlen + parp->ar_hwlen);

        write(pni->ni_dev, pep, arplen);
    } else
        freebuf(pep);
    return OK;
}
```

The protocol standard specifies that ARP should discard any messages that specify a high-level protocol the machine does not recognize. Thus, our implementation of arp_in only recognizes ARP packets that specify a protocol address type IP and a hardware address type that matches the hardware type of the network interface over which the packet arrives. If packets arrive containing other address types, ARP discards them.

When processing a valid packet, arp_in calls arpfnd to search the ARP cache for an entry that matches the sender's IP address. The protocol specifies that a receiver should first use incoming requests to satisfy pending entries (i.e., it should use the sender's addresses to update its cache). Thus, if a matching entry is found, arp_in updates the hardware address from the sender's hardware address field in the packet and sets the timeout field of the entry to ARP_TIMEOUT.

The protocol also specifies that if the incoming packet contains a request directed at



the receiver, the receiver must add the sender's address to its cache (even if the receiver did not have an entry pending for that address). Thus, arp_in checks to see if the target IP address matches the local machine's IP address. If it does, arp_in calls arpadd to insert it. After inserting an entry in the cache, arp_in checks to see whether the address was pending resolution. If so, it calls arpqsend to transmit the queue of waiting packets.

Finally, arp_in checks to see if the packet contained a request. If it does, arp_in forms a reply by interchanging the target and sender address fields, supplying the requested hardware address, and changing the operation from AR_REQUEST to AR_REPLY. It transmits the reply directly.

4.7 ARP Cache Management

So far, we have focused on input and output processing. However, management of the ARP cache requires coordination between the input and output software. It also requires periodic computation independent of either input or output. The next sections explain the cache policy and show how the software enforces it.

4.7.1 Allocating A Cache Entry

If a process (e.g., the IP process) needs to send a datagram but no entry is present in the ARP cache for the destination IP address, IP must create a new cache entry, broadcast a request, and enqueue the packet awaiting transmission. Procedure arpalloc chooses an entry in the ARP cache that will be used for a new binding.

```
/* arpalloc.c - arpalloc */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

void arpdq();

/*
 * arpalloc - allocate an entry in the ARP table
 *   N.B. Assumes interrupts DISABLED
 */
struct arpentry *arpalloc()
{
    static int aenext = 0;
    struct arpentry *pae;
```



```
int i;

for (i=0; i<ARP_TSIZE; ++i) {
    if (arphtable[aenext].ae_state == AS_FREE)
        break;
    aenext = (aenext + 1) % ARP_TSIZE;
}

pae = & arphtable[aenext];
aenext = (aenext + 1) % ARP_TSIZE;

if (pae->ae_state == AS_PENDING && pae->ae_queue >= 0)
    arpdq(pae);
pae->ae_state = AS_PENDING;
return pae;
}
```

Arpalloc implement the cache replacement policy because it must decide which existing entry to eliminate from a full cache when finding space for a new entry. We have chosen a simple replacement policy.

When allocating space for a new addition to the ARP cache, choose an unused entry in the table if one exists. Otherwise, delete entries in a round-robin fashion.

That is, each time it selects an entry to delete, the cache manager moves to the next entry. It cycles around the table completely before returning to an entry. Thus, once it deletes an entry and reuses it for a new binding, the cache manager will leave that binding in place until it has been forced to delete and replace all other bindings.

In considering an ARP cache policy, it is important to remember that a full cache is always undesirable because it means the system is operating at saturation. If a datagram transmission causes the system to insert a new binding in the cache, the system must delete an existing binding. When the old, deleted binding is needed again, ARP will delete yet another binding and broadcast a request. In the worst case, ARP will broadcast a request each time it needs to deliver a datagram. We assume that a system manager will monitor and detect such situations, and then reconfigure the system with a larger cache. Thus, preemption of existing entries will seldom occur, so our simple round-robin policy works well in practice.

To implement the preemption policy, arpalloc maintains a static integer, aenext. The for-loop in arpalloc searches the entire table, starting at the entry with index aenext, wrapping around to the beginning of the table, and finishing back at position aenext. The



search stops immediately if an unused entry is found. If no unused space remains in the cache, arpalloc removes the old entry with index aenext. Finally, arpalloc increments aenext so the neat search will start beyond the newly allocated entry.

4.7.2 Periodic Cache Maintenance

Our design arranges to have an independent timer process execute procedure arptimer periodically.

```
/* arptimer.c - arptimer */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * arptimer - Iterate through ARP cache, aging (possibly removing) entries
 */
void arptimer(gran)
int gran;           /* time since last iteration */
{
    struct arpentry *pae;
    STATWORD ps;
    int i;

    disable(ps); /* mutex */

    for (i=0; i<ARP_TSIZE; ++i) {
        if ((pae = &arphtable[i])->ae_state == AS_FREE)
            continue;
        if ((pae = ae_ttl == ARP_INF)
            continue; /* don't time out permanent entry */
        if ((pae->ae_ttl -= gran) <= 0)
            if (pae->ae_state == AS_RESOLVED)
                pae->ae_state = AS_FREE;
            else if (++pae->ae_attempts > ARP_MAXRETRY) {
                pae->ae_state = AS_FREE;
                arpdq(pae);
            } else {
                pae->ae_ttl = ARP_RESEND;
                arpsend(pae);
            }
    }
}
```



```
        }
        restore(ps);
    }
```

When it calls arptimer, the timer process passes an argument that specifies the time elapsed since the previous call. Arptimer uses the elapsed time to "age" entries in the cache. It iterates through each entry and decrements the time-to-live field in the entry by gran, where gran is the number of seconds since the last iteration. If the time-to-live becomes zero or negative, arptimer removes the entry from the cache. Removing a resolved entry merely means changing the state to AS_FREE, which allows arpalloc to use the entry the next time it needs one. If the time-to-live expires on an entry that is pending resolution, arptimer examines field ae_attempts to see whether the request has been rebroadcast ARP_MAXRETRY times. If not, arptimer calls arpsend to broadcast the request again. If the request has already been rebroadcast ARP_MAXRETRY times, arptimer deallocates the queue of waiting packets and removes the entry.

4.7.3 Deallocating Queued Packets

If the ARP cache is full, the existing entry arpalloc selects to remove may have a queue of outgoing packets associated with it. If so, arpalloc calls arpdq to remove packets from the list and discard them.

```
/* arpdq.c - arpdq */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * arpdq - destroy an arp queue that has expired
 */
void arpdq(pae)
struct    arpentry *pae;
{
    struct    ep    *pep;
    struct    ip    *pip;

    if (pae->ae_queue < 0)      /* nothing to do */
        return;

    while (pep = (struct ep *)deq(pae->ae_queue)) {
```



```
    if (gateway && pae->ae_prtype == EPT_IP) {
        pip = (struct ip *)pep->ep_data;
        icmp(ICK_DESTUR, ICC_HOSTUR, pip->ip_src, pep);
    } else
        freebuf(pep);
}
freeq(pae->ae_queue);
pae->ae_queue = EMPTY;
}
```

Arpdq iterates through the queue of packets associated with an ARP cache entry and discards them. If the packet is an IP datagram and the machine is a gateway, arpdq calls procedure icmp to generate an ICMP destination unreachable message for the datagram it discards. Finally, arpdq calls freeq to release the queue itself.

4.8 ARP Initialization

The system calls procedure arpinit once, at system startup. Arpinit creates rarpsem, the mutual exclusion semaphore used with RARP, and assigns state AS_FREE to all entries in the ARP cache. In addition, arpinit initializes a few data items for the related RARP protocol; these are irrelevant to the code in this chapter. Note that arpinit does not initialize the timer process or set up calls to arptimer. These details are handled separately because our design uses a single timer process for many protocols.

```
/* arpinit.c - arpinit */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*
 * arpinit - initialize data structures for ARP processing
 */
void arpinit()
{
    int i;

    rarpsem = screate(1);
    rarppid = BADPID;
```



```
    for (i=0; i<ARP_TSIZE; ++i)
        arptable[i].ae_state = AS_FREE;
}

int rarpsem;
int rarppid;

struct arpentry arptable[ARP_TSIZE];
```

4.9 ARP Configuration Parameters

When, building ARP software, the programmer configures the system by choosing values for parameters such as:

- Size of the ARP cache
- Timeout interval the sender waits for an ARP response
- Number of times a sender retries a request
- Time interval between retries
- Timeout (time-to-live) for a cache entry
- Size of packet retransmission queue

Typical designs use symbolic constants for parameters such as cache size, allowing the system manager to change the configuration for specific installations. For installations in which managers need more control, utility programs can be written that allow a manager to make changes at run time. For example, in some software it is possible for a manager to examine the ARP cache, delete an entry, or change values (e.g., the time-to-live field). However, some parameters cannot be changed easily. For example, many programmers choose between fixed retransmission delays or exponential backoff and embed their choice in the code itself, as in our example.

4.10 Summary

Our implementation of ARP uses a single, global cache to hold bindings obtained from all networks, it permits multiple packets to be enqueued waiting for an address to be resolved, and uses an independent timer to age cache entries. Eventually, entries timeout. If the cache is completely full when a new entry must be inserted, an old entry must be discarded. Our design uses a round-robin replacement policy, implemented with a global pointer that moves to the next cache entry each time one is taken. The example code shows the declarations of data structures that comprise the cache and the procedures that operate on them.



4.11 FOR FURTHER STUDY

Plummer [RFC 826] defines the ARP standard, while Clark [RFC 814] discusses addresses and bindings in general. Parr [RFC 1029] considers fault tolerant address resolution.

4.12 EXERCISES

1. What network hardware uses ARP?
2. Sketch the design of address binding software for a network interface that does not use ARP.
3. What is the chief disadvantage of using a single table to hold the ARP cache in a gateway? What is the chief advantage?
4. Suppose a site decided to use ARP on its proNET-10 ring networks (even though it is possible to bind proNET-10 addresses without ARP). Would our implementation operate correctly on a gateway that connected multiple rings?
Hint: proNET-10 addresses are only unique within a given network.
5. The Ethernet hardware specification enforces a minimum packet size of 60 octets. Examine the Ethernet device driver software in an operating system. How does the driver send an ARP packet, which is shorter than 60 octets?
6. Would users perceive any difference in performance if the ARP software did not allow multiple packets to be enqueued for a pending ARP binding?
7. How does one choose reasonable values for ARP_MAXRETRY, ARP_TIMEOUT, and the granularity of aging?
8. ARP is especially susceptible to "spoofing" because an arbitrary machine can answer an ARP broadcast. Revise the example software by adding checks that detect when (a) two or more machines answer a request for a given IP address, (b) a machine receives an ARP binding for its own IP address, and (c) a single machine answers requests for multiple IP addresses.
9. As an alternative solution to the spoofing problem mentioned in the previous exercise, modify the example software by adding a check to insure that the hardware address reported in field SHA of the ARP packet matches the hardware address in the source field of the hardware frame. What are the advantages and disadvantages of each approach?
10. Read about addressing for bridged token ring networks. Should ARP use the local ring broadcast address or the all ring broadcast address? Why?



5 IP: Global Software Organization

5.1 Introduction

This chapter considers the organization of software that implements the Internet Protocol (IP). While the functionality IP provides may seem simple, intricacies make implementing the software complicated and subtleties make it difficult to insure correctness. To help explain IP without becoming overwhelmed with all the parts at once, we will consider the implementation in three chapters. This chapter presents data structures and describes the overall software organization. It discusses the conceptual operation of IP software and the flow of datagrams through the IP layer. Later chapters, which provide details on routing and error handling, show how various pieces of IP software use these data structures.

5.2 The Central Switch

Conceptually, IP is a central switching point in the protocol software. It accepts incoming datagrams from the network interface software as well as outgoing datagrams that higher-level protocols generate. After routing a datagram, IP either sends it to one of the network interfaces or to a higher-level protocol on the local machine.

In a host, it seems natural to think of IP software in two distinct parts: one that handles input and one that handles output. The input part uses the PROTO field of the IP header to decide which higher-level protocol module should receive an incoming datagram. The output part uses the local routing table to choose a next hop for outgoing datagrams.

Despite its intuitive appeal, separating IP input and output makes the interaction between IP and the higher-level protocol software awkward. In addition, IP software must work in gateways, where routing is more complex than in hosts. In particular, gateway software cannot easily be partitioned into input and output parts because a



gateway must forward an arriving datagram on to its next hop. Thus, IP may generate output while handling an incoming datagram. A gateway must also generate ICMP error messages when arriving datagrams cause errors, which further blurs the distinction between input and output. In the discussion that follows, we will concentrate on gateways and treat hosts as a special case.

5.3 IP Software Design

To keep the IP software simple and uniform, our implementation uses three main organizational techniques:

- **Uniform Input Queue and Uniform Routing.** The IP process uses the same input queue style for all datagrams it must handle, independent of whether they arrive from the network or are generated by the local machine. IP extracts each datagram from a queue and routes it without regard to the datagram's source. A uniform input structure results in simplicity: IP does not need a special case in the code for locally generated datagrams. Furthermore, because IP uses a single routing algorithm to route all datagrams, humans can easily understand the route a datagram will take.
- **Independent IP Process.** The IP software executes as a single, self-contained process. Using a process for IP keeps the software easy to understand and modify. It allows us to create IP software that does not depend on hardware interrupts or procedure calls, by application programs.
- **Local Host Interface.** To avoid making delivery to the local machine a special case, our implementation creates a pseudo-network interface for local delivery. Recall that the local interface has the same structure as other network interfaces, but corresponds to the local protocol software instead of a physical network. The IP algorithm routes each datagram and passes it to a network interface, including datagrams destined for the local machine. When a conventional network interface receives a datagram, it sends the datagram over a physical network. When the local interface receives a datagram, it uses the PHOTON field to determine which protocol software module on the local machine should receive the datagram. Thus, IP views all routing as uniform and symmetric: it accepts a datagram from any interface and routes it to another interface; no exceptions need to be made for datagrams generated by (or sent to) the local machine.

Although the need to build gateways motivates many of the design decisions, a gateway design works equally well for hosts, and allows us to use the same code for both hosts and gateways. Obviously, combining a uniform routing algorithm with a local



machine interface eliminate several special cases in the code. More important, because the local machine is a valid destination controlled by entries in the routing table, it is possible to add access protections that permit managers to enforce policies on delivery. For example, managers can allow or disallow exchange of information between two application on a given machine as easily as they can allow or disallow communication between applications on separate machine.

5.4 IP Software Organization And Datagram Flow

Chapter 2 described the conceptual organization of IP software, and showed datagram flow for both input and output; this section expands the description and fill in details. Recall that IP consists of a single process and a set of network interface queues through which datagrams must be sent to that process. IP repeatedly extracts a datagram from one of the queues, uses a routing table to choose a next hop for the datagram, and sends the datagram to the appropriate network output process for transmission.

5.4.1 A Policy For Selecting Incoming Datagram

Chapter 3 states that each network interface, including the pseudo-network interface has its own queue of datagram sent to IP. Figure 5.1 illustrates the flow.

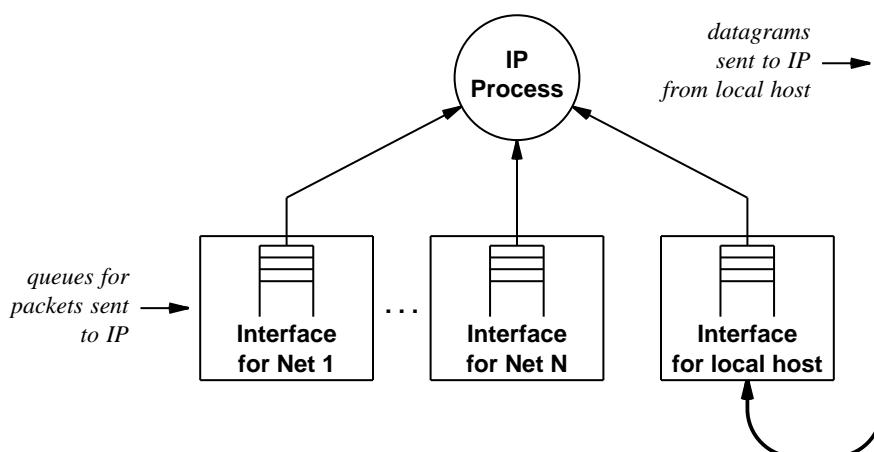


Figure 5.1 IP must select a datagram for processing from the queues associated with network interface. The pseudo-network interface provides a queue used for datagrams generated locally.

If multiple datagrams are waiting in the input queues, the IP process must select one of them to route. The choice of which datagram IP will route determines the



behavior of the system:

The IP code that chooses a datagram to route implements an important polity — it decides the relative priorities of datagram sources.

For example, if IP always selects from the pseudo-network interface queue first, it gives highest priority to outgoing datagrams generated by the local machine. If IP only chooses the pseudo-network queue when all others are empty, it gives highest priority to datagrams that arrive from the network and lowest priority to datagrams generated locally.

It should be obvious that neither extreme is desirable. On one hand, assigning high priority to arriving datagrams means that local software can be blocked arbitrarily long while waiting for IP to route datagrams. For a gateway attached to busy networks, the delay can prevent local applications, including network management applications, from communicating. On the other hand, giving priority to datagrams generated locally means that any application program running on the local machine takes precedent over IP traffic that arrives from the network. If an error causes a local application program to emit datagrams continuously, the outgoing datagrams will prevent arriving datagrams from reaching the network management software. Thus, the manager will not be able to use network management tools to correct the problem.

A correct policy assigns priority fairly and allows both incoming and outgoing traffic to be routed with equal priority. Our implementation achieves fairness by selecting datagrams in a round-robin manner. That is, it selects and routes one datagram from a queue, and then moves on to check the next queue. If K queues contain datagrams waiting to be routed, IP will process one datagram from each of the K queues before processing a second datagram from any of them.

Procedure ipgetp implements the round-robin selection policy.

```
/* ipgetp.c - ipgetp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

static    int ifnext = NI_LOCAL;

/*-----
 * ipgetp -- choose next IP input queue and extract a packet
 *-----
 */
```



```
struct ep *ipgetp(pifnum)
int *pifnum;
{
    struct ep *pep;
    int i;

    recvclr(); /* make sure no old messages are waiting */
    while (TRUE) {
        for (i=0; i < Net.nif; ++i, ++ifnext) {
            if (ifnext >= Net.nif)
                ifnext = 0;
            if (nif[ifnext].ni_state == NIS_DOWN)
                continue;
            if (pep = NIGET(ifnext)) {
                *pifnum = ifnext;
                return pep;
            }
        }
        ifnext = receive();
    }
    /* can't reach here */
}
```

As the code shows, the static variable ifnext serves as an index into the array of interfaces. It iterates through the entire set of network interface structures. At each interface, it checks the state variable ni_state to make sure the interface is enabled. As soon as ipgetp finds an enabled interface with datagrams waiting, it uses macro NIGET to extract and return the first datagram. The next call to ipgetp will continue searching where the previous one left off.

5.4.2 Allowing The IP Process To Block

Procedure ipgetp contains a subtle optimization:

When all input queues are empty, the IP process blocks in a call to procedure ipgetp. Once a datagram arrives, the IP process resumes execution and immediately examines the interface on which the datagram arrived.

To understand the optimization, it is necessary to understand two facts. First, the device driver associated with a particular interface sends the IP process a message whenever it



deposits a datagram on its input queue. Second, the loop in ipgetp ends with a call to receive. After ipgetp iterates through all network interfaces without finding any datagrams, it calls receive, which blocks until a message arrives. When the call to receive returns, it passes the message back to its caller as the function value. The message contains the index of an interface on which a datagram has arrived. Ipgetp assigns the interface index to ifnext and begins the iteration again.

Now that we understand the datagram selection policy IP uses, we can examine the structure of the IP process. The basic algorithm is straightforward. IP repeatedly calls ipgetp to select a datagram, calls a procedure to compute the next-hop address, and deposits the datagram on a queue associated with the network interface over which the datagram must be sent.

Despite its conceptual simplicity, many details complicate the code. For example, if the datagram has arrived from a network, IP must verify that the datagram checksum is correct. If the routing table does not contain a route to the specified destination, IP must generate an ICMP destination unreachable message. If the routing table specifies that the datagram should be sent to a destination on the network on which it originated, IP must generate an ICMP redirect message. Finally, IP must handle the special case of a directed broadcast by sending a copy of the datagram on the specified network and delivering a copy to higher-level protocol software on the gateway itself. The IP process begins execution at procedure ipproc.

```
/* ipproc.c - ipproc */

#include <conf.h>
#include <kernel.h>
#include <network.h>

struct    ep    *ipgetp();
struct    route    *rtget();

/*
 * ipproc - handle an IP datagram coming in from the network
 */
PROCESS ipproc()
{
    struct    ep    *pep;
    struct    ip    *pip;
    struct    route    *prt;
    Bool      nonlocal;
    int       ifnum, rdtype;
```



```
ippid = getpid(); /* so others can find us */

signal(Net.sema); /* signal initialization done */

while (TRUE) {
    pep = ipgetp(&ifnum);
    pip = (struct ip *)pep->ep_data;

    if ((pip->ip_verlen>>4) != IP_VERSION) {
        IpInHdrErrors++;
        freebuf(pep);
        continue;
    }
    if (IP_CLASSE(pip->ip_dst)) {
        IpInAddrErrors++;
        freebuf(pep);
        continue;
    }
    if (ifnum != NI_LOCAL) {
        if (cksum(pip, IP_HLEN(pip)>>1)) {
            IpInHdrErrors++;
            freebuf(pep);
            continue;
        }
        ipnet2h(pip);
    }
    prt = rtget(pip->ip_dst, (ifnum == NI_LOCAL));

    if (prt == NULL) {
        if (gateway) {
            iph2net(pip);
            icmp(ICK_DESTUR, ICC_NETUR,
                 pip->ip_src, pep);
        } else {
            IpOutNoRoutes++;
            freebuf(pep);
        }
        continue;
    }
    nonlocal = ifnum != NI_LOCAL && prt->rt_ifnum != NI_LOCAL;
    if (!gateway && nonlocal) {
```



```
    IpInAddrErrors++;
    freebuf(pep);
    rtfree(prt);
    continue;
}
if (nonlocal)
    IpForwDatagrams++;
/* fill in src IP, if we're the sender */

if (ifnum == NI_LOCAL) {
    if (blkque(pip->ip_src, ip_anyaddr, IP_ALEN))
        if (prt->rt_ifnum == NI_LOCAL)
            blkcopy(pip->ip_src, pip->ip_dst,
                    IP_ALEN);
        else
            blkcopy(pip->ip_src,
                    nif[prt->rt_ifnum].ni_ip,
                    IP_ALEN);
} else if (--(pip->ip_ttl) == 0 &&
           prt->rt_ifnum != NI_LOCAL) {
    IpInHdrErrors++;
    iph2net(pip);
    icmp(ICK_TIMEX, ICC_TIMEX, pip->ip_src, pep);
    rtfree(prt);
    continue;
}
ipdbc(ifnum, pep, prt); /* handle directed broadcasts */
ipredirect(pep, ifnum, prt); /* do redirect, if needed */
if (prt->rt_metric != 0)
    ipputp(prt->rt_ifnum, prt->rt_gw, pep);
else
    ipputp(prt->rt_ifnum, pip->ip_dst, pep);
rtfree(prt);
}
}

int ippid, gateway, bsdbrc;
```

After storing its process id in global variable ippid and signaling the network initialization semaphore, ipproc enters an infinite loop. During each iteration of the loop, ipproc processes one datagram. It calls ipgetp to select a datagram and set ifnum to the



index of the interface from which the datagram was obtained. After checking the datagram version, and verifying that the datagram does not contain a class E address, ipproc calls cksum to verify the checksum (unless the datagram was generated on the local machine).

Once it has obtained a valid datagram, ipproc calls procedure rtget to route the datagram. The next chapter reviews the details of rtget; for now, it is only important to understand that rtget computes a route and returns a pointer to a structure that describes the route. If no route exists, ipproc calls procedure icmp^① to form and send an ICMP destination unreachable message.

Ipproc must fill in a correct source address for datagrams that originate on the local machine. To do so, it examines the datagram to see if higher-level protocol software has specified a fixed source address. If not, ipproc fills in the source address field. Following the standard, ipproc assigns the datagram source the IP address of the network interface over which the datagram will be sent. If the route refers to the local host interface (i.e., the datagram is being routed from the local machine back to the local machine), ipproc copies the datagram destination address into the source address field.

Once routing is complete, ipproc decrements the time-to-live counter (ip_ttl). If the time-to-live field reaches zero, ipproc generates an ICMP time exceeded message.

Ipproc calls procedure ipdbc to handle directed broadcasts. Ipdbc, shown in section 5.4.5, creates a copy of those directed broadcast datagrams destined for the local machine, and sends a copy to the local software. Ipproc transmits the original copy to the specified network.

Ipproc also generates ICMP redirect messages. To determine if such a message is needed, ipproc compares the interface from which the datagram was obtained to the interface to which it was routed. If they are the same, a redirect is needed. Ipproc examines the network's subnet mask to determine whether it should send a network redirect or a host redirect.

Finally, ipproc examines the routing metric to determine whether it should deliver the datagram to its destination or send it to the next-hop address. A routing metric of zero means the gateway can deliver the datagram directly; any larger value means the gateway should send the datagram to the next-hop address. After selecting either the next-hop address or the destination address, ipproc calls rpputp to insert the datagram on one of the network output queues.

5.4.3 Definitions Of Constants Used By IP

File ip.h contains definitions of symbolic constants used in the IP software. It also defines the format of an IP datagram with structure ip.

^① Chapter 8 describes the implementation of icmp.



```
/* ip.h - IP_HLEN */

/* Internet Protocol (IP) Constants and Datagram Format      */

#define IP_ALEN 4           /* IP address length in bytes (octets)      */
typedef char IPAddr[IP_ALEN]; /* internet address                      */

#define IP_CLASSA(x) ((x[0] & 0x80) == 0x00)/* IP Class A address      */
#define IP_CLASSB(x) ((x[0] & 0xc0) == 0x80)/* IP Class B address      */
#define IP_CLASSC(x) ((x[0] & 0xe0) == 0xc0)/* IP Class C address      */
#define IP_CLASSD(x) ((x[0] & 0xf0) == 0xe0)/* IP Class D address      */
#define IP_CLASSE(x) ((x[0] & 0xf8) == 0xf0)/* IP Class E address      */

/* Some Assigned Protocol Numbers */

#define IPT_ICMP 1    /* protocol type for ICMP packets      */
#define IPT_IGMP 2    /* protocol type for IGMP packets      */
#define IPT_TCP 6     /* protocol type for TCP packets       */
#define IPT_EGP 8     /* protocol type for EGP packets       */
#define IPT_UDP 17    /* protocol type for UDP packets       */
#define IPT_OSPF 89   /* protocol type for OSPF packets      */

struct ip {
    char ip_verlen;    /* IP version & header length (in longs) */
    char ip_tos;        /* type of service                      */
    short ip_len;       /* total packet length (in octets)      */
    short ip_id;        /* datagram id                         */
    short ip_fragoff;  /* fragment offset (in 8-octet's)      */
    char ip_ttl;        /* time to live, in gateway hops     */
    char ip_proto; /* IP protocol (see IPT_* above) */
    short ip_cksum; /* header checksum                     */
    IPAddr ip_src;    /* IP address of source                */
    IPAddr ip_dst;    /* IP address of destination          */
    char ip_data[1];  /* variable length data               */
};

#define IP_VERSION 4    /* current version value              */
#define IP_MINHLEN 5    /* minimum IP header length (in longs) */
#define IP_TTL 255      /* Initial time-to-live value        */

/* IP Precedence values */
```



```
#define IPP_NETCTL    0xe0 /* network control          */
#define IPP_INCTL     0xc0 /* internet control           */
#define IPP_CRIT      0xa0 /* critical                   */
#define IPP_FLASHO    0x80 /* flash over-ride            */
#define IPP_FLASH     0x60 /* flash                      */
#define IPP_IMMED     0x40 /* immediate                 */
#define IPP_PRIO      0x20 /* priority                  */
#define IPP_NORMAL    0x00 /* normal                     */

/* macro to compute a datagram's header length (in bytes)          */
#define IP_HLEN(pip) ((pip->ip_verlen & 0xf)<<2)
#define IPMHLEN      20 /* minimum IP header length (in bytes) */

/* IP options */
#define IPO_COPY     0x80 /* copy on fragment mask       */
#define IPO_CLASS    0x60 /* option class                */
#define IPO_NUM      0x17 /* option number               */

#define IPO_EOOP     0x00 /* end of options             */
#define IPO_NOP      0x01 /* no operation                */
#define IPO_SEC      0x82 /* DoD security/compartimentalization */
#define IPO_LSRCRT   0x83 /* loose source routing        */
#define IPO_SSRCRT   0x89 /* strict source routing       */
#define IPO_RECRT   0x07 /* record route                */
#define IPO_STRID    0x88 /* stream ID                  */
#define IPO_TIME     0x44 /* internet timestamp          */

#define IP_MAXLEN    BPMAXB-EP_HLEN /* Maximum IP datagram length */

/* IP process info */

extern int ipproc();

#define IPSTK      1000 /* stack size for IP process */
#define IPPRI     100 /* IP runs at high priority */
#define IPNAM     "ip" /* name of IP process        */
#define IPARGC    0 /* count of args to IP      */

extern IPAddr ip_maskall; /* = 255.255.255.255 */
extern IPAddr ip_anyaddr; /* = 0.0.0.0 */
extern IPAddr ip_loopback; /* = 127.0.0.1 */
```



```
extern int ippid, gateway;
```

5.4.4 Checksum Computation

Ipproc uses procedure cksum to compute or verify the header checksum. The header checksum treats the header as a sequence of 16-bit integers, and defines the checksum to be the ones complement of the sum of all 16-bit integers in the header. Also, the sum and complement are defined to use ones complement arithmetic.

Most machines compute in twos-complement arithmetic, so merely accumulating a 16-bit checksum will not produce the desired result. To make it portable and avoid coding in assembler language, procedure cksum has been written in C. The implementation uses 32-bit (long) arithmetic to accumulate a sum, and then folds the result to a 16-bit value by adding any carry bits into the sum explicitly. Finally, cksum returns the ones complement of the result.

```
/* cksum.c - cksum */

/*-----
 * cksum - Return 16-bit ones complement of 16-bit ones complement sum
 *-----
 */
short cksum(buf, nwords)
unsigned short    *buf;
int      nwords;
{
    unsigned long sum;

    for (sum=0; nwords>0; nwords--)
        sum += *buf++;
    sum = (sum >> 16) + (sum & 0xffff); /* add in carry */
    sum += (sum >> 16);                /* maybe one more */
    return ~sum;
}
```

5.4.5 Handling Directed Broadcasts

Whenever a datagram is sent to a directed broadcast address, all machines on the specified destination network must receive a copy. The subtle point to remember is that:

Directed broadcast includes both gateways and hosts on the destination network, even if one of those gateways is responsible for forwarding the datagram onto the network.



However, most network hardware does not deliver a copy of a broadcast packet back to the machine that transmits the broadcast. If a gateway needs a copy of a broadcast datagram, software must take explicit action to keep one. Thus, if a gateway receives a datagram with destination address equal to the directed broadcast address for one of its directly connected networks, the gateway must do two things: (1) make a copy of the datagram for protocol software on the local machine, and (2) broadcast the datagram on the specified network. Procedure ipdbc contains the code to handle such broadcasts.

```
/* ipdbc.c - ipdbc */

#include <conf.h>
#include <kernel.h>
#include <network.h>

struct route *rtget();

/*-----
 * ipdbc - handle IP directed broadcast copying
 *-----
 */

void ipdbc(ifnum, pep, prt)
int ifnum;
struct ep *pep;
struct route *prt;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct ep *pep2;
    struct route *prt2;
    int len;

    if (prt->rt_ifnum != NI_LOCAL)
        return; /* not ours */
    if (!isbrc(pip->ip_dst))
        return; /* not broadcast */

    prt2 = rtget(pip->ip_dst, RTF_LOCAL);
    if (prt2 == NULL)
        return;
    if (prt2->rt_ifnum == ifnum) { /* not directed */
        rtfree(prt2);
        return;
    }
}
```



```
/* directed broadcast; make a copy */

/* len = ether header + IP packet */

len = EP_HLEN + pip->ip_len;
if (len > EP_MAXLEN)
    pep2 = (struct ep *)getbuf(Net.lrgpool);
else
    pep2 = (struct ep *)getbuf(Net.netpool);
if (pep2 == (struct ep *)SYSERR) {
    rtfree(prt2);
    return;
}
blkcopy(pep2, pep, len);
/* send a copy to the net */

ipputp(prt2->rt_ifnum, pip->ip_dst, pep2);
rtfree(prt2);

return;      /* continue; "pep" goes locally in IP */
}
```

Ipproc calls ipdbc for all datagrams, most of which do not specify directed broadcast. Ipdbc begins by checking the source of the datagram because datagrams that originate on the local machine do not need copies. Ipdbc then calls isbrc to compare the destination address to the directed broadcast addresses for all directly connected networks, because nonbroadcasts do not need copies. For cases that do not need copies, ipdbc returns without taking any action; ippoc will choose a route and forward the datagram as usual.

Datagrams sent to the directed broadcast address for one of the directly connected networks must be duplicated. One copy must be sent to the local host software, while the other copy is forwarded as usual. To make a copy, ipdbc allocates a buffer, choosing from the standard network buffer pool or the pool for large buffers, depending on the datagram size. If the buffer allocation is successful, ipdbc copies the datagram into the new buffer and deposits the new buffer on the output port associated with the network interface over which it must be sent. After ipdbc returns, ippoc passes the original copy to the local machine through the pseudo-network interface.



5.4.6 Recognizing A Broadcast Address

The IP protocol standard specifies three types of broadcast addresses: a local network broadcast address (all 1's), a directed network broadcast address (a class A, B, or C IP address with host portion of all 1's), and a subnet broadcast address (subnetted IP address with host portion all 1's). Unfortunately, when Berkeley incorporated TCP/IP into the BSD UNIX distribution, they decided to use nonstandard broadcast addresses. Sometimes called Berkeley broadcast, these forms of broadcast use all 0's in place of all 1's.

While the Berkeley form of broadcast address is definitely nonstandard, many commercial systems derived from the Berkeley code have adopted it. To accommodate the widespread Berkeley convention, our example code accepts broadcasts using either all 0's or all 1's. Procedure `isbrc` contains the code.

```
/* isbrc.c - isbrc */

#include <conf.h>
#include <kernel.h>
#include <sleep.h>
#include <network.h>

/*
 *-----*
 * isbrc - Is "dest" a broadcast address?
 *-----*
 */
Bool isbrc(dest)
    IPIaddr dest;
{
    int inum;

    /* all 0's and all 1's are broadcast */

    if (blkequ(dest, ip_anyaddr, IP_ALEN) ||
        blkequ(dest, ip_maskall, IP_ALEN))
        return TRUE;

    /* check real broadcast address and BSD-style for net & subnet */

    for (inum=0; inum < Net.nif; ++inum)
        if (blkequ(dest, nif[inum].ni_brc, IP_ALEN) ||
            blkequ(dest, nif[inum].ni_nbrc, IP_ALEN) ||
            blkequ(dest, nif[inum].ni_subnet, IP_ALEN) ||
```



```
    blkequ(dest, nif[inum].ni_net, IP_ALEN))  
    return TRUE;  
  
    return FALSE;  
}
```

5.5 Byte-Ordering In The IP Header

To keep the Internet Protocol independent of the machines on which it runs, the protocol standard specifies network byte ordering for all integer quantities in the header:

Before sending a datagram, the host must convert all integers from the local machine byte order to standard network byte order; upon receiving a datagram, the host must convert integers from standard network byte order to the local machine byte order.

Procedures iph2net and ipnet2h perform the conversions; ipnet2h is called from ipproc, and iph2net is called from ipfsend, ipproc, and ippup. To convert individual fields, the utility routines use functions net2hs (network-to-host-short) and hs2net (host-short-to-network). The terminology is derived from the C programming language, where short generally refers to a 16-bit integer and long generally refers to a 32-bit integer.

To optimize processing time, our code stores all IP addresses in network byte order and does not convert address fields in protocol headers. Thus, the code only converts integer fields that do not contain IP addresses.

```
/* iph2net.c - iph2net */  
  
#include <conf.h>  
#include <kernel.h>  
#include <network.h>  
  
/*-----  
 * iph2net - convert an IP packet header from host to net byte order  
 *-----  
 */  
struct ip *iph2net(pip)  
struct ip *pip;  
{  
    /* NOTE: does not include IP options */
```



```
    pip->ip_len = hs2net(pip->ip_len);
    pip->ip_id = hs2net(pip->ip_id);
    pip->ip_fragoff = hs2net(pip->ip_fragoff);
    return pip;
}

/* ipnet2h.c - ipnet2h */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ipnet2h - convert an IP packet header from net to host byte order
 */
struct ip *ipnet2h(pip)
struct    ip   *pip;
{
    /* NOTE: does not include IP options */

    pip->ip_len = net2hs(pip->ip_len);
    pip->ip_id = net2hs(pip->ip_id);
    pip->ip_fragoff = net2hs(pip->ip_fragoff);
    return pip;
}
```

5.6 Sending A Datagram To IP

5.6.1 Sending Locally-Generated Datagrams

Given a locally-generated datagram and an IP destination address, procedure ipsend fills in the IP header and enqueues the datagram on the local host interface, where the IP process will extract and send it.

```
/* ipsend.c - ipsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>

static ipackid = 1;
```



```
/*
 * ipsend - fill in IP header and send datagram to specified address
 */
int ipsend(faddr, pep, datalen, proto, ptos, ttl)
IPAddr faddr;
struct ep *pep;
int datalen;
unsigned char proto; /* IP protocol */
unsigned char ptos; /* Precedence / Type-of-Service */
unsigned char ttl; /* time to live */
{
    struct ip *pip = (struct ip *) pep->ep_data;

    pep->ep_type = EPT_IP;
    pip->ip_verlen = (IP_VERSION<<4) | IP_MINHLEN;
    pip->ip_tos = ptos;
    pip->ip_len = datalen+IP_HLEN(pip);
    pip->ip_id = ipackid++;
    pip->ip_fragoff = 0;
    pip->ip_ttl = ttl;
    pip->ip_proto = proto;
    blkcopy(pip->ip_dst, faddr, IP_ALEN);

    /*
     * special case for ICMP, so source matches destination
     * on multi-homed hosts.
     */
    if (pip->ip_proto != IPT_ICMP)
        blkcopy(pip->ip_src, ip_anyaddr, IP_ALEN);

    if (enq(nif[NI_LOCAL].ni_ipinq, pep, 0) < 0) {
        freebuf(pep);
        IpOutDiscards++;
    }
    send(ippid, NI_LOCAL);
    IpOutRequests++;
    return OK;
}
/* special IP addresses */
```



```
IPAddr    ip_anyaddr = { 0, 0, 0, 0 };
IPAddr    ip_loopback = { 127, 0, 0, 1 };
```

Arguments permit the caller to specify some of the values used in the IP header. Argument proto contains a value used for the protocol type, ptos contains a value used for the field that represents precedence and type-of-service, and argument ttl contains a value for the time-to-live field.

Ipsend fills in each of the header fields, including the specified destination address. To guarantee that each outgoing datagram has a unique value in its identification fields, ippoc assigns the identification the value of global variable ipackid and then increments the variable. After it assigns the header, ippoc calls enq to enqueue the datagram on the queue located in the local host (pseudo-network) interface.

Observe that although the ni_ipinq queues in network interfaces normally contain incoming datagrams (i.e., datagrams arriving from other sites), the queue in the pseudo-network interface contains datagrams that are "outgoing" from the point of view of application software. Finally, ipsend calls send to send a message to the IP process in case it was blocked waiting for datagrams to arrive.

5.6.2 Sending Incoming Datagrams

When an IP datagram arrives over a network, device driver code in the network interface layer must deposit it on the appropriate queue for IP. To do so, it calls ip_in.

```
/* ip_in.c - ip_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ip_in - IP input function
 */
int ip_in(pni, pep)
struct netif *pni;
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;

    IpInReceives++;
    if (enq(pni->ni_ipinq, pep, pip->ip_tos & IP_PREC) < 0) {
```



```
    IpInDiscards++;
    freebuf(pep);
}
send(ippid, (pni-&nif[0]));
return OK;
}
```

Given a pointer to a buffer that contains a packet, ip_in calls enq to enqueue the packet on the queue in the interface. If the queue is full, ip_in increments variable IpInDiscards to record the queue overflow error and discards the packet. Finally, ip_in sends a message to the IP process in case it is blocked waiting for a datagram.

5.7 Table Maintenance

IP software needs a timing mechanism for maintenance of network data structures, including the IP routing table and fragment reassembly table. Our example implements such periodic tasks with a timer process. In fact, the timer is not limited to IP tasks — it also triggers ARP cache timeouts, and can be used for any other long-term periodic tasks that do not have stringent delay requirements. The code, in procedure slowtimer, shows how easily new tasks can be added to the list.

```
/* slowtimer.c - slowtimer */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

#define STGRAN 1           /* Timer granularity (delay) in seconds */

/*
 * slowtimer - handle long-term periodic maintenance of network tables
 */
PROCESS slowtimer()
{
    long lasttime, now;      /* previous and current times in seconds*/
    int delay;               /* actual delay in seconds */
    signal(Net.sema);
```



```
    gettimeofday(&lasttime);
    while (1) {
        sleep(STGRAN);
        gettimeofday(&now);
        delay = now - lasttime;
        if (delay <= 0 || delay > 4*STGRAM)
            delay = STGRAM; /* likely clock reset */
        lasttime = now;
        arptimer(delay);
        ipftimer(delay);
        rttimer(delay);
        ospftimer(delay);
    }
}
```

As the code shows, slowtimer consists of an infinite loop that repeatedly invokes a set of maintenance procedures. A given maintenance procedure may take arbitrarily long to complete its chore, and the execution time may vary between one invocation and the next. Thus, slowtimer computes the actual delay between executions and reports it to the maintenance procedures as an argument.

5.8 Summary

To simplify the code, our implementation of IP executes as a single, independent process, and interaction with higher-level protocol software on the local machine occurs through a pseudo-network interface. When no datagrams are available, the IP process blocks. As soon as one or more datagrams are available from any source, the IP process awakens and processes them until they have all been routed. To make processing fair and avoid starvation, our implementation uses a round-robin policy among input sources, including the pseudo-interface that corresponds to the local machine. Thus, neither locally generated traffic nor incoming traffic from the network connections has priority.

Directed broadcasting means delivery to all hosts and gateways on the specified network. The protocol standard allows designers to decide whether to forward directed broadcasts that originate on foreign networks. If the gateway chooses to allow directed broadcasts, it routes them as usual. If the destination address specifies a directly connected network, IP must be sure that higher-level protocol software on the local machine receives a copy of the datagram. To increase its utility, our example implementation allows either the TCP/IP standard (all 1's) or 4.2 BSD UNIX (all 0's) forms of broadcast address. It creates a copy of a broadcast datagram and arranges for the network interface to broadcast the copy, while it routes the original to protocol



software on the local machine.

The IP checksum consists of a 16-bit 1's complement value that can be computed using 32-bit 2's complement arithmetic and carry propagation.

5.9 FOR FURTHER STUDY

The standard for IP is found in Postel [RFC 791]. Braden and Postel [RFC 1009] summarizes requirements for Internet gateways. Mallory [RFC 1141] discusses incremental update of IP checksums. Braden, Borman, and Partridge [RFC 1071] gives an earlier discussion. Mogul and Postel [RFC 950] gives the standard for subnet addressing. Padlipsky [RFC 875], and Hinden and Sheltzer [RFC 823] describe early ideas about gateways.

5.10 EXERCISES

1. One's complement arithmetic tins two values for zero. Which will cksum return?
2. Rewrite cksum in assembly language. How does the speed compare to a version written in C?
3. Consider an implementation that uses a single input queue for all datagrams sent to IP. What is the chief disadvantage of such a solution?
4. Study the code in procedure ipproc carefully. Identify all instances where a datagram sent to/from the local machine it treated as a special case.
5. Can any of the special cases in the previous exercise be eliminated by requiring higher-level protocols to perform computation(s) when they enqueue a datagram for ouput?
6. Show that it is possible for ipproc to make one last iteration through all interfaces even though there are not datagrams waiting to be processed. Hint: consider the timing between the IP process and a device driver that deposits a datagram and sends IP a message.
7. Consider the AT&T STREAMS mechanism used to build device driver and protocol software. Can it be used to implement IP? How?
8. What is the chief advantage of implementing IP in an independent process? What is the chief disadvantage?
9. Procedure ipsend supplies a fixed value for the time-to-live field in the datagram header. Is this reasonable?
10. Look carefully at the initial value used for the datagram identification field. Argue that if a machine boots, sends a datagram, crashes, quickly reboots and



sends a different datagram to the same destination, fragmentation can cause severe errors

11. Procedure ip_in discards an incoming datagram when it finds that an interface queue is full. Read the RFC to determine whether IP should generate an error message when the situation occurs.
12. Design a minor modification to the code for slowtimer that produces more accurate values in calls to maintenance procedures. What are the advantages and disadvantages of each implementation?



6 IP: Routing Table And Routing Algorithm

6.1 Introduction

The previous chapter described the overall structure of Internet Protocol (IP) software and showed the code for the central procedure, ipproc. This chapter continues the discussion by presenting the details of routing. It examines the organization of an IP routing table and the definitions of data structures that implement it. It discusses the routing algorithm and shows how IP uses subnet masks when selecting a route. Finally, it shows how IP distinguishes between network-specific routes, subnet-specific routes, and host-specific routes.

6.2 Route Maintenance And Lookup

Conceptually, routing software can be divided into two groups. One group includes procedures used to determine the correct route for a datagram. The other group includes procedures used to add, change, or delete routes. Because a gateway must determine a route for each datagram it processes, the route lookup code determines the overall performance of the gateway. Thus, the lookup code is usually optimized for highest speed.

Route insertions, changes, or deletions usually occur at much slower rates than datagram routing. Programs that compute new routes communicate with other machines to establish reachability; they can take arbitrarily long before changing routes. Thus, route update procedures need not be as optimized as lookup operations. The fundamental idea is:

IP data structure, and algorithms should be selected to optimize the cost of route lookup; the cost of route maintenance is not as important.



Although early TCP/IP software often used linear search for routing table lookup, most systems now use a hash table that permits arbitrarily large routing tables to be searched quickly.

Our software uses a form of bucket hashing. It partitions route table entries into many "buckets" and uses a hash function to find the appropriate bucket quickly,

6.3 Routing Table Organization

Figure 6.1 illustrates the data structure used for the route table.

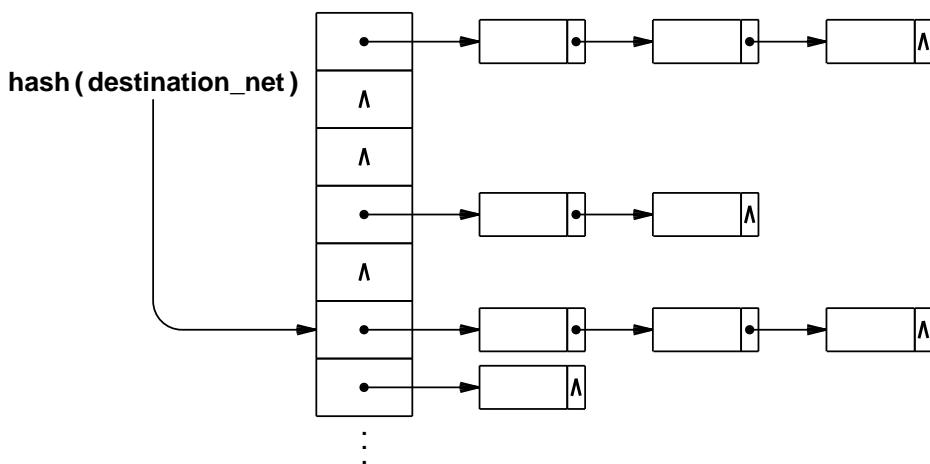


Figure 6.1 Implementation of a hashed route table using an array. Each entry in the array points to a linked list of records that each contain a destination address and a route to that destination.

The main data structure for storing routes is an array. Each entry in the array corresponds to a bucket and contains a pointer to a linked list of records for routes to destinations that hash into that bucket. Each record on the list contains a destination IP address, subnet mask, next-hop address for that destination, and the network interface to use for sending to the next-hop address, as well as other information used in route management. Because it cannot know subnet masks a priori, IP uses only the network portion of the destination IP address when computing the hash function. When searching entries on a linked list, however, IP uses the entire destination address to make comparisons. Later sections, present the details.

6.4 Routing Table Data Structures

File route.h contains the declarations of routing table data structures.



```
/* route.h - RTFREE */

/* Routing Table Entries: */
struct route {
    IPAddr    rt_net;          /* network address for this route      */
    IPAddr    rt_mask;         /* mask for this route                  */
    IPAddr    rt_gw;           /* next IP hop                         */
    short     rt_metric;       /* distance metric                     */
    short     rt_ifnum;        /* interface number                    */
    short     rt_key;          /* sort key                           */
    short     rt_ttl;          /* time to live   (seconds)           */
    struct    route *rt_next;  /* next entry for this hash value   */
/* stats */
    int      rt_refcnt;       /* current reference count            */
    int      rt_usecnt;        /* total use count so far           */
};

/* Routing Table Global Data: */
struct rtinfo {
    struct    route    *ri_default;
    int       ri_bpool;
    Bool     ri_valid;
    int       ri_mutex;
};

#define  RT_DEFAULT ip_anyaddr /* the default net      */
#define  RT_LOOPBACK ip_loopback /* the loopback net    */
#define  RT_TSIZE 512 /* these are pointers; it's cheap */
#define  RT_INF      999 /* no timeout for this route */
#define  RTM_INF      16 /* an infinite metric */

/* rtget()'s second argument... */

#define  RTF_REMOTE    0 /* traffic is from a remote host */
#define  RTF_LOCAL1    /* traffic is locally generated */

#define  RT_BPSIZE100 /* max number of routes      */

/* RTFREE - remove a route reference (assumes ri_mutex HELD) */
```



```
#define RTFREE(prt)                                \
    if (--prt->rt_refcnt <= 0) {                   \
        freebuf(prt);                                \
    }                                                 \
                                                    \
extern struct rtinfo Route;                      \
extern struct route *rttable[];
```

Structure route defines the contents of a node on the linked lists, and contains routing information for one possible destination. Field rt_net specifies the destination address (either a network, subnet, or complete host address); field rt_mask specifies the 32-bit mask used with that destination. The mask entries can cover the network portion, network, plus subnet portion, or the entire 32 bits (i.e., they can include the host portion).

Field rt_gw specifies the IP address of the next-hop gateway for the route, and field rt_metric gives the distance of the gateway (measured in hops). Field rt_ifnum gives the internal number of the network interface used for the route (i.e., the network used to reach the next-hop gateway).

Remaining fields are used by the IP software. Field rt_key contains a sort key used when inserting the node on the linked list. Field rt_refcnt contains a reference count of processes that hold a pointer to the route, and field rt_usecnt records the number of times the route has been used. Finally, field rt_next contains a pointer to the next node on the linked list (the last node in a list contains NULL).

In addition to the route structure, file route.h defines the routing table, rttable. As Figure 6.1 shows, rttable is an array of pointers to route structures.

In addition to the routing table, IP requires a few other data items. The global structure rtinfo holds them. For example, the system provides a single default route that is used for any destination not contained in the table. Field rt_default points to a route structure that contains the next-hop address for the default route. Field ri_valid contains a Boolean variable that is TRUE if the routing data structures have been initialized.

6.5 Origin Of Routes And Persistence

Information in the routing table comes from several sources. When the system starts, initialization routines usually obtain an initial set of routes from secondary storage and install them in the table. During execution, incoming messages can cause ICMP or routing protocol software to change existing routes or install new routes. Finally, network managers can also add or change routes.

The volatility of a routing entry depends on its origin. For example, initial routes



are usually chosen to be simplistic estimates, which should be replaced as soon as routing information arrives from any other source. However, network managers must be able to override any route and install permanent, unalterable routes that allow them to debug network routing problems without interference from routing protocols.

To accommodate flexibility in routes, field `rt_ttl` (time-to-live) in each routing entry specifies a time, in seconds, that the entry should remain valid. When `rt_ttl` reaches zero, the route is no longer considered valid and will be discarded. Routing protocols can install routes with time-to-live values computed according to the rules of the protocol, while managers can install routes with infinite time-to-live, guaranteeing that they will not be removed.

6.6 Routing A Datagram

6.6.1 Utility Procedures

Several utility procedures provide function used in routing. Procedure `netnum` extracts the network portion of a given IP address, using the address class to determine which octets contain the network part and which contain the host part. It returns the specified address with all host bytes set to zero.

```
/* netnum.c - netnum */

#include <conf.h>
#include <kernel.h>
#include <network.h>

-----
 * netnum - compute the network portion of a given IP address
 -----
 */
int netnum(net, ipa)
IPAddr net, ipa;
{
    int bc = IP_ALEN;

    blkcopy(net, ipa, IP_ALEN);
    if (IP_CLASSA(net)) bc = 1;
    if (IP_CLASSB(net)) bc = 2;
    if (IP_CLASSC(net)) bc = 3;
    for (; bc < IP_ALEN; ++bc)
        net[bc] = 0;
    return OK;
```



}

IP uses procedure netmatch during routing to compare a destination (host) address to a routing entry. The routing entry contains the subnet mask and IP address for a given network. Netmatch uses the subnet mask to mask off the host bits in the destination address and compares the results to the network entry. If they match, netmatch returns TRUE; otherwise it returns FALSE.

Broadcasting is a special case because the action to be taken depends on the source of the datagram. Broadcast datagrams that arrive from network interfaces must be delivered to the local machine via the pseudo-network interface, while locally-generated broadcast datagrams must be sent to the appropriate network interface. To distinguish between the two, the software uses a host-specific route (a mask of all 1's) to route arriving broadcast datagrams, and a network-specific route (the mask covers only the network portion) to route outgoing broadcasts. Thus, netmatch tests for a broadcast datagram explicitly, and uses the IP source address to decide whether the broadcast matches a given route.

```
/* netmatch.c - netmatch */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * netmatch - Is "dst" on "net"?
 */
Bool netmatch(dst, net, mask, islocal)
    IPAddr dst, net, mask;
    Bool islocal;
{
    int i;

    for (i=0; i<IP_ALEN; ++i)
        if ((mask[i] & dst[i]) != net[i])
            return FALSE;
    /*
     * local srcs should not match broadcast addresses (host routes)
     */
    if (islocal)
        if (isbrc(dst))
```



```
        return !blkequ(mask, ip_maskall, IP_ALEN);
    return TRUE;
}
```

To route a datagram, IP must first see if it knows a valid subnet mask for the destination address. To do so, it calls procedure netmask.

```
/* netmask.c - netmask */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * netmask - set the default mask for the given net
 */
int netmask(mask, net)
IPAddr    mask;
IPAddr    net;
{
    IPAddr    netpart;
    Bool isdefault = TRUE;
    int i;
    int bc = IP_ALEN;

    for (i=0; i<IP_ALEN; ++i) {
        mask[i] = ~0;
        isdefault &= net[i] == 0;
    }
    if (isdefault) {
        blkcopy(mask, net, IP_ALEN);
        return OK;
    }
    /* check for net match (for subnets) */

    netnum(netpart, net);
    for (i=0; i<Net.nif; ++i) {
        if (nif[i].ni_svalid && nif[i].ni_invalid &&
            blkequ(nif[i].ni_net, netpart, IP_ALEN)) {
            blkcopy(mask, nif[i].ni_mask, IP_ALEN);
            return OK;
    }
}
```



```
        }

    }

    if (IP_CLASSA(net)) bc = 1;
    if (IP_CLASSB(net)) bc = 2;
    if (IP_CLASSC(net)) bc = 3;
    for (; bc < IP_ALEN; ++bc)
        mask[bc] = 0;
    return OK;
}
```

Netmask takes the address a subnet mask variable in its first argument and the address of a destination IP address in its second. It begins by setting the subnet mask to all 0's, and then checks several cases. By convention, if the destination address is all 0's, it specifies a default route, so netmask returns a subnet mask of all 0's. For other destination, netmask calls netnum to extract the network portion of the destination address, and then checks each locally-connected network matches the network portion of the destination, netmask extracts the subnet mask from the network interface structure for that network and returns it to the caller. Finally, if IP has no information about the subnet mask of the destination address, it sets the subnet mask to cover the network part of the address, depending on whether the address is class A, B, or C.

The routing function calls utility procedure rhash to hash a destination network address.

```
/* rhash.c - rhash */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * rhash - compute the hash for "net"
 */
int rhash(net)
IPAddr net;
{
    int bc = IP_ALEN; /* # bytes to count */
    int hv = 0; /* hash value */

    if (IP_CLASSA(net)) bc = 1;
    else if (IP_CLASSB(net)) bc = 2;
```



```
    else if (IP_CLASSC(net)) bc = 3;
    else if (IP_CLASSD(net))
        return (net(0) & 0xf0) % RT_TSIZE;
    while (--bc)
        hv += net[bc] & 0xff;
    return hv % RT_TSIZE;
}
```

The hash function used is both simple efficient to compute. Rthash sums the individual octets of the network address, divides by the hash table size, and returns the remainder.

6.6.2 Obtaining A Route

Given a destination address procedure rtget searches the routing table and returns a pointer to the entry for that route.

```
/* rtget.c - rtget */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * rtget - get the route for a given IP destination
 */
struct route *rtget(dest, local)
IPAddr dest;
Bool local; /* TRUE <=> locally generated traffic */
{
    struct route *prt;
    int hv;

    if (!Route.ri_valid)
        rtinit();
    wait(Route.ri_mutex);
    hv = rthash(dest);
    for (prt=rttable[hv]; prt; prt=prt->rt_next) {
        if (prt->rt_ttl <= 0)
            continue; /* route has expired */
        if (netmatch(dest, prt->rt_net, prt->rt_mask, local))
```



```
        if (prt->rt_metric < RTM_INF)
            break;
    }
    if (prt == 0)
        prt = Route.ri_default; /* may be NULL too... */
    if (prt != 0 && prt->rt_metric >= RTM_INF)
        prt = 0;
    if (prt) {
        prt->rt_refcnt++;
        prt->rt_usecnt++;
    }
    signal(Route.ri_mutex);
    return prt;
}
```

The global variable Route.ri_valid specifies whether the table has been initialized. If it has not, rtget calls rtinit. Once the routing table and associated data structures have been initialized, rtget waits on the mutual exclusion semaphore to insure that only one process accesses the table at any time. It then computes the hash value of the destination address, uses it as an index into the table, and follows the linked list of routing entries.

At each entry, rtget calls netmatch to see if the destination specified by its argument matches the address in the entry. If no explicit match is found during the search, rtget uses the default route found in Route.ri_default.

Of course, it is possible that there is no default route and no explicit match. Thus, after performing route lookup, rtget must still check to see if it found a valid pointer, if it has, rtget increments the reference count and use count fields of the route entry before returning to the caller. Maintenance software uses the reference count field to determine whether it is safe to delete storage associated with the route. The reference count will remain nonzero as long as the procedure that called rtget needs to use the route entry. The use count provides a way for network administrators to find out how often each entry has been used to route datagrams.

6.6.3 Data Structure Initialization

Procedure rtinit initializes the routing table and default route, creates the mutual exclusion semaphore, allocates storage for nodes on the linked lists of routes, and links the storage onto a free list. The implementation is straightforward.

```
/* rtinit.c - rtinit */

#include <conf.h>
#include <kernel.h>
```



```
#include <sleep.h>
#include <network.h>

struct    rtinfo    Route;
struct    route     *rttable[RT_TSIZE];

/*-----
 * rtinit - initialize the routing table
 *-----
 */
void rtinit()
{
    int i;

    for (i=0; i<RT_TSIZE; ++i)
        rttable[i] = 0;
    Route.ri_bpool = mkpool(sizeof(struct route), RT_BPSIZE);
    Route.ri_valid = TRUE;
    Route.ri_mutex = screate(1);
    Route.ri_default = NULL;
}
```

6.7 Periodic Route Table Maintenance

The system initiates a periodic sweep of the routing table to decrement time-to-live values and dispose of routes that have expired. Procedure rttimer implements the periodic update.

```
/* rttimer.c - rttimer */

#include <conf.h>
#include <kernel.h>
#include <network.h>

extern    Bool dorip;          /* TRUE if we're running RIP output */
extern    int  rippid;         /* RIP output pid, if running */

/*-----
 * rttimer - update ttls and delete expired routes
 *-----
 */
int rttimer(delta)
```



```
{  
    struct    route      *prt, *prev;  
    Bool      ripnotify;  
    int       i;  
  
    if (!Route.ri_valid)  
        return;  
    wait(Route.ri_mutex);  
  
    ripnotify = FALSE;  
    for (i=0; i<RT_TSIZE; ++i) {  
        if (rttable[i] == 0)  
            continue;  
        for (prev = NULL, prt = rttable[i]; prt != NULL;) {  
            if (prt->rt_ttl != RT_INF)  
                prt->rt_ttl -= delta;  
            if (prt->rt_ttl <= 0) {  
                if (dorip && prt->rt_metric < RTM_INF) {  
                    prt->rt_metric = RTM_INF;  
                    prt->rt_ttl = RIPZTIME;  
                    ripnotify = TRUE;  
                    continue;  
                }  
                if (prev) {  
                    prev->rt_next = prt->rt_next;  
                    RTFREE(prt);  
                    prt = prev->rt_next;  
                } else {  
                    rttable[i] = prt->rt_next;  
                    RTFREE(prt);  
                    prt = rttable[i];  
                }  
                continue;  
            }  
            prev = prt;  
            prt = prt->rt_next;  
        }  
    }  
    prt = Route.ri_default;  
    if (prt && (prt->rt_ttl<RT_INF) && (prt->rt_ttl -= delta) <= 0)  
        if (dorip && prt->rt_metric < RTM_INF) {
```



```
    prt->rt_metric = RTM_INF;
    prt->rt_ttl = RIPZTIME;
} else {
    RTFREE(Route.ri_default);
    Route.ri_default = 0;
}
signal(Route.ri_mutex);
if (dorip && ripnotify)
    send(rippid, 0); /* send anything but TIMEOUT */
return;
}
```

The timer process (executing slowtimer) calls rttimer approximately once per second, passing in argument delta, the time that has elapsed since the last call. After waiting for the mutual exclusion semaphore, rttimer iterates through the routing table. For each entry, it traverses the linked list of routes, and examines each. For normal routes, rttimer decrements the time-to-live counter, and unlinks the node from the list if the counter reaches zero. However, if the gateway runs RIP, rttimer marks the expired route as having infinite cost, so it cannot be used for routing, and retains the expired route in the table for a short period^①. Finally, rttimer decrements the time-to-live counter on the default route.

6.7.1 Adding A Route

Network management software and routing information protocols call functions that add, delete, or change routes. For example, procedure rtadd adds a new route to the table.

```
/* rtadd.c - rtadd */

#include <conf.h>
#include <kernel.h>
#include <network.h>

struct route *rtnew();

/*
 * rtadd - add a route to the routing table
 */

```

^① Chapter 18 describes RIP and explains how it uses the routing table.



```
int rtadd(net, mask, gw, metric, intf, ttl)
IPAddr    net, mask, gw;
int      metric, intf, ttl;
{
    struct    route     *prt, *srt, *prev;
    Bool      isdup;
    int       hv, i, j;

    if (!Route.ri_valid)
        rtinit();

    prt = rtnew(net, mask, gw, metric, intf, ttl);
    if (prt == (struct route *)SYSERR)
        return SYSERR;

    /* compute the queue sort key for this route */
    for (prt->rt_key = 0, i=0; i<IP_ALEN; ++i)
        for (j=0; j<8; ++j)
            prt->rt_key += (mask[i] >> j) & 1;
    wait(Route.ri_mutex);

    /* special case for default routes */
    if (blkequ(net, RT_DEFAULT, IP_ALEN)) {
        if (Route.ri_default)
            RTFREE(Route.ri_default);
        Route.ri_default = prt;
        signal(Route.ri_mutex);
        return OK;
    }
    prev = NULL;
    hv = rthash(net);
    isdup = FALSE;
    for (srt=rtable[hv]; srt; srt = srt->rt_next) {
        if (prt->rt_key > srt->rt_key)
            break;
        if (blkequ(srt->rt_net, prt->rt_net, IP_ALEN) &&
            blkequ(srt->rt_mask, prt->rt_mask, IP_ALEN)) {
            isdup = TRUE;
            break;
        }
        prev = srt;
    }
}
```



```
}

if (isdup) {
    struct route *tmprt;

    if (blkequ(srt->rt_gw, prt->rt_gw, IP_ALEN)) {
        /* just update the existing route */
        if (dorip) {
            srt->rt_ttl = ttl;
            if (srt->rt_metric != metric) {
                if (metric == RTM_INF)
                    srt->rt_ttl = RIPZTIME;
                send(rippid, 0);
            }
        }
        srt->rt_metric = metric;
        RTFREE(prt);
        signal(Route.ri_mutex);
        return OK;
    }

    /* else, someone else has a route there... */
    if (srt->rt_metric <= prt->rt_metric) {
        /* no better off to change; drop the new one */

        RTFREE(prt);
        signal(Route.ri_mutex);
        return OK;
    } else if (dorip)
        send(rippid, 0);
    tmprt = srt;
    srt = srt->rt_next;
    RTFREE(tmprt);
} else if (dorip)
    send(rippid, 0);
prt->rt_next = srt;
if (prev)
    prev->rt_next = prt;
else
    rttable[hv] = prt;
signal(Route.ri_mutex);
return OK;
}
```



Rtadd calls procedure rtnew to allocate a new node and initialize the fields. It then checks for the default route as a special case. For non-default routes, rtadd uses rthash to compute the index in the routing table for the new route, and follows the linked list of routes starting at that location. Once it finds the position in the list at which the new route should be inserted, it checks to see if the list contains an existing route for the same destination. If so, rtadd compares the metrics for the old and new route to see if the new route is better, and discards the new route if it is not. Finally, rtadd either inserts the new node on the list or copies information into an existing node for the same address.

Procedure rtnew allocates and initializes a new routing table entry. It calls getbuf to allocate storage for the **new** node, and then fills in the header.

```
/* rtnew.c - rtnew */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * rtnew - create a route structure
 */
struct route *rtnew(net, mask, gw, metric, ifnum, ttl)
IPAddr net, mask, gw;
int metric, ifnum, ttl;
{
    struct route *prt;

    prt = (struct route *)getbuf(Route.ri_bpool);
    if (prt == (struct route *)SYSERR) {
        IpRoutingDiscards++;
        return (struct route *)SYSERR;
    }

    blkcopy(prt->rt_net, net, IP_ALEN);
    blkcopy(prt->rt_mask, mask, IP_ALEN);
    blkcopy(prt->rt_gw, gw, IP_ALEN);
    prt->rt_metric = metric;
    prt->rt_ifnum = ifnum;
    prt->rt_ttl = ttl;
    prt->rt_refcnt = 1; /* our caller */
    prt->rt_usecnt = 0;
```



```
    prt->rt_next = NULL;
    return prt;
}
```

6.7.2 Deleting A Route

Procedure rtdel takes a destination address as an argument and deletes the route to that destination by removing the node from the routing table.

```
/* rtdel.c - rtdel */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * rtdel - delete the route with the given net, mask
 */
int rtdel(net, mask)
IPAddr    net, mask;          /* destination network and mask */
{
    struct    route    *prt, *prev;
    int        hv, i;

    if (!Route.ri_valid)
        return SYSERR;
    wait(Route.ri_mutex);
    if (Route.ri_default &&
        blkequ(net, Route.ri_default->rt_net, IP_ALEN)) {
        RTFREE(Route.ri_default);
        Route.ri_default = 0;
        signal(Route.ri_mutex);
        return OK;
    }
    hv = rthash(net);

    prev = NULL;
    for (prt = rttable[hv]; prt; prt = prt->rt_next) {
        if (blkequ(net, prt->rt_net, IP_ALEN) &&
            blkequ(mask, prt->rt_mask, IP_ALEN))
            break;
        prev = prt;
    }
    if (prev)
        prev->rt_next = prt->rt_next;
    else
        rttable[hv] = prt->rt_next;
    RTFREE(prt);
}
```



```
    }

    if (prt == NULL) {
        signal(Route.ri_mutex);
        return SYSERR;
    }

    if (prev)
        prev->rt_next = prt->rt_next;
    else
        rttable[hv] = prt->rt_next;

    RTFREE(prt);
    signal(Route.ri_mutex);
    return OK;
}
```

As usual, the code checks for the default route as a special case. If no match occurs, rt~~d~~e~~l~~ hashes the destination address and searches the linked list of routes. Once it finds the correct route, rt~~d~~e~~l~~ unlinks the node from the linked list, and uses macro RTFREE to decrement the reference count. Recall that if the reference count reaches zero, RTFREE returns the node to the free list. If the reference count remains positive, some other process or processes must still be using the node; the node will be returned to the free list when the last of those processes decrements the reference count to zero.

Macro RTFREE assumes that the executing process has already obtained exclusive access to the routing table. Thus, it can be used in procedures like rt~~d~~e~~l~~. Arbitrary procedures that need to decrement the reference count on a route call procedure rtfree. When invoked, rtfree waits on the mutual exclusion semaphore, invokes macro RTFREE, and then signals the semaphore.

```
/* rtfree.c - rtfree */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 * rtfree - remove one reference to a route
 *-----*
 */
int rtfree(route *prt)
{
    if (!Route.ri_valid)
```



```
        return SYSERR;

    wait(Route.ri_mutex);
    RTFREE(prt);
    signal(Route.ri_mutex);
    return OK;
}
```

6.8 IP Options Processing

IP supports several options that control the way IP handles datagrams in hosts and gateways. To keep the example code simple and easy to understand, we have elected to omit option processing. However, the code contains a skeleton of two routines that scan options in the IP header. Gateways call procedure ipdoopts, which merely returns to its caller, leaving the options untouched in case the gateway forwards the datagram.

```
/* ipdoopts.c - ipdoopts */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ipdoopts - do gateway handling of IP options
 */
int ipdoopts(pni, pep)
struct netif *pni;
struct ep *pep;
{
    return OK; /* not implemented yet */
}
```

Hosts call procedure ipdstopts to handle options in arriving datagrams. Although our procedure does not implement option processing, it parses the option length octets and deletes the options field from the IP header.

```
/* ipdstopts.c - ipdstopts */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```



```
/*
 * ipdstopts - do host handling of IP options
 */
int ipdstopts(pni, pep)
struct netif *pni;
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    char *popt, *popend;
    int len;

    if (IP_HLEN(pip) == IPMHLEN)
        return OK;
    popt = pip->ip_data;
    popend = &pep->ep_data[IP_HLEN(pip)];

    /* NOTE: options not implemented yet */

    /* delete the options */
    len = pip->ip_len-IP_HLEN(pip); /* data length */
    if (len)
        blkcopy(pip->ip_data, &pep->ep_data[IP_HLEN(pip)], len);
    pip->ip_len = IPMHLEN + len;
    pip->ip_verlen = (pip->ip_verlen&0xf0) | IP_MINHLEN;
    return OK;
}
```

6.9 Summary

The IP routing table serves as a central data structure. When routing datagrams the IP process uses the routing table to find a next-hop route for the datagram's destination. Because route lookup must be performed frequently, the table is organized to make lookup efficient. Meanwhile, the high-level protocol software that learns about new routes will insert, delete, or change routes.

This chapter examined the procedures for both lookup and table maintenance. It showed how a routing table can use hashing to achieve efficiency, and how reference counts allow one process to use a route while another process deletes it concurrently.



6.10 FOR FURTHER STUDY

Postel [RFC 791] gives the standard for the Internet Protocol, Hornig [RFC 894] specifies the standard for the transmission of IP datagrams across an Ethernet, and Mogul and Postel et. al. [RFCs 950 and 940] discuss subnetting. Specific constants used throughout IP can be found in Reynolds and Postel [RFC 1010].

Braden and Postel [RFC 10091 provides a summary of how Internet gateways handle IP datagrams. Postel [RFC 791] describes IP option processing, and Su [RFC 781] comments on the timestamp option. Mills [RFC 981] considers multipath routing, while Braun [RFC 1104] discusses policy-based -routing.

6.11 EXERCISES

1. Consider the automatic initialization of the routing table by two processes at system startup. Is it possible for the two processes to interfere with one another? Explain.
2. The number of buckets used determines the efficiency of a bucket hashing scheme because it determines the average length of the linked lists. How much memory would be required to store 1000 routes if one wanted the average list to have no more than 3 entries?
3. What happens if procedure rtdel calls rtfree instead of using macro RTFREE?
4. ICMP redirect messages only allow gateways to specify destinations as host redirects or network redirects. How tan the code in this chapter help one deduce that an address is a subnet address?
5. Assume that in the next version of IP, all addresses are self-identifying (e.g., each address comes with a correct subnet mask). How would you redesign the routing table data structures to make them more efficient?
6. Consider the routing of broadcast datagrams (see netmatch). The code carefully distinguishes between locally-generated broadcasts and incoming broadcasts. Why?
7. The special case that arises when routing broadcast datagrams can be eliminated by adding an extra field to each route entry that specifies whether the entry should be used for inbound traffic, outbound traffic, or both. How does adding such a field make network management more difficult?
8. We said that implementing the local host interface as a pseudo-network helped eliminate special cases. How many times do routines in this chapter make an explicit test for the local machine?
9. Does it make sense to design a routing table that stores backup routes (i.e., a



table that keeps several routes to a given destination)? Explain.

10. Add type-of-service routing to the IP routing table in this chapter by allowing the route to be chosen as a function of the datagram's type of service, as well as its destination address.
11. Add security routing to the IP routing table in this chapter by allowing the route to be chosen as a function of the datagram's source address and protocol type as well as its destination address.



7 IP: Fragmentation And Reassembly

7.1 Introduction

This chapter examines software that fragments outgoing datagrams and reassembles incoming datagrams. Because the ultimate destination performs fragment reassembly, every computer using TCP/IP must include the code for reassembly, or it might not be able to communicate with all computers on its internet.

The protocol standard specifies that all implementations of IP must be able to fragment and reassemble datagrams. In practice, any gateway that connects two or more networks with different MTU sizes will fragment often. Because well-designed application software takes care to generate datagrams small enough to travel across directly connected networks, hosts do not need to perform fragmentation as frequently.

7.2 Fragmenting Datagrams

Fragmentation occurs after IP has routed a datagram and is about to deposit it on the queue associated with a given network interface. IP compares the datagram length to the network MTU to determine whether fragmentation is needed. In the simplest case, the entire datagram fits in a single network packet or frame, and will not need fragmentation.

For cases where fragmentation is required, IP creates multiple datagrams, each with the fragment bit set, and places consecutive pieces of data from the original datagram in them. It sets the more fragments bit in the IP header of all fragments from a datagram, except for the fragment that carries the final octets of data. As it constructs fragments, IP passes them to the network interface for transmission.



7.2.1 Fragmenting Fragments

Fragmentation becomes slightly more complex if the datagram being fragmented is already a fragment. Such cases can arise when a datagram passes through two or more gateways. If one gateway fragments the original datagram, the fragments themselves may be too large for a subsequent network along the path. Thus, a gateway may receive fragments that it must fragment into even smaller pieces.

The subtle distinction between datagram fragmentation and fragment fragmentation arises from the way a gateway must handle the more fragments bit. When a gateway fragments an original (unfragmented) datagram, it sets the more fragments bit on all but the final fragment. Similarly, if the more fragments bit is not set on a fragment, the gateway treats it exactly like an original datagram and sets the more fragments bit in every subfragment except the last. When a gateway fragments a nonfinal fragment, however, it sets the more fragments bit on all (sub)fragments it produces because none of them can be the final fragment for the entire datagram.

7.3 Implementation Of Fragmentation

In the example code, procedure `ippotp` makes the decision about fragmentation.

```
/* ippotp.c - ippotp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

-----*
 * ippotp - send a packet to an interface's output queue
 *-----
 */

int ippotp(inum, nh, pep)
int      inum;
IPAddr      nh;
struct    ep   *pep;
{
    struct    netif    *pni = &nif[inum];
    struct    ip     *pip;
    int       hlen, maxlen, tosend, offset, offindg;

    if (pni->ni_state == NIS_DOWN) {
        freebuf(pep);
        return SYSERR;
    }
}
```



```
}

pip = (struct ip *)pep->ep_data;
if (pip->ip_len <= pni->ni_mtu) {
    blkcopy(pep->ep_nexthop, nh, IP_ALEN);
    pip->ip_cksum = 0;
    iph2net(pip);
    pip->ip_cksum = cksum(pip, IP_HLEN(pip)/2);
    return netwrite(pni, pep, EP_HLEN+net2hs(pip->ip_len));
}

/* else, we need to fragment it */

if (pip->ip_fragoff & IP_DF) {
    IpFragFails++;
    icmp(ICK_DESTUR, ICC_FNADF, pip->ip_src, pep);
    return OK;
}

maxdlen = (pni->ni_mtu - IP_HLEN(pip)) &~ 7;
offset = 0;
offindg = (pip->ip_fragoff & IP_FRAGOFF)<<3;
tosend = pip->ip_len - IP_HLEN(pip);

while (tosend > maxdlen) {
    if (ipfsend(pni,nh,pep,offset,maxdlen,offindg) != OK) {
        IpOutDiscards++;
        freebuf(pep);
        return SYSERR;
    }
    IpFragCreates++;
    tosend -= maxdlen;
    offset += maxdlen;
    offindg += maxdlen;
}
IpFragOKs++;

IpFragCreates++;

hlen = ipfhcopy(pep, pep, offindg);
pip = (struct ip *)pep->ep_data;
/* slide the residual down */
blkcopy(&pep->ep_data[hlen], &pep->ep_data[IP_HLEN(pip)+offset],
       tosend);

/* keep MF, if this was a frag to start with */
pip->ip_fragoff = (pip->ip_fragoff & IP_MF)|(offindg>>3);
```



```
    pip->ip_len = tosend + hlen;
    pip->ip_cksum = 0;
    iph2net(pip);
    pip->ip_cksum = cksum(pip, hlen>>1);
    blkcopy(pep->ep_nexthop, nh, IP_ALEN);
    return netwrite(pni, pep, EP_HLEN+net2hs(pip->ip_len));
}
```

Arguments to ippputp give the interface number over which to route, the next-hop address, and a packet. If the packet length is less than the network MTU, ippputp calls netwrite to send the datagram and returns to its caller. If the datagram cannot be sent in one packet, ippputp divides the datagram into a sequence of fragments that each fit into one packet. To do so, ippputp computes the maximum possible fragment length, which must be a multiple of 8, and divides the datagram into a sequence of maximum-sized fragments plus a final fragment of whatever remains. Once it has computed a maximum fragment size, ippputp iterates through the datagram, calling procedure ipfsend to send each fragment.

The code contains a few subtleties. First, because each fragment must contain an IP header, the maximum amount of data that can be sent equals the MTU minus the IP header length, truncated to the nearest multiple of 8. Second, the iteration proceeds only while the data remaining in the datagram is strictly greater than the maximum that can be sent. Thus, the iteration will stop before sending the last fragment even in the case where all fragments happen to be of equal size. Third, to send the final fragment, ippputp modifies the original datagram and does not copy the fragment into a new buffer. Fourth, the more fragments (MF) bit is not usually set in the final fragment of a datagram. However, in the case where a gateway happens to further fragment a non-final fragment, it must leave MF set in all fragments.

7.3.1 Sending One Fragment

Procedure ipfsend creates and sends a single fragment. It allocates a new buffer for the copy, calls ipfhcopy to copy the header and IP options, copies the data for this fragment into the new datagram, and passes the result to netwrite.

```
/* ipfsend.c - ipfsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ipfsend - send one fragment of an IP datagram
```



```
*-----  
*/  
  
int ipfsend(pni, nexthop, pep, offset, maxlen, offindg)  
struct netif *pni;  
IPAddr nexthop;  
struct ep *pep;  
int offset, maxlen, offindg;  
{  
    struct ep *pepnew;  
    struct ip *pip, *pipnew;  
    int hlen, len;  
  
    pepnew = (struct ep *)getbuf(Net.netpool);  
    if (pepnew == (struct ep *)SYSERR)  
        return SYSERR;  
    hlen = ipfhcopy(pepnew, pep, offindg); /* copy the headers */  
  
    pip = (struct ip *)pep->ep_data;  
    pipnew = (struct ip *)pepnew->ep_data;  
    pipnew->ip_fragoff = IP_MF | (offindg>>3);  
    pipnew->ip_len = len = maxlen + hlen;  
    pipnew->ip_cksum = 0;  
  
    iph2net(pipnew);  
    pipnew->ip_cksum = cksum(pipnew, hlen>>1);  
  
    blkcopy(&pepnew->ep_data[hlen],  
            &pep->ep_data[IP_HLEN(pip)+offset], maxlen);  
    blkcopy(pepnew->ep_nexthop, nexthop, IP_ALEN);  
  
    return netwrite(pni, pepnew, EP_HLEN+len);  
}
```

7.3.2 Copying A Datagram Header

Procedure ipfhcopy copies a datagram header. Much of the code is concerned with the details of IP options. According to the protocol standard, some options should only appear in the first fragment, while others must appear in all fragments. Ipfhcopy iterates through the options, and examines each to see whether it should be copied into all fragments. Finally, when ipfhcopy returns, ipfsend calls netwrite to send the fragment.

```
/* ipfhcopy.c - ipfhcopy */
```



```
#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ipfhcopy - copy the hardware, IP header, and options for a fragment
 */
int ipfhcopy(pepto, pepfrom, offindg)
struct ep *pepto, *pepfrom;
{
    struct ip *pipfrom = (struct ip *)pepfrom->ep_data;
    unsigned i, maxhlen, olen, otype;
    unsigned hlen = (IP_MINHLEN<<2);

    if (offindg == 0) {
        blkcopy(pepto, pepfrom, EP_HLEN+IP_HLEN(pipfrom));
        return IP_HLEN(pipfrom);
    }
    blkcopy(pepto, pepfrom, EP_HLEN+hlen);

    /* copy options */

    maxhlen = IP_HLEN(pipfrom);
    i = hlen;
    while (i < maxhlen) {
        otype = pepfrom->ep_data[i];
        olen = pepfrom->ep_data[++i];
        if (otype & IPO_COPY) {
            blkcopy(&pepto->ep_data[hlen],
                    pepfrom->ep_data[i-1], olen);
            hlen += olen;
        } else if (otype == IPO_NOP || otype == IPO_EOOP) {
            pepfrom->ep_data[hlen++] = otype;
            olen = 1;
        }
        i += olen-1;

        if (otype == IPO_EOOP)
            break;
    }
}
```



```
/* pad to a multiple of 4 octets */
while (hlen % 4)
    pepto->ep_data[hlen++] = IPO_NOP;
return hlen;
}
```

7.4 Datagram Reassembly

Reassembly requires IP on the receiving machine to accumulate incoming fragments until a complete datagram can be reassembled. Once reassembled, IP routes the datagram on toward its destination. Because IP does not guarantee order of delivery, the protocol requires IP to accept fragments that arrive out of order or after delay. Furthermore, fragments for a given datagram may arrive **intermixed** with fragments from other datagrams.

7.4.1 Data Structures

To make the implementation efficient, the data structure used to store fragments must permit: quick location of the group of fragments that comprise a given datagram, fast insertion of a new fragment into a group, efficient test of whether a complete datagram has arrived, timeout of fragments, and eventual removal of fragments if the timer expires before reassembly can be completed.

Our example code uses an array of lists to store fragments. Each item in the array corresponds to a single datagram for which one or more fragments have arrived, and contains a pointer to a list of fragments for that datagram. File ipreass.h declares the data structures.

```
/* ipreass.h */

/* Internet Protocol (IP) reassembly support */

#define IP_FQSIZE 10 /* max number of frag queues */
#define IP_MAXNF 10 /* max number of frags/datagram */
#define IP_FTTL 60 /* time to live (secs)

/* ipf_state flags */

#define IPFF_VALID 1 /* contents are valid */
#define IPFF_BOGUS 2 /* drop frags that match */
#define IPFF_FREE 3 /* this queue is free to be allocated

struct ipfq {
```



```
char ipf_state;           /* VALID, FREE or BOGUS      */
IPAddr    ipf_src;        /* IP address of the source   */
short     ipf_id;         /* datagram id                 */
int      ipf_ttl;         /* countdown to disposal      */
int      ipf_q;          /* the queue of fragments     */
};

extern  int ipfmutex;      /* mutex for ipfq[]          */
extern  struct ipfq ipfq[]; /* IP frag queue table       */
```

Array ipfq forms the main data structure for fragments: each entry in the array corresponds to a single datagram. Structure ipfq defines the information kept. In addition to the datagram source address and identification fields (ipf_src and ipf_id), the entry contains a time-to-live counter (ipf_ttl) that specifies how long (in seconds) before the entry will expire if not all fragments arrive. Field ipf_q points to a linked list of all fragments that have arrived for the datagram.

Reassembly software must test whether all fragments have arrived for a given datagram. To make the test efficient, each fragment list is stored in sorted order. In particular, the fragments on a given list are ordered by their offset in the original datagram. The protocol design makes the choice of sort key easy because even fragmented fragments have offsets measured from the original datagram. Thus, it is possible to insert any fragment in the list without knowing whether it resulted from a single fragmentation or multiple fragmentations.

7.4.2 Mutual Exclusion

To guarantee that processes do not interfere with one another while accessing the list of fragments, the reassembly code uses a single mutual exclusion semaphore, ipfmutex. File ipreas.h declares the value to be an external integer, accessible to all the code. As we will see, mutual exclusion is particularly important because it allows the system to use separate processes for timeout and reassembly.

7.4.3 Adding A Fragment To A List

IP uses information in the header of an incoming fragment to identify the appropriate list. Fragments belong to the same datagram if they have identical values in both their source address and IP identification fields. Procedure ipreas takes a fragment, finds the appropriate list, and adds the fragment to the list. Given a fragment, it searches the fragment table to see if it contains an existing entry for the datagram to which the fragment belongs. At each entry, it compares the source and identification fields, and calls ipfadd to add the fragment to the list if it finds a match. It then calls ipfjoin to see if



all fragments can be reassembled into a datagram. If no match is found, ipreass allocates the first unused entry in the array, copies in the source and identification fields, and places the fragment on a newly allocated queue.

Our implementation uses a linear search to locate the appropriate list for an incoming fragment, and may seem too inefficient for production use. Of course, some computers do receive fragments from many datagrams simultaneously and will require a faster search method. However, because most computers communicate frequently with machines in the local environment, they rarely receive fragments. Furthermore, because reassembly only happens for datagrams destined for the local machine and not for transit traffic, gateways do not need to reassemble datagrams as fast as they need to route them. So, for typical computer systems, a linear search suffices.

```
/* ipreass.c - ipreass */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <q.h>

struct ep *ipfjoin();

/*-----
 * ipreass - reassemble an IP datagram, if necessary
 * returns packet, if complete; 0 otherwise
 *-----
 */
struct ep *ipreass(pep)
struct ep *pep;
{
    struct ep *pep2;
    struct ip *pip;
    int firstfree;
    int i;

    pip = (struct ip *)pep->ep_data;

    wait(ipfmutex);

    if ((pip->ip_fragoff & (IP_FRAGOFF|IP_MF)) == 0) {
        signal(ipfmutex);
        return pep;
    }
}
```



```
IpReasmReqds++;
firstfree = -1;
for (i=0; i<IP_FQSIZE; ++i) {
    struct ipfq *piq = &ipfqt[i];

    if (piq->ipf_state == IPFF_FREE) {
        if (firstfree == -1)
            firstfree = i;
        continue;
    }
    if (piq->ipf_id != pip->ip_id)
        continue;
    if (!blkequ(piq->ipf_src, pip->ip_src, IP_ALEN))
        continue;
    /* found a match */
    if (ipfadd(piq, pep) == 0) {
        signal(ipfmutex);
        return 0;
    }
    pep2 = ipfjoin(piq);
    signal(ipfmutex);
    return pep2;

}
/* no match */

if (firstfree < 0) {
    /* no room-- drop */
    freebuf(pep);
    signal(ipfmutex);
    return 0;
}
ipfqt[firstfree].ipf_q = newq(IP_FQSIZE, QF_WAIT);
if (ipfqt[firstfree].ipf_q < 0) {
    freebuf(pep);
    signal(ipfmutex);
    return 0;
}
blkcopy(ipfqt[firstfree].ipf_src, pip->ip_src, IP_ALEN);
ipfqt[firstfree].ipf_id = pip->ip_id;
ipfqt[firstfree].ipf_ttl = IP_FTTL;
```



```
    ipfqt[firstfree].ipf_state = IPFF_VALID;
    ipfadd(&ipfqt[firstfree], pep);
    signal(ipfmutex);
    return 0;
}

int ipfmutex;
struct ipfq ipfqt[IP_FQSIZE];
```

7.4.4 Discarding During Overflow

Procedure ipfadd inserts a fragment on a given list. For the normal case, the procedure is trivial; ipfadd merely calls enq to enqueue the fragment and resets the time-to-live field for the datagram.

In the case where the fragment list has reached its capacity, the new fragment cannot be added to the list. When that occurs, ipfadd discards all fragments that correspond to the datagram, and frees the entry in array ipfqt. At first this may seem strange. However, the reason for discarding the entire list is simple: a single missing fragment will prevent IP from ever reassembling and processing the datagram, so freeing the memory used by the remaining fragments may make it possible to complete other datagrams. Furthermore, once the list reaches capacity, it cannot grow. Therefore, keeping the list consumes memory resources but does not contribute to the success of reassembling the datagram.

```
/* ipfadd.c - ipfadd */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*
 *-----*
 * ipfadd - add a fragment to an IP fragment queue
 *-----*
 */
Bool ipfadd(iq, pep)
struct ipfq *iq;
struct ep *pep;
{
    struct ip *pip;
    int fragoff;
```



```
if (iq->ipf_state != IPFF_VALID) {
    freebuf(pep);
    return FALSE;
}

pip = (struct ip *)pep->ep_data;
fragoff = pip->ip_fragoff & IP_FRAGOFF;

if (enq(iq->ipf_q, pep, -fragoff) < 0) {
    /* overflow-- free all frags and drop */
    freebuf(pep);
    IpReasmFails++;
    while (pep = (struct ep *)deq(iq->ipf_q)) {
        freebuf(pep);
        IpReasmFails++;
    }
    freeq(iq->ipf_q);
    iq->ipf_state = IPFF_BOGUS;
    return FALSE;
}
iq->ipf_ttl = IP_FTTL;      /* restart timer */
return TRUE;
}
```

7.4.5 Testing For A Complete Datagram

When adding a new fragment to a list, IP must check to see if it has all the fragments that comprise a datagram. Procedure ipfjoin examines a list of fragments to see if they form a complete datagram.

```
/* ipfjoin.c - ipfjoin */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

struct    ep    *ipfcons();

/*-----
 * ipfjoin - join fragments, if all collected
 *-----
 */
struct ep *ipfjoin(iq)
```



```
struct    ipfq *iq;
{
    struct    ep    *pep;
    struct    ip    *pip;
    int       off, packoff;

    if (iq->ipf_state == IPFF_BOGUS)
        return 0;
    /* see if we have the whole datagram */

    off = 0;
    while (pep=(struct ep *)seeq(iq->ipf_q)) {
        pip = (struct ip *)pep->ep_data;
        packoff = (pip->ip_fragoff & IP_FRAGOFF)<<3;
        if (off < packoff) {
            while(seeq(iq->ipf_q))
                /*empty*/;
            return 0;
        }
        off = packoff + pip->ip_len - IP_HLEN(pip);
    }
    if (off > MAXLRGBUF) {      /* too big for us to handle */
        while (pep = (struct ep *)deq(iq->ipf_q))
            freebuf(pep);
        freeq(iq->ipf_q);
        iq->ipf_state = IPFF_FREE;
        return 0;
    }
    if ((pip->ip_fragoff & IP_MF) == 0)
        return ipfcons(iq);

    return 0;
}
```

After verifying that the specified fragment list is in use, ipfjoin enters a loop that iterates through the fragments. It starts variable off at zero, and uses it to see if the current fragment occurs at the expected location in the datagram. First, ipfjoin checks to see that the offset in the current fragment matches off. If the offset of the current fragment exceeds off, there must be a missing fragment, so ipfjoin returns zero (which means that the fragments cannot be joined). If the fragment matches, ipfjoin computes the expected offset of the next fragment by adding the current fragment length to off.



Once ipfjoin verifies that all fragments have been collected, it tests to make sure the datagram will fit into a large buffer. The software can only handle datagrams that fit into large buffers because the datagram must be reassembled into contiguous memory before it can be passed to an application program. Thus, if the datagram cannot fit into a single buffer, ipfjoin discards the fragments. Finally, for datagrams that do fit, ipfjoin calls ipfcons to collect the fragments and rebuild a complete datagrams

7.4.6 Building A Datagram From Fragments

Procedure ipfcons reassembles fragments into a complete datagram. In addition to copying the data from each fragment into place, it builds a valid datagram header. Information for the datagram header comes from the header in the first fragment, modified to reflect the full datagram's size. Ipfcons turns off the fragment bit to show that the reconstructed datagram is not a fragment and sets the offset field to zero. If it reassembles the datagram, ipfcons releases the buffers that hold individual fragments. When it finishes reassembly, ipfcons releases the entry in the fragment table ipfq.

```
/* ipfcons.c - ipfcons */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ipfcons - construct a single packet from an IP fragment queue
 */
struct ep *ipfcons(iq)
struct ipfq *iq;
{
    struct ep   *pep, *peptmp;
    struct ip   *pip;
    int       off, seq;

    pep = (struct ep *)getbuf(Net.lrgpool);
    if (pep == (struct ep *)SYSERR) {
        while (peptmp = (struct ep *)deq(iq->ipf_q)) {
            IpReasmFails++;
            freebuf(peptmp);
        }
        freeq(iq->ipf_q);
        iq->ipf_state = IPFF_FREE;
        return 0;
    }
}
```



```
}

/* copy the Ether and IP headers */

peptmp = (struct ep *)deq(iq->ipf_q);
pip = (struct ip *)peptmp->ep_data;
off = IP_HLEN(pip);
seq = 0;
blkcopy(pep, peptmp, EP_HLEN+off);

/* copy the data */
while (peptmp != 0) {
    int dlen, doff;

    pip = (struct ip *)peptmp->ep_data;
    doff = IP_HLEN(pip) + seq
        - ((pip->ip_fragoff&IP_FRAGOFF)<<3);
    dlen = pip->ip_len - doff;
    blkcopy(pep->ep_data+off, peptmp->ep_data+doff, dlen);
    off += dlen;
    seq += dlen;
    freebuf(peptmp);
    peptmp = (struct ep *)deq(iq->ipf_q);
}

/* fix the large packet header */
pip = (struct ip *)pep->ep_data;
pip->ip_len = off;
pip->ip_fragoff = 0;

/* release resources */
freeq(iq->ipf_q);
iq->ipf_state = IPFF_FREE;
IpReasmOKs++;
return pep;
}
```

7.5 Maintenance Of Fragment Lists

Because IP is an unreliable delivery mechanism, datagrams can be lost as they traverse an internet. If a fragment is lost, the IP software on the receiving end cannot reassemble the original datagram. Furthermore, because IP does not provide an



acknowledgement facility, no fragment retransmissions are possible. Thus, once a fragment is lost, IP will never recover the datagram to which it belonged. Instead, higher-level protocols, like TCP, use a new datagram to retransmit^①.

To keep lost fragments from consuming memory resources and to keep IP from becoming confused by reuse of the identification field, IP must periodically check the fragment lists and discard an old list when reception of the remaining fragments is unlikely. Procedure ipftimer performs the periodic sweep.

```
/* ipftimer.c - ipftimer */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 * ipftimer - update time-to-live fields and delete expired fragments
 *-----*
 */
void ipftimer(gran)
int gran;           /* granularity of this run */
{
    struct ep *pep;
    struct ip *pip;
    int i;

    wait(ipfmutex);
    for (i=0; i<IP_FQSIZE; ++i) {
        struct ipfq *iq = &ipfq[i];

        if (iq->ipf_state == IPFF_FREE)
            continue;
        iq->ipf_ttl -= gran;
        if (iq->ipf_ttl <= 0) {
            if (iq->ipf_state == IPFF_BOGUS) {
                /* resources already gone */
                iq->ipf_state = IPFF_FREE;
                continue;
            }
            if (pep = (struct ep *)deq(iq->ipf_q)) {

```

^① Each retransmission of a TCP segment uses a datagram that has a unique IP identification, so IP cannot intermix fragments from two transmissions when reassembling.



```
    IpReasmFails++;

    pip = (struct ip *)pep->ep_data;
    icmp(ICK_TIMEX, ICC_FTIMEX,
         pip->ip_src, pep);
}

while (pep = (struct ep *)deq(iq->ipf_q)) {
    IpReasmFails++;
    freebuf(pep);
}
freeq(iq->ipf_q);
iq->ipf_state = IPFF_FREE;
}

signal(ipfmutex);
}
```

Iptimer iterates through the fragment lists each time it is called (usually once per second). It decrements the time-to-live field in each entry and discards the list if the timer reaches zero. When discarding a list, iptimer extracts the first node, and uses the packet buffer to send an ICMP time exceeded message back to the source. After sending the ICMP message, iptimer frees the list of fragments and marks the entry in ipfq free for use again,

7.6 Initialization

Initialization of the data structures used for fragment reassembly is trivial. Procedure ipfinit creates the mutual exclusion semaphore and marks each entry in the fragment array available for use.

```
/* ipfinit.c - ipfinit */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ipfinit - initialize IP fragment queue data structures
 */
void ipfinit()
{
```



```
int i;

ipfmutex = screate(1);
for (i=0; i<IP_FQSIZE; ++i)
    ipfqt[i].ipf_state = IPFF_FREE;
}
```

7.7 Summary

All machines that implement IP must be able to fragment outgoing datagrams and to reassemble fragmented datagrams that arrive.

In practice, gateways usually fragment datagrams when they encounter a datagram that is too large for the network MTU over which it must travel. Fragmentation consists of duplicating the datagram header for each fragment, setting the offset and fragment bits, copying part of the data, and sending the resulting fragments one at a time. The software fragments a datagram after IP routes it, but before IP deposits it on the output queue associated with a particular network interface. Compared to reassembly, fragmentation is straightforward. To perform reassembly, IP uses a data structure that collects together fragments from a given datagram. Once all fragments have been collected, the datagram can be reassembled (reconstructed) and processed.

Reassembly works in parallel with a maintenance process. Each time a new fragment arrives for a datagram, IP resets the time-to-live field in the fragment table for that datagram. The separate maintenance process periodically checks the lists of fragments and decrements the time-to-live field in each entry. If the time-to-live reaches zero before all fragments arrive, the maintenance process discards the entire datagram.

7.8 FOR FURTHER STUDY

Many textbooks describe algorithms and data structures that apply to storage of linked lists. More information on fragment management can be found in the IP specification [RFC 791] and the host requirements document [RFC 1122],

7.9 EXERCISES

1. Read the IP specification carefully. Can two fragments from different datagrams ever have the same value for IP source and identification fields? Explain. (Hint: consider machine reboot.)
2. Look carefully at ippup and ipfhcopy. Can ippup ever underestimate the maximum size fragment that can be sent? Why or why not?



3. The example code chooses the maximum possible fragment size and divides a datagram into many pieces of that size followed by an odd piece. Is there any advantage to making all fragments as close to the same size as possible? Explain.
4. Procedure ipreasss assigns each newly created fragment list a fixed value for time-to-live. Is there a better way to choose an initial time-to-live value? Explain.
5. Modify the fragment data structure to use hashing instead of sequential lookup and measure the improvement in performance. What can you conclude? Under what circumstances will hashing save time?
6. Use the ping command to generate datagrams of various sizes destined for a remote machine. See if you can detect the threshold of fragmentation from a discontinuity in the round trip delay. What does the result tell you about fragmentation cost?
7. Read the IP specification carefully. Does the example code correctly handle the do not fragment bit? Explain.
8. Consider a network capable of accepting 1000 datagrams per second. What constraint does such a network place on the choice of a fragment time-to-live (assuming IP uses a constant timeout for all fragments)?
9. What are the advantages and disadvantages of resetting the time-to-live for a datagram whenever a fragment arrives, as opposed to setting the timer once when the first fragment arrives?



8 IP: Error Processing (ICMP)

8.1 Introduction

The Internet Control Message Protocol (ICMP) is an integral part of IP that provides error reporting. ICMP handles several types of error conditions and always reports errors back to the original source. Any computer using IP must accept ICMP messages and change behavior in response to the reported error. Gateways must also be prepared to generate ICMP error messages when incoming datagrams cause problems.

This chapter reviews the details of ICMP processing. It shows code for generating error messages as well as the code for handling such messages when they arrive.

8.2 ICMP Message Formats

Unlike protocols that have a fixed message format, ICMP messages are type-dependent. The number of fields in a message, the interpretation of each field, and the amount of data the message carries depend on the message type.

8.3 Implementation Of ICMP Messages

File icmp.h, shown below, contains the declarations used for ICMP error messages. Type-dependent messages make the declaration of ICMP message formats more complex than those of other protocols. Structure icmp defines the message format. All ICMP messages begin with a fixed header, defined by fields ic_type (message type), ic_code (message subtype), and ic_cksum (message checksum). The next 32 bits in an ICMP message depend on the message type, and are declared in C using a union. In ICMP echo requests and replies, the message contains a 16-bit identification and 16-bit sequence number. In an ICMP redirect, the 32 bits specify the IP address of a gateway. In parameter problem messages, the 32 bits contain an 8-bit pointer and three octets of padding. In other messages, the 32 bits contain zeroes. Finally, field ic_data defines the data area of an ICMP message. As with the protocols we have seen earlier, the structure



only declares the first octet of data even though a message will contain multiple octets of data.

In addition to symbolic constants needed for all ICMP messages, icmp.h defines abbreviations that can be used to refer to short names in the union. For example, using an abbreviation, a programmer can specify the gateway address subfield using something.ic_gw instead of the fully qualified something.icu.ic2_gw.

```
/* icmp.h */

/* Internet Control Message Protocol Constants and Packet Format */

/* ic_type field */
#define ICT_ECHORP    0   /* Echo reply           */
#define ICT_DESTUR    3   /* Destination unreachable */
#define ICT_SRCQ 4   /* Source quench        */
#define ICT_REDIRECT 5   /* Redirect message type */
#define ICT_ECHORQ    8   /* Echo request          */
#define ICT_TIMEX11 11   /* Time exceeded         */
#define ICT_PARAMP    12  /* Parameter Problem     */
#define ICT_TIMERQ    13  /* Timestamp request     */
#define ICT_TIMERP    14  /* Timestamp reply       */
#define ICT_INFORQ    15  /* Information request   */
#define ICT_INFORP    16  /* Information reply     */
#define ICT_MASKRQ    17  /* Mask request          */
#define ICT_MASKRP    18  /* Mask reply            */

/* ic_code field */
#define ICC_NETUR 0    /* dest unreachable, net unreachable */
#define ICC_HOSTUR 1    /* dest unreachable, host unreachable */
#define ICC_PROTOUR 2   /* dest unreachable, proto unreachable */
#define ICC_PORTUR 3    /* dest unreachable, port unreachable */
#define ICC_FNADF 4    /* dest unr, frag needed & don't frag */
#define ICC_SRCRT 5    /* dest unreachable, src route failed */

#define ICC_NETRD 0    /* redirect: net          */
#define ICC_HOSTRD 1    /* redirect: host          */
#define IC_TOSNRD 2    /* redirect: type of service, net */
#define IC_TOSH RD 3   /* redirect: type of service, host */

#define ICC_TIMEX 0    /* time exceeded, ttl      */
#define ICC_FTIMEX 1    /* time exceeded, frag      */
```



```
#define IC_HLEN      8      /* octets          */
#define IC_PADLEN 3     /* pad length (octets)      */

#define IC_RDTTL 300   /* ttl for redirect routes      */

/* ICMP packet format (following the IP header)          */

struct icmp {           /* ICMP packet          */
    char ic_type;        /* type of message (ICT_* above) */
    char ic_code;        /* code (ICC_* above)      */
    short ic_cksum;      /* checksum of ICMP header+data */
}

union {
    struct {
        short icl_id;    /* for echo type, a message id */
        short icl_seq;   /* for echo type, a seq. number */
    } icl;
    IPAddr ic2_gw;       /* for redirect, gateway */
    struct {
        char ic3_ptr; /* pointer, for ICT_PARAMP */
        char ic3_pad[IC_PADLEN];
    } ic3;
    int ic4_mbz; /* must be zero */
} icu;
char ic_data[1]; /* data area of ICMP message */

/* format 1 */
#define ic_id    icu.icl.icl_id
#define ic_seq   icu.icl.icl_seq

/* format 2 */
#define ic_gw    icu.ic2_gw

/* format 3 */
#define ic_ptr   icu.ic3.ic3_ptr
#define ic_pad   icu.ic3.ic3_pad

/* format 4 */
#define ic_mbz  icu.ic4_mbz
```



8.4 Handling Incoming ICMP Messages

When an IP datagram carrying an ICMP message arrives destined for the local machine, the IP process passes it to procedure icmp_in.

```
/* icmp_in.c - icmp_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * icmp_in - handle ICMP packet coming in from the network
 */
int icmp_in(pni, pep)
struct netif      *pni;           /* not used */
struct ep     *pep;
{
    struct ip   *pip;
    struct icmp *pic;
    int       i, len;

    pip = (struct ip *)pep->ep_data;
    pic = (struct icmp *) pip->ip_data;

    len = pip->ip_len - IP_HLEN(pip);
    if (cksum(pic, len>>1)) {
        IcmpInErrors++;
        freebuf(pep);
        return SYSERR;
    }
    IcmpInMsgs++;
    switch(pic->ic_type) {
    case ICT_ECHORQ:
        IcmpInEchos++;
        return icmp(ICT_ECHORP, 0, pip->ip_src, pep, 0);
    case ICT_MASKRQ:
        IcmpInAddrMasks++;
        if (!gateway) {
            freebuf(pep);
            return OK;
        }
    }
```



```
        }

pic->ic_type = (char) ICT_MASKRP;

netmask(pic->ic_data, pip->ip_dst);

break;

case ICT_MASKRP:

    IcmpInAddrMaskReps++;

    for (i=0; i<Net.nif; ++i)

        if (blkequ(nif[i].ni_ip, pip->ip_dst, IP_ALEN))

            break;

    if (i != Net.nif) {

        setmask(i, pic->ic_data);

        send(pic->ic_id, ICT_MASKRP);

    }

    freebuf(pep);

    return OK;

case ICT_ECHORP:

    IcmpInEchoReps++;

    if (send(pic->ic_id, pep) != OK)

        freebuf(pep);

    return OK;

case ICT_REDIRECT:

    IcmpInRedirects++;

    icredirect(pep);

    return OK;

case ICT_DESTUR: IcmpInDestUnreachs++;   freebuf(pep); return OK;

case ICT_SRCQ:     IcmpInSrcQuenches++;   freebuf(pep); return OK;

case ICT_TIMEX:    IcmpInTimeExcds++;    freebuf(pep); return OK;

case ICT_PARAMP:   IcmpInParmProbs++;   freebuf(pep); return OK;

case ICT_TIMERQ:   IcmpInTimestamps++;   freebuf(pep); return OK;

case ICT_TIMESTAMP: IcmpInTimestampReps++; freebuf(pep); return OK;

default:

    IcmpInErrors++;

    freebuf(pep);

    return OK;

}

icsetsrc(pip);

len = pip->ip_len - IP_HLEN(pip);

pic->ic_cksum = 0;

pic->ic_cksum = cksum(pic, len>>1);
```



```
IcmpOutMsgs++;
ipsend(pip->ip_dst, pep, len, IPT_ICMP, IPP_INCTL, IP_TTL);
return OK;
}
```

The second argument to icmp_in is a pointer to a buffer that contains an IP datagram. Icmp_in locates the ICMP message in the datagram, and uses the ICMP type field to select one of six ICMP message types. The code handles each type separately.

To handle an ICMP echo request message, icmp_in calls icmp (discussed below) to generate an ICMP echo reply message. By contrast, to handle an ICMP echo reply message, ICMP extracts the message id field, assumes it is the process id of the process that sent the echo request, and sends the reply packet to that process.

In response to an ICMP address mask request, icmp_in changes the message to an address mask reply, uses netmask to find the appropriate subnet mask, and breaks out of the switch statement to send the reply.

For an ICMP address mask reply, icmp_in iterates through the interfaces until it finds one that matches the network address in the reply packet, and then calls procedure setmask (shown below) to set the subnet mask for that interface. It passes setmask the subnet mask found in the reply.

Icmp_in calls procedure icredirect to handle an incoming ICMP redirect message. The next section shows how icredirect changes the routing table.

In all cases, even for ICMP messages that it does not handle, icmp_in accumulates a count of incoming messages. As later chapters show, SNMP uses these counts.

8.5 Handling An ICMP Redirect Message

Procedure icredirect handles a request to change a route.

```
/* icredirect.c - icredirect */

#include <conf.h>
#include <kernel.h>
#include <network.h>

struct    route    *rtget();

/*
 * icredirect - handle an incoming ICMP redirect
 */
-----
```



```
*/  
  
int icredirect(pep)  
struct    ep    *pep;  
{  
    struct    route    *prt;  
    struct    ip     *pip, *pip2;  
    struct    icmp   *pic;  
    IPAddr      mask;  
  
    pip = (struct ip *)pep->ep_data;  
    pic = (struct icmp *)pip->ip_data;  
    pip2 = (struct ip *)pic->ic_data;  
  
    if (pic->ic_code == ICC_HOSTRD)  
        blkcopy(mask, ip_maskall, IP_ALEN);  
    else  
        netmask(mask, pip2->ip_dst);  
    prt = rtget(pip2->ip_dst, RTF_LOCAL);  
    if (prt == 0) {  
        freebuf(pep);  
        return OK;  
    }  
    if (blkque(pip->ip_src, prt->rt_gw, IP_ALEN)) {  
        rtdel(pip2->ip_dst, mask);  
        rtadd(pip2->ip_dst, mask, pic->ic_gw, prt->rt_metric,  
              prt->rt_ifnum, IC_RDTTL);  
    }  
    rtfree(prt);  
    freebuf(pep);  
    return OK;  
}
```

Icrediret extracts the specified destination address from the redirect message, calls netmask to compute the appropriate subnet mask, and uses rtget to look up the existing route. If the current route points to the gateway that sent the redirect message, icredirect deletes the existing route, and adds a new route that uses the new gateway specified in the redirect message.



8.6 Setting A Subnet Mask

When icmp_in receives a subnet mask reply, it calls procedure setmask to record the subnet mask in the network interface structure.

```
/* setmask.c - setmask */

#include <conf.h>
#include <kernel.h>
#include <network.h>

extern    int  bsdbrc;           /* use Berkeley (all-0's) broadcast   */

/*
 *  setmask - set the net mask for an interface
 */
int setmask(inum, mask)
int  inum;
IPAddr  mask;
{
    IPAddr  aobrc;           /* all 1's broadcast */
    IPAddr  defmask;
    int  i;

    if (nif[inum].ni_svalid) {
        /* one set already-- fix things */

        rtdel(nif[inum].ni_subnet, nif[inum].ni_mask);
        rtdel(nif[inum].ni_brc, ip_maskall);
        rtdel(nif[inum].ni_subnet, ip_maskall);
    }
    blkcopy(nif[inum].ni_mask, mask, IP_ALEN);
    nif[inum].ni_svalid = TRUE;
    netmask(defmask, nif[inum].ni_ip);

    for (i=0; i<IP_ALEN; ++i) {
        nif[inum].ni_subnet[i] =
            nif[inum].ni_ip[i] & nif[inum].ni_mask[i];
        if (bsdbrc) {
            nif[inum].ni_brc[i] = nif[inum].ni_subnet[i];
            aobrc[i] = nif[inum].ni_subnet[i] |
                ~nif[inum].ni_mask[i];
        }
    }
}
```



```
    } else
        nif[inum].ni_brc[i] = nif[inum].ni_subnet[i] |
            ~nif[inum].ni_mask[i];
    /* set network (not subnet) broadcast */
    nif[inum].ni_nbrc[i] =
        nif[inum].ni_ip[i] | ~defmask[i];
}

/* install routes */
/* net */
rtadd(nif[inum].ni_subnet, nif[inum].ni_mask, nif[inum].ni_ip,
    0, inum, RT_INF);
if (bsdbrc)
    rtadd(aobrc, ip_maskall, nif[inum].ni_ip, 0,
        NI_LOCAL, RT_INF);
else /* broadcast (all 1's) */
    rtadd(nif[inum].ni_brc, ip_maskall, nif[inum].ni_ip, 0,
        NI_LOCAL, RT_INF);
/* broadcast (all 0's) */
rtadd(nif[inum].ni_subnet, ip_maskall, nif[inum].ni_ip, 0,
    NI_LOCAL, RT_INF);
return OK;
}

IPAddr ip_maskall = { 255, 255, 255, 255 };
```

Because changing the subnet mask should also change routes that correspond to the network address, setmask begins by calling rtDEL to delete existing routes for the current interface address, broadcast address, and subnet broadcast address. It then copies the new subnet mask to field ni_mask, and sets ni_svalid to TRUE.

After the new mask has been recorded, setmask computes a new subnet address and subnet broadcast address for the interface. Finally, it calls rtADD to install new routes to the subnet and subnet broadcast addresses.

8.7 Choosing A Source Address For An ICMP Packet

For those cases that require a reply (e.g., ICMP echo request), ICMP must reverse the datagram source and destination addresses. To do so, procedure icmp, shown below, calls icsetsrc.

```
/* icsetsrc.c - icsetsrc */
```



```
#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 * icsetsrc - set the source address on an ICMP packet
 *-----
 */
void icsetsrc(pip)
struct ip *pip;
{
    int i;

    for (i=0; i<Net.nif; ++i) {
        if (i == NI_LOCAL)
            continue;
        if (netmatch(pip->ip_dst,nif[i].ni_ip,nif[i].ni_mask,0))
            break;
    }
    if (i == Net.nif)
        blkcopy(pip->ip_src, ip_anyaddr, IP_ALEN);
    else
        blkcopy(pip->ip_src, nif[i].ni_ip, IP_ALEN);
}
```

Icsetsrc iterates through each network interface and compares the network or subnet IP address associated with that interface to the destination IP address of the ICMP message. If it finds a match, icsetsrc copies the local machine address for that interface network into the source field of the datagram. In the event that no match can be found, icsetsrc fills the datagram source field with ip_anyaddr (all 0's), allowing the routing routines to replace it with the address of the interface over which it is routed.

8.8 Generating ICMP Error Messages

Gateways generate ICMP error messages in response to congestion, time-to-live expiration, and other error conditions. They call procedure icmp to create and send one message.

```
/* icmp.c - icmp */
```



```
#include <conf.h>
#include <kernel.h>
#include <network.h>

struct    ep    *icsetbuf();

/*
 * ICT_REDIRECT      - pa2 == gateway address
 * ICT_PARAMP - pa2 == (packet) pointer to parameter error
 * ICT_MASKRP - pa2 == mask address
 * ICT_ECHORQ - pal == seq, pa2 == data size
 */

/*-----
 * icmp - send an ICMP message
 *-----
 */
icmp(type, code, dst, pal, pa2)
short    type, code;
IPAddr   dst;
char *pal, *pa2;
{
    struct    ep    *pep;
    struct    ip    *pip;
    struct    icmp *pic;
    Bool      isresp, iserr;
    IPAddr     src, tdst;
    int       i, datalen;

    IcmpOutMsgs++;
    blkcopy(tdst, dst, IP_ALEN);      /* worry free pass by value */

    pep = icsetbuf(type, pal, &isresp, &iserr);
    if (pep == SYSERR) {
        IcmpOutErrors++;
        return SYSERR;
    }
    pip = (struct ip *)pep->ep_data;
    pic = (struct icmp *) pip->ip_data;

    datalen = IC_HLEN;
```



```
/* we fill in the source here, so routing won't break it */

if (isresp) {
    if (iserr) {
        if (!icerrok(pep)) {
            freebuf(pep);
            return OK;
        }
        blkcopy(pic->ic_data, pip, IP_HLEN(pip)+8);
        datalen += IP_HLEN(pip)+8;
    }
    icsetsrc(pip);
} else
    blkcopy(pip->ip_src, ip_anyaddr, IP_ALEN);
blkcopy(pip->ip_dst, tdst, IP_ALEN);

pic->ic_type = (char) type;
pic->ic_code = (char) code;
if (!isresp) {
    if (type == ICT_ECHORQ)
        pic->ic_seq = (int) pa1;
    else
        pic->ic_seq = 0;
    pic->ic_id = getpid();
}
datalen += icsetdata(type, pip, pa2);

pic->ic_cksum = 0;
pic->ic_cksum = cksum(pic, (datalen+1)>>1);

pip->ip_proto = IPT_ICMP; /* for generated packets */
ipsend(tdst, pep, datalen, IPT_ICMP, IPP_INCTL, IP_TTL);
return OK;
}
```

Icmp takes the ICMP message type and code as arguments, along with a destination IP address and two final arguments that usually contain pointers. The exact meaning and type of the two final arguments depends on the ICMP message type. For example, for an ICMP echo request, the argument pa1 contains an (integer) sequence number, while argument pa2 contains the (integer) data size. For an ICMP echo response, argument pa1



contains a pointer to a packet containing the ICMP echo request that caused the reply, while argument pa2 is not used (it contains zero).

To build an ICMP message, procedure icmp calls icsetbuf to allocate a buffer. To insure compliance with the protocol, it fills in the datagram source address before sending the message to IP. For responses, icmp uses the destination address to which the request was sent; otherwise, it fills the source field with ip_anyaddr and allows the IP routing procedures to choose an outgoing address. For responses, icmp also calls icerrok to verify that it is not generating an error message about an error message.

Icmp then fills in remaining header fields, including the type and code fields. For an echo request, it sees the identification field to the process id of the sending process. Finally, it calls icsetdata to fill in the data area, computes the ICMP checksum, and calls ipsend to send the datagram.

8.9 Avoiding Errors About Errors

Procedure icerrok checks a datagram that caused a problem to verify that the gateway is allowed to send an error message about it. The rules are straightforward: a gateway should never generate an error message about an error message, or for any fragment other than the first, or for broadcast datagrams. The code checks each condition and returns FALSE if an error message is prohibited and TRUE if it is allowed.

```
/* icerrok.c - icerrok */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * icerrok - is it ok to send an error response?
 */
Bool icerrok(pep)
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct icmp *pic = (struct icmp *)pip->ip_data;

    /* don't send errors about error packets... */

    if (pip->ip_proto == IPT_ICMP)
```



```
        switch(pic->ic_type) {
            case ICT_DESTUR:
            case ICT_REDIRECT:
            case ICT_SRCQ:
            case ICT_TIMEX:
            case ICT_PARAMP:
                return FALSE;
            default:
                break;
        }
        /* ...or other than the first of a fragment */

        if (pip->ip_fragoff & IP_FRAGOFF)
            return FALSE;
        /* ...or broadcast packets */

        if (isbrc(pip->ip_dst) || IP_CLASSD(pip->ip_dst))
            return FALSE;
        return TRUE;
    }
}
```

8.10 Allocating A Buffer For ICMP

Procedure icsetbuf allocates a buffer for an ICMP error message, and sets two Boolean variables, one that tells whether the message is an error message (or an information request), and another that tells whether this message type is a response to a previous request.

```
/* icsetbuf.c - icsetbuf */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * icsetbuf - set up a buffer for an ICMP message
 */
struct ep *icsetbuf(type, pal, pisresp, piserr)
int type;
char *pal;           /* old packet, if any */
Bool *pisresp,      /* packet is a response */
```



```
*piserr;      /* packet is an error */  
{  
    struct ep *pep;  
  
    *pisresp = *piserr = FALSE;  
  
    switch (type) {  
    case ICT_REDIRECT:  
        pep = (struct ep *)getbuf(Net.netpool);  
        if (pep == SYSERR)  
            return SYSERR;  
        blkcopy(pep, pal, MAXNETBUF);  
        pal = (char *)pep;  
        *piserr = TRUE;  
        break;  
    case ICT_DESTUR:  
    case ICT_SRCQ:  
    case ICT_TIMEX:  
    case ICT_PARAMP:  
        pep = (struct ep *)pal;  
        *piserr = TRUE;  
        break;  
    case ICT_ECHORP:  
    case ICT_INFOP:  
    case ICT_MASKRP:  
        pep = (struct ep *)pal;  
        *pisresp = TRUE;  
        break;  
    case ICT_ECHORQ:  
    case ICT_TIMERQ:  
    case ICT_INFORQ:  
    case ICT_MASKRQ:  
        pep = (struct ep *)getbuf(Net.lrgpool);  
        if (pep == SYSERR)  
            return SYSERR;  
        break;  
    case ICT_TIMERP:      /* Not Implemented */  
        /* IcmpOutTimestampsReps++; */  
        IcmpOutErrors--; /* Kludge: we increment above */  
        freebuf(pal);  
        return SYSERR;  
    }
```



```
    }
    if (*piserr)
        *pisresp = TRUE;
    switch (type) {           /* Update MIB Statistics */
    case ICT_ECHORP:   IcmpOutEchos++;      break;
    case ICT_ECHORQ:   IcmpOutEchoReps++;   break;
    case ICT_DESTUR:  IcmpOutDestUnreachs++; break;
    case ICT_SRCQ:    IcmpOutSrcQuenches++; break;
    case ICT_REDIRECT: IcmpOutRedirects++;   break;
    case ICT_TIMEX:   IcmpOutTimeExcds++;   break;
    case ICT_PARAMP:  IcmpOutParmProbs++;   break;
    case ICT_TIMERQ:  IcmpOutTimestamps++;  break;
    case ICT_TIMERP:  IcmpOutTimestampReps++; break;
    case ICT_MASKRQ:  IcmpOutAddrMasks++;   break;
    case ICT_MASKRP:  IcmpOutAddrMaskReps++; break;
    }
    return pep;
}
```

The code is straightforward and divides into four basic cases. For most replies, icsetbuf reuses the buffer in which the request arrived (i.e., returns the address supplied in argument pa1). For unimplemented message types, icsetbuf deallocates the datagram that caused the problem and returns SYSERR. For ICMP messages that could contain large amounts of data (e.g., an echo reply), icsetbuf allocates a large buffer. For other messages that cannot use the original buffer, icsetbuf allocates a standard buffer.

8.11 The Data Portion Of An ICMP Message

Procedure icsetdata creates the data portion of an ICMP message. The action taken depends on the message type, which icsetdata receives as an argument.

```
/* icsetdata.c - icsetdata */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/* ECHOMAX must be an even number */
#define ECHOMAX(pip) (MAXLRGBUF-IC_HLEN-IP_HLEN(pip)-EP_HLEN-EP_CRC)

/*-----
```



```
* icsetdata - set the data section. Return value is data length
*-----
*/
int icsetdata(type, pip, pa2)
int      type;
struct   ip    *pip;
char     *pa2;
{
    struct   icmp *pic = (struct icmp *)pip->ip_data;
    int      i, len;

    switch (type) {
    case ICT_ECHORP:
        len = pip->ip_len - IP_HLEN(pip) - IC_HLEN;
        if (isodd(len))
            pic->ic_data[len] = 0; /* so cksum works */
        return len;
    case ICT_DESTUR:
    case ICT_SRCQ:
    case ICT_TIMEX:
        pic->ic_mbz = 0;           /* must be 0 */
        break;
    case ICT_REDIRECT:
        blkcopy(pic->ic_gw, pa2, IP_ALEN);
        break;
    case ICT_PARAMP:
        pic->ic_ptr = (char) pa2;
        for (i=0; i<IC_PADLEN; ++i)
            pic->ic_pad[i] = 0;
        break;
    case ICT_MASKRP:
        blkcopy(pic->ic_data, pa2, IP_ALEN);
        break;
    case ICT_ECHORQ:
        if (pa2 > ECHOMAX(pip))
            pa2 = ECHOMAX(pip);
        for (i=0; i<(int)pa2; ++i)
            pic->ic_data[i] = i;
        if (isodd(pa2))
            pic->ic_data[(int)pa2] = 0;
        return (int)pa2;
    }
```



```
    case ICT_MASKRQ:
        blkcopy(pic->ic_data, ip_anyaddr, IP_ALEN);
        return IP_ALEN;
    }
    return 0;
}
```

For replies, icmp has created the outgoing message from the incoming request, so there is no need to copy data. However, icsetdata must compute and return the correct data length. For most messages, the data length is zero because the header contains all necessary information. Icsetdata fills in the appropriate fields. For example, in an ICMP redirect message, the caller supplies a pointer to the new gateway address in argument pa1, and icsetdata copies it into the message.

For ICMP echo reply messages, icsetdata computes the length from the incoming request message. To do so, it subtracts the IP header length and the ICMP header length from the datagram length. In addition, for odd-length echo reply messages, icsetdata must place an additional zero octet after the message, so the 16-bit checksum algorithm works correctly. For ICMP echo request messages, argument pa2 specifies the data length.

8.12 Generating An ICMP Redirect Message

With the above ICMP procedures in place, it becomes easy to generate an ICMP error message. For example, procedure ipredirect generates an ICMP redirect message.

```
/* ipredirect.c - ipredirect */

#include <conf.h>
#include <kernel.h>
#include <network.h>

struct route *rtget();

/*
 * ipredirect - send redirects, if needed
 */
void ipredirect(pep, ifnum, prt)
struct ep *pep;          /* the current IP packet */
int ifnum;               /* the input interface */
struct route *prt;       /* where we want to route it */

```



```
{  
    struct ip *pip = (struct ip *) pep->ep_data;  
    struct route *tprt;  
    int rdtype, isonehop;  
    IPAddr nmask; /* network part's mask */  
  
    if (ifnum == NI_LOCAL || ifnum != prt->rt_ifnum)  
        return;  
    tprt = rtget(pip->ip_src, RTF_LOCAL);  
    if (!tprt)  
        return;  
    isonehop = tprt->rt_metric == 0;  
    rtfree(tprt);  
    if (!isonehop)  
        return;  
    /* got one... */  
  
    netmask(nmask, prt->rt_net); /* get the default net mask */  
    if (blkequ(prt->rt_mask, nmask, IP_ALEN))  
        rdtype = ICC_NETRD;  
    else  
        rdtype = ICC_HOSTRD;  
    icmp(ICK_REDIRECT, rdtype, pip->ip_src, pep, prt->rt_gw);  
}
```

The three arguments to ipredirect specify a pointer to a buffer that contains a packet, an interface number over which the packet arrived, and a pointer to a new route. After checking to insure that the interface does not refer to the local host and that the new route specifies an interface other than the one over which the packet arrived, ipredirect calls rtget to compute the route to the machine that sent the datagram.

Because the protocol specifies that a gateway can only send an ICMP redirect to a host on a directly connected network, ipredirect checks the metric on the route it found to the destination. A metric greater than zero means the host is not directly connected and causes ipredirect to return without sending a message. Once ipredirect finds that the offending host is on a directly connected network, it must examine the new route to determine whether it is a host-specific route or network-specific route. To do so, it examines the subnet mask associated with the route. If the mask covers more than the network portion, ipredirect declares the message to be a host redirect; otherwise, it declares the message a network redirect.



8.13 Summary

Conceptually, ICMP can be divided into two parts: one that handles incoming ICMP messages and another that generates outgoing ICMP messages. While both hosts and gateways must handle incoming messages, most outgoing messages are restricted to gateways. Thus, ICMP code is usually more complex in gateways than in hosts.

In practice, many details and the interaction between incoming and outgoing messages make ICMP code complex. Our design uses two primary procedures: `icmp_in` to handle incoming messages, and `icmp` to generate outgoing messages. Each of these calls several subprocedures to handle the details of creation of buffers, setting subnet masks, filling the header and data fields, and computing correct source addresses.

8.14 FOR FURTHER STUDY

Postel [RFC 792] describes the ICMP protocol. Mogul and Postel [RFC 950] adds subnet mask request and reply messages, while Braden et. al. specifies many refinements [RFC 1122]. The gateway requirements document [RFC 1009] discusses how gateways should generate and handle ICMP messages.

8.15 EXERCISES

1. Consider procedure `icsetsrc`. Under what circumstances can the loop iterate through all interfaces without finding a match?
2. When it forms a reply, can ICMP merely reverse the source and destination address fields from the request? Explain. (Hint: read the protocol specification)
3. What should a host do when it receives an ICMP time exceeded message?
4. What should a host do when it receives an ICMP source quench message?
5. Suppose a gateway generates an ICMP redirect message for a destination that it knows has a subnet address (i.e., the subnet mask extends past the network portion of the address). Should it specify the redirect as a host redirect or as a network redirect? Explain. (Hint: see RFC 1009.)
6. What does the example code do in response to an ICMP source quench message? What other messages are handled the same way?
7. Look carefully at `setmask`. It handles two types of broadcast address (all 0's and all 1's). Find pertinent statement(s) in the protocol standard that specify whether using two types of broadcast address is required, allowed, or forbidden.



9 IP: Multicast Processing (IGMP)

9.1 Introduction

Hosts and gateways use the Internet Group Management Protocol (IGMP) to manage groups of computers that participate in multicast datagram delivery. This chapter examines the details of multicast routing and IGMP processing. It shows how a host manages information about multicast groups, recognizes incoming multicast datagrams, and sends outgoing datagrams. The chapter also discusses how a host joins or leaves a multicast group, responds to a query from a gateway, and maps an IP multicast address to a corresponding physical address.

9.2 Maintaining Multicast Group Membership Information

IP uses class D addresses for multicast delivery. Conceptually, a class D address defines a set of hosts that all receive a copy of any datagram sent to the address. The standard uses the term host group to define the set of all hosts associated with a given multicast address. Host group membership is dynamic — a given host can choose when to join or leave a group. At any time, a given host can be a member of zero or more host groups, and a host group can contain zero or more members.

Each host that participates in IP multicast maintains its own record of host group membership — no single host or gateway knows the members of a given host group. Because all records are kept locally, a host can choose to join or leave a host group without obtaining permission from other hosts or gateways.^① Keeping membership information locally means multicast datagram delivery must operate with the same best-effort semantics used for conventional IP datagrams. Because it does not know the membership, a computer that sends a multicast datagram cannot determine if all members of the host group receive a copy.

^① A host informs multicast gateways when it joins a group, but it does not need permission nor does it receive an acknowledgement.



9.3 A Host Group Table

Most implementations of IP multicast software use a table to store information about the host groups to which the machine currently belongs. If a multi-homed host participates in IP multicast, it must choose between two alternatives. The host can support multicasting on multiple interfaces, in which case it must keep separate host group information for each interface. Alternatively, a multi-homed host can designate one interface to be used for multicasting. If it chooses the latter alternative, the multi-homed host must disallow multicasts except on the designated interface.

Our implementation keeps the host group information for all interfaces in a single, global table. Although the code restricts multicasting to a single interface, each entry in the table includes a field that specifies the interface to which the entry corresponds. File igmp.h contains the declarations.

```
/* igmp.h - IG_VER, IG_TYP */

#define HG_TSIZE 15           /* host group table size */

#define IG_VERSION 1          /* RFC 1112 version number */

#define IG_HLEN 8              /* IGMP header length */
#define IGT_HQUERY 1           /* host membership query */
#define IGT_HREPORT 2          /* host membership report */

struct igmp {
    unsigned char ig_vertyp;   /* version and type field */
    char ig_unused;           /* not used by IGMP */
    unsigned short ig_cksum;  /* compl. of 1's compl. sum */
    IPAddr ig_gaddr;          /* host group IP address */
};

#define IG_VER(pig) ((pig)->ig_vertyp>>4) & 0xf
#define IG_TYP(pig) ((pig)->ig_vertyp & 0xf)

#define IG_NSEND 2             /* # IGMP join messages to send */
#define IG_DELAY 5              /* delay for resends (1/10 secs) */

/* Host Group Membership States */

#define HGS_FREE 0             /* unallocated host group table entry*/
#define HGS_DELAYING 1          /* delay timer running for this group*/
#define HGS_IDLE 2              /* in the group but no report pending*/
```



```
#define HGS_STATIC 3 /* for 224.0.0.1; no state changes */

struct hg {
    unsigned char hg_state; /* HGS_* above */
    unsigned char hg_ifnum; /* interface index for group */
    IPAddr hg_ipa; /* IP multicast address */
    unsigned long hg_refs; /* reference count */
    Bool hg_ttl; /* max IP ttl for this group */
};

/* Host Group Update Process Info. */

extern int igmp_update();

#define IGUSTK 4096 /* stack size for update proc. */
#define IGUPRI 50 /* update process priority */
#define IGUNAM "igmp_update" /* name of update process */
#define IGUARGC 0 /* count of args to hgupdate */

struct hginfo {
    Bool hi_valid; /* TRUE if hginit() has been called */
    int hi_mutex; /* table mutual exclusion */
    int hi_uport; /* listen port for delay timer expires */
};

extern struct hginfo HostGroup;

extern IPAddr ig_allhosts; /* "all hosts" group address (224.0.0.1)*/
extern IPAddr ig_allDmask; /* net mask to match all class D addrs. */
extern struct hg hhtable[];
```

Array hhtable implements the host group table. Each entry in hhtable corresponds to one host group, and contains four fields defined by structure hg. Field hg_state records the current state of an entry. When hg_state contains the value HGS_FREE, the entry is not currently used and all other fields are invalid. Field hg_ifnum specifies the interface to which an entry corresponds. Field hg_ipa contains the IP multicast address for the host group, and field hg_refs contains a reference count that specifies how many processes are currently using an entry.

File igmp.h also defines symbolic constants, message format, and other data structures used by multicasting code. For example, to insure that only one process searched or modifies entries in hhtable at any time, the code uses a mutual exclusion



semaphore. Field `hi_mutex` of structure `hginfo` contains the semaphore identifier. Each procedure that uses `hhtable` waits on the semaphore before using the table, and signals the semaphore when it finishes.

9.4 Searching For A Host Group

When a host changes membership in a host group or when a multicast datagram arrives, IGMP software calls procedure `hglookup` to find the multicast address in the host group table.

```
/* hglookup.c - hglookup */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/*
 * hglookup - get host group entry (if any) for a group
 * N.B. - Assumes HostGroup.hi_mutex *held*
 */
struct hg *hglookup(ifnum, ipa)
int ifnum;          /* interface for the host group */
IPAddr ipa;        /* IP multicast address */
{
    struct hg *phg;
    int i;

    phg = &hhtable[0];
    for (i=0; i < HG_TSIZE; ++i, ++phg) {
        if (phg->hg_state == HGS_FREE)
            continue;
        if (ifnum == phg->hg_ifnum && ipa == phg->hg_ipa)
            return phg;
    }
    return 0;
}
```

Hglookup searches `hhtable` until it finds an entry that matches the multicast address



specified by argument ipa and the interface number specified by argument ifnum. It returns the address of the entry if one is found, and zero otherwise.

Our implementation of hglookup uses a sequential search because it assumes the host group table will contain only a few entries. However, the code has been isolated in a procedure to make it easy to substitute an alternative scheme that handles large host group tables efficiently.

9.5 Adding A Host Group Entry To The Table

When an application first joins a host group, a new entry must be inserted in hhtable. In addition, the network hardware must be configured to recognize the hardware multicast address that the host group uses. Procedure hgadd performs both operations.

```
/* hgadd.c - hgadd */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/*
 * hgadd - add a host group entry for a group
 */
int hgadd(ifnum, ipa, islocal)
int ifnum;          /* interface for the host group */
IPAddr ipa;         /* IP multicast address */
Bool islocal; /* true if this group is local */
{
    struct hg *phg;
    static int     start;
    int          i;

    wait(HostGroup.hi_mutex);
    for (i=0; i < HG_TSIZE; ++i) {
        if (++start >= HG_TSIZE)
            start = 0;
        if (hhtable[start].hg_state == HGS_FREE)
            break;
    }
    phg = &hhtable[start];
```



```
if (phg->hg_state != HGS_FREE) {
    signal(HostGroup.hi_mutex);
    return SYSERR;           /* table full */
}
if (hgarpadd(ifnum, ipa) == SYSERR) {
    signal(HostGroup.hi_mutex);
    return SYSERR;
}
phg->hg_ifnum = ifnum;
phg->hg_refs = 1;
if (islocal)
    phg->hg_ttl = 1;
else
    phg->hg_ttl = IP_TTL;
blkcopyy(phg->hg_ipa, ipa, IP_ALEN);
if (blkque(ipa, ig_allhosts, IP_ALEN))
    phg->hg_state = HGS_STATIC;
else
    phg->hg_state = HGS_IDLE;
signal(HostGroup.hi_mutex);
return OK;
}
```

When procedure hgadd begins, it waits on semaphore HostGroup.hi_mutex to guarantee exclusive access to the host group table. Hgadd then searches all locations of hptable, beginning with the location given by static variable start. The search terminates when hgadd finds an unused table entry or finishes examining all locations. If an unused entry exists, field hg_state of the entry will contain HGS_FREE. If no free entry exists, hgadd signals the mutual exclusion semaphore, and returns SYSERK to its caller. If an unused entry exists in the host group table, hgadd configures the network interface and hardware, initializes fields of the entry in hptable, signals the mutual exclusion semaphore, and returns OK to its caller.

9.6 Configuring The Network Interface For A Multicast Address

When a host first joins a host group, the software and hardware must be configured to handle both transmission and reception of datagrams for the group. To accommodate multicast transmission, changes must be made at two levels: a route must be installed in the IP routing table and the network interface software must be configured to bind the IP



multicast address to an appropriate hardware address. To accommodate reception, the system must be configured to recognize datagrams sent to the group.

If the underlying hardware supports multicast, IP uses the hardware multicast facility to send and receive IP multicast datagrams. Each host group is assigned a unique hardware multicast address that is used for all transmissions to the group. If the hardware does not support multicast, IP uses hardware broadcast to deliver all multicast datagrams. To distinguish the two cases, our example network interface structure includes field `ni_mcast`. If the network supports hardware multicast, `ni_mcast` contains the address of a device driver procedure that initializes the hardware to accept the correct multicast address. If the network does not support hardware multicast, `ni_mcast` contains zero.

Procedure `hgarpadd` configures the hardware to accept packets sent to the hardware multicast address, and configures the address binding mechanism to map the IP multicast address in outgoing datagrams to the appropriate hardware address. File `hgarpadd.c` contains the code.

```
/* hgarpadd.c - hgarpadd */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/*
 * hgarpadd - add an ARP table entry for a multicast address
 */
int hgarpaddifnum, ipa)
int ifnum;
IPAddr ipa;
{
    struct netif *pni = &nif[ifnum];
    struct arpentry *pae, *arpalloc();
    int ifdev = nif[ifnum].ni_dev;
    STATWORD ps;

    disable(ps);
    pae = arpalloc();
    if (pae == 0) {
        restore(ps);
        return SYSERR;
    }
```



```
pae->ae_hwtype = pni->ni_hwtype;
pae->ae_prtype = EPT_IP;
pae->ae_pni = pni;
pae->ae_hwlen = pni->ni_hwa.ha_len;
pae->ae_prlen = IP_ALEN;
pae->ae_queue = EMPTY;
blkcopy(pae->ae_pra, ipa, IP_ALEN);
if (pni->ni_mcast)
    (pni->ni_mcast)(NI_MADD, ifdev, pae->ae_hwa, ipa);
else
    blkcopy(pae->ae_hwa, pni->ni_hwb.ha_addr, pae->ae_hwlen);
pae->ae_ttl = ARP_INF;
pae->ae_state = AS_RESOLVED;
restore(ps);
return OK;
}
```

The code uses an ARP cache entry to hold the binding between an IP multicast address and corresponding hardware address. Hgarpadd calls arpalloc to allocate an entry in the ARP cache, and fills in the fields of the entry. It copies the IP multicast address from argument ipa to field ae_pra, and consults the network interface to obtain values for hardware address type and length. To insure that ARP software does not time out and remove the entry, hgarpadd assigns field ae_ttl the value ARP_INF, which specifies an infinite lifetime.

Hgarpadd tests ni_mcast in the network interface to determine how to compute a hardware address. If ni_mcast contains zero, the network does not support hardware multicast; hgarpadd copies the hardware broadcast address into the ARP entry. If ni_mcast is nonzero, it gives the address of a function that translates an IP multicast address into the corresponding hardware multicast address; hgarpadd calls the function to compute a hardware address. In either case, hgarpadd fills in the hardware address field of the ARP entry so the ARP code will find a valid hardware address for outgoing multicast datagrams.

9.7 Translation Between IP and Hardware Multicast Addresses

The details of hardware multicast addressing depend on network technologies. In the case of Ethernet, the IGMP standard specifies that the hardware multicast address a host group uses is computed by adding the low-order 23 bits of the class D address to 0x01005E000000. Procedure ethmcast performs the operations needed to translate an IP



multicast address into an Ethernet multicast address.

```
/* ethmcast.c - ethmcast */

#include <conf.h>
#include <kernel.h>
#include <network.h>

Eaddr      template = { 0x01, 0x00, 0x5E, 0x00, 0x00, 0x00 };

/*-----
 * ethmcast - generate & set an IP multicast hardware address
 *-----
 */
int ethmcast(op, dev, hwa, ipa)
int      op;
int      dev;
Eaddr      hwa;
IPAddr     ipa;
{
    blkcopy(hwa, template, EP_ALEN);
    /* add in low-order 23 bits of IP multicast address */
    hwa[3] = ipa[1] & 0x7;
    hwa[4] = ipa[2];
    hwa[5] = ipa[3];

    switch (op) {
    case NI_MADD:
        return control(dev, EPC_MADD, hwa);
        break;
    case NI_MDEL:
        return control(dev, EPC_MDEL, hwa);
        break;
    }
    return OK;
}
```

Ethmcast takes four arguments that specify an operation (i.e., whether to add or delete the address), a hardware device number, a location to store the hardware multicast address, and the location of an IP multicast address. Because the low-order bits of the base Ethernet address used for multicasting contain zeroes, addition becomes unnecessary. Instead, ethmcast copies the base address from variable template into the



location given by argument hwa, and then moves in the 23 low-order bits of the IP multicast address from argument ipa.

After ethmcast forms a hardware multicast address, it calls the Xinu function control to request that the device driver inform the Ethernet hardware. Once the hardware has been informed about a new address, it will begin accepting Ethernet packets destined for that address.

9.8 Removing A Multicast Address From The Host Group Table

When a host leaves a host group, it calls procedure hgarpdel to remove the ARP cache entry and inform the hardware.

```
/* hgarpdel.c - hgarpdel */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

struct arpentry *arpfind(u_char *, u_short, struct netif *);

/*
 * hgarpdel - remove an ARP table entry for a multicast address
 */
int hgarpdel(ifnum, ipa)
int ifnum;
IPAddr ipa;
{
    struct netif *pni = &nif[ifnum];
    struct arpentry *pae, *arpfind();
    int ifdev = nif[ifnum].ni_dev;
    STATWORD ps;

    disable(ps);
    if (pae = arpfind(ipa, EPT_IP, pni))
        pae->ae_state = AS_FREE;
    if (pni->ni_mcast)
        (pni->ni_mcast)(NI_MDEL, ifdev, pae->ae_hwa, ipa);
    restore(ps);
```



```
        return OK;  
    }
```

Hgarpdel operates as expected. It calls arpfnd to locate the ARP cache entry for the specified IP address, and changes the state of the entry to AS_FREE. Hgarpdel also examines field ni_mcast in the network interface structure to determine whether the network supports multicast. If the hardware supports multicast, hgarpdel calls the device driver function to inform the hardware that it should no longer accept incoming packets sent to the group's hardware multicast address.

9.9 Joining A Host Group

An application calls function hgjoin to request that its host join a host group. Hgjoin configures the host to send and receive multicast datagrams addressed to the host group, and then notifies other machines on the network that the host has joined the group. File hgjoin.c contains the code.

```
/* hgjoin.c - hgjoin */  
  
#include <conf.h>  
#include <kernel.h>  
#include <network.h>  
#include <sleep.h>  
#include <igmp.h>  
  
/*-----  
 * hgjoin - handle application request to join a host group  
 *-----  
 */  
int  
hgjoin(ifnum, ipa, islocal)  
int ifnum; /* interface for the host group */  
IPAddr ipa; /* IP multicast address */  
Bool islocal; /* true if this group is local */  
{  
    struct hg *phg;  
    int i;  
  
    if (!IP_CLASSD(ipa))  
        return SYSERR;  
    /* restrict multicast in multi-homed host to primary interface */
```



```
if (ifnum != NI_PRIMARY)
    return SYSERR;

wait(HostGroup.hi_mutex);

if (phg = hglookup(ifnum, ipa)) {
    phg->hg_refs++;
    signal(HostGroup.hi_mutex);
    return OK; /* already in it */
}

signal(HostGroup.hi_mutex);

/* add to host group and routing tables */
if (hgadd(ifnum, ipa, islocal) == SYSERR)
    return SYSERR;

rtadd(ipa, ip_maskall, ipa, 0, NI_LOCAL, RT_INF);
/*
 * advertise membership to multicast router(s); don't advertise
 * 224.0.0.1 (all multicast hosts) membership.
 */
if (ipa != ig_allhosts)
    for (i=0; i < IG_NSEND; ++i) {
        igmp(IGT_HREPORT, ifnum, ipa);
        sleep10(IG_DELAY);
    }
return OK;
}
```

Hgjoin first checks argument ipa to verify that it contains a class D address, and returns SYSERR if it does not. It then verifies that the host group table does not already contain the specified address. To do so, it obtains exclusive use of the host group table, and calls hglookup to search for the specified IP address. If hglookup finds the address in the table, hgjoin increments the reference count on the entry, releases exclusive use of the table, and returns to its caller.

If address ipa is valid and not present in the host group table, hgjoin configures the host to participate in the host group. To do so, hgjoin first calls hgadd to add the new address to the host group table, insert a permanent entry in the ARP cache for the address, and inform the hardware that it should accept packets sent to the corresponding hardware multicast address. If hgadd returns successfully, hgjoin calls rtadd to add a permanent route to the IP routing table. The route handles incoming multicast the same way the IP routing table handles broadcast — any incoming datagram destined for multicast address ipa will be forwarded to the local interface.



9.10 Maintaining Contact With A Multicast Router

To eliminate unnecessary multicast traffic on a network, hosts and multicast routers use the internet Group Management Protocol (IGMP). In essence, a multicast router periodically sends an IGMP query that requests all hosts participating in IP multicast to report the set of groups in which they have membership. A host sends an IGMP report message for each group in which the host has membership. If several cycles pass during which no host reports membership in a given group, the multicast routers stop transmitting datagrams destined for the group.

The standard specifies that when a host first joins a host group, it should send an announcement to the group. The final lines of hgjoin implement the announcement. Hgjoin calls procedure igmp to send the announcement to the host group. To help insure that the message does not become lost, the code sends the message IG_NSEND times, with a delay of IG_DELAY tenths of seconds after each transmission.

Procedure igmp forms and sends one ICMP message,

```
/* igmp.c - igmp */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/*
 * igmp - send IGMP requests/responses
 */
int
igmp(int typ, unsigned ifnum, IPAddr hga)
{
    int typ;      /* IGT_* from igmp.h */
    int ifnum;     /* intreface # this group (currently unused) */
    IPAddr hga;    /* host group multicast addr. */
{
    struct ep *pep;
    struct ip *pip;
    struct igmp *pig;
    int i, len;

    pep = (struct ep *)getbuf(Net.netpool);
    if (pep == (struct ep *)SYSERR)
        return SYSERR;
    pip = (struct ip *)pep->ep_data;
```



```
    pig = (struct igmp *) pip->ip_data;
    pig->ig_vertyp = (IG_VERSION<<4) | (typ & 0xf);
    pig->ig_unused = 0;
    pig->ig_cksum = 0;
    blkcopy(pig->ig_gaddr, hga, IP_ALEN);
    pig->ig_cksum = cksum((WORD *)pig, IG_HLEN>>1);

    ipsend(hga, pep, IG_HLEN, IPT_IGMP, IPP_INCTL, 1);
    return OK;
}

/* special IGMP-relevant address & mask */

IPAddr ig_allhosts = { 224, 0, 0, 1 };
IPAddr ig_allDmask = { 240, 0, 0, 0 };
```

Procedure `igmp` allocates a network buffer to hold one packet, and fills in the IGMP message. Structure `imgp` defines the format of an IGMP message^①. Field `ig_vertyp` contains a protocol version number and message type: the caller specifies the IGMP message type in argument `typ`. In a report message, field `ig_gaddr` contains the host group address, which the caller passes to `igmp` in argument `hga`.

Multicast routers send IGMP query messages to address 224.0.1, the all hosts group. Because all multicast routers receive all multicast packets, a host does not need to know the routers' addresses, nor does it need to send a response directly to each router. Instead, a host sends a response for a given group using the group's multicast address. Thus, each host that participates in a given host group receives all membership reports.

To avoid an explosion of responses after an IGMP query, the protocol specifies that a host must delay each report for a random time between one and ten seconds. Furthermore, as soon as a host sends a report for a particular host group all other hosts cancel their timers for that host group until another query arrives.

9.11 Implementing IGMP Membership Reports

When an IGMP query arrives, a host must set a random timer for each host group. Procedure `igmp_settimers` uses the general-purpose timer mechanism described in Chapter 14 to perform the task.

```
/* igmp_settimers.c - igmp_settimers */
```

^① The declaration of structure `igmp` can be found in file `igmp.h` on page 148.



```
#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/*
 * igmp_settimers - generate timer events to send IGMP reports
 */
int
igmp_settimers(ifnum)
int ifnum;
{
    int i;

    wait(HostGroup.hi_mutex);
    for (i=0; i<HG_TSIZE; ++i) {
        struct hg *phg = &hhtable[i];
        if (phg->hg_state != HGS_IDLE || phg->hg_ifnum != ifnum)
            continue;
        phg->hg_state = HGS_DELAYING;
        tmset(HostGroup.hi_uport, HG_TSIZE, phg, hgrand());
    }
    signal(HostGroup.hi_mutex);
    return OK;
}
```

igmp_settimers iterates through the host group table and examines each entry. If field hg_state contains HGS_IDLE, the entry represents an active host group for which no timer event has been scheduled. For each such entry, igmp_settimers changes the state to HGS_DELAYING and calls tmset to create a timer event for the entry.

The first argument to tmset specifies a Xinu port to which a message will be sent when the timer expires, and the second argument specifies the maximum size of the port. The third argument contains a message to be sent, while the fourth specifies a delay to hundredths of seconds. Igmp_settimers passes a pointer to the host group entry as the message to be sent.

9.12 Computing A Random Delay

Because the standard specifies that the report for each host group should be delayed



a random time, igmp_settimers calls function hgrand to compute a delay. File hgrand.c contains the code.

```
/* hgrand.c - hgrand */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

int modulus = 1009;           /* ~10 secs in 1/100'th secs */
int offset = 523;            /* additive constant */
int hgseed;                  /* initialized in hginit() */

/*
 * hgrand - return "random" delay between 0 & 10 secs (in 1/100 secs)
 */
int hgrand()
{
    int rv;

    rv = ((modulus+1) * hgseed + offset) % modulus;
    if (rv < 0)
        rv += modulus;      /* return only positive values */
    hgseed = rv;
    return rv;
}
```

Because the underlying timer mechanism requires delays to be specified using an integer that represents hundredths of seconds, hgrand uses integer computation. Like most pseudo-random number generators, hgrand returns an integer value between zero and a modulus on each call such that the sequence of values produced appears to be random. The modulus value chosen, 1009, is a prime number approximately equal to ten seconds of delay.

If all hosts on a network use the same pseudo-random number generator algorithm, they can generate the same sequence of delays. Because identical delays can cause collisions and generate unnecessary traffic, the IGMP standard specifies that hosts using a pseudo-random number generator must use an initial seed value that guarantees a unique sequence of delays. As a result, two hosts will not use exactly the same delays, even if they have identical hardware and run identical code. The example code



guarantees a unique sequence by initializing the seed, kept in global variable hgseed, to the host's IP address

9.13 A Process To Send IGMP Reports

When a timer expires, the timing mechanism sends a message to an operating system port. A process must be waiting at the port to receive the message. Process igmp_update receives IGMP timer events.

```
/* igmp_update.c - igmp_update */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/*
 * igmp_update - send (delayed) IGMP host group updates
 */
PROCESS igmp_update()
{
    struct hg *phg;

    HostGroup.hi_uport = pcreate(HG_TSIZE);
    while (1) {
        phg = (struct hg *)preceive(HostGroup.hi_uport);
        wait(HostGroup.hi_mutex);
        if (phg->hg_state == HGS_DELAYING) {
            phg->hg_state = HGS_IDLE;
            igmp(IGT_HREPORT, phg->hg_ifnum, phg->hg_ipa);
        }
        signal(HostGroup.hi_mutex);
    }
}
```

After creating a port, igmp_update enters an infinite loop. During each iteration, it calls preceive to block on the port until a message arrives. Once a message arrives, igmp_update waits on the mutual exclusion semaphore to obtain, exclusive access to the table, sends the report, and then releases exclusive use.

The call to preceive returns a pointer to a single entry in the host group table for



which the timer has expired; igmp update should send an IGMP report for that entry. Because igmp_update runs as a process, scheduling and context switching can delay its execution. In particular, datagrams can arrive and other processes can run during the delay. Thus, the state of an entry can change between the instant the timer expires and the instant igmp_update executes. To insure that exactly one report is sent, igmp_update examines field hg_state. If the entry has state HGS_DELAYING, igmp_update calls igmp to send a report, and then changes the state to HGS_IDLE. If the state has already changed, igmp_update does not send a report.

9.14 Handling Incoming IGMP Messages

When an IGMP message arrives, IP calls procedure igmp_in to handle it.

```
/* igmp_in.c - igmp_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/*
 * igmp_in - handle IGMP packet coming in from the network
 */
int igmp_in(pni, pep)
struct netif *pni; /* not used */
struct ep *pep;
{
    struct ip *pip;
    struct igmp *pig;
    struct hg *phg, *hglookup();
    int ifnum = pni - &nif[0];
    int i, len;

    pip = (struct ip *)pep->ep_data;
    pig = (struct igmp *) pip->ip_data;

    len = pip->ip_len - IP_HLEN(pip);
    if (len != IG_HLEN || IG_VER(pig) != IG_VERSION ||
        cksum((WORD *)pig, len>>1)) {
        freebuf(pep);
        return SYSERR;
    }
}
```



```
    }

    switch(IG_TYP(pig)) {
        case IGT_HQUERY:
            igmp_settimers(NI_PRIMARY);
            break;

        case IGT_HREPORT:
            wait(HostGroup.hi_mutex);
            if ((phg = hglookup(NI_PRIMARY, pig->ig_gaddr)) &&
                phg->hg_state == HGS_DELAYING) {
                tmclear(HostGroup.hi_uport, phg);
                phg->hg_state = HGS_IDLE;
            }
            signal(HostGroup.hi_mutex);
            break;

        default:
            break;
    }
    freebuf(pep);
    return OK;
}
```

Igmp_in first checks the header of the incoming message by computing the actual length and comparing it to the length stored in the header. Igmp_in then examines the version number in the IGMP header to insure that it matches the version number of the software, and verifies the checksum in the header. If any comparison fails, igmp_in discards the message.

Once igmp_in accepts a message, it uses macro IG_TYP to extract the message type. If the message is a query, igmp_in calls igmp_settimers to start a timer for each entry in the host group table.

If the message is a report, it means that another host has sent a reply to a query. Igmp_in calls hglookup to determine if an entry in its host group table corresponds to the host group. If an entry exists, igmp_in calls tmclear to cancel the pending timer event. In any case, after igmp_in handles a message, it calls freebuf to deallocate the buffer.

9.15 Leaving A Host Group

Conceptually, leaving a host group consists of deleting the entry from the host group table, removing the multicast route from the IP routing table, and configuring the network hardware to ignore packets addressed to the group's hardware multicast address. In practice, however, a few details complicate leaving a group. For example, an entry



cannot be removed from the host group table until all processes that are using it have finished.

Procedure hgleave handles the details; an application calls procedure hgleave whenever it decides to leave a particular host group. File hgleave.c contains the code.

```
/* hgleave.c - hgleave */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/*
 * hgleave - handle application request to leave a host group
 */
int hgleave(ifnum, ipa)
int ifnum;
IPAddr ipa;
{
    struct hg *phg, *hglookup();
    int i;

    if (!IP_CLASSD(ipa))
        return SYSERR;
    wait(HostGroup.hi_mutex);
    if (!(phg = hglookup(ifnum, ipa)) || --(phg->hg_refs)) {
        signal(HostGroup.hi_mutex);
        return OK;
    }
    /* else, it exists & last reference */
    rtdel(ipa, ip_maskall);
    hgarpdel(ifnum, ipa);
    if (phg->hg_state == HGS_DELAYING)
        tmclear(HostGroup.hi_uport, phg);
    phg->hg_state = HGS_FREE;
    signal(HostGroup.hi_mutex);
    return OK;
}
```

As expected, procedure hgleave checks its argument to insure that the caller passes



a valid class D address. It also calls hglookup to verify that the specified address currently exists in the host group table. If so, it decrements hgresfs, the reference count of the entry. If the reference count remains positive, hgleave returns to its caller because other processes are currently using the entry.

When the last process using an entry decrements the reference count, the count reaches zero, and the entry can be removed. To remove an entry, hgleave calls rtDEL to delete the route from the routing table, and then calls hgarpdel to remove the ARP cache entry and stop the network hardware from accepting packets for the group.

Before returning to its caller, hgleave checks field hg_state to see whether a timer event exists for the entry. If so, hgleave calls tmclear to remove the event before it marks the entry free.

9.16 Initialization Of IGMP Data Structures

The system calls procedure hginit when it begins. Hginit creates a process to handle multicast updates, initializes the host group table, and joins the all-hosts multicast group.

```
/* hginit.c - hginit */

#include <conf.h>
#include <kernel.h>
#include <sleep.h>
#include <network.h>
#include <igmp.h>

extern    int hgseed;
struct    hginfo   HostGroup;
struct    hg    hgtable[HG_TSIZE];

/*
 * hginit - initialize the host group table
 */
void hginit()
{
    int i;

    HostGroup.hi_mutex = screate(0);
    HostGroup.hi_valid = TRUE;
    resume(create(igmp_update, IGUSTK, IGUPRI, IGUNAM, IGUARGC));
    for (i=0; i<HG_TSIZE; ++i)
```



```
    hgtable[i].hg_state = HGS_FREE;
    hgseed = nif[NI_PRIMARY].ni_ip;
    signal(HostGroup.hi_mutex);
    rtadd(ig_allhosts, ig_allDmask, ig_allhosts, 0, NI_PRIMARY,
          RT_INF);
    hgjoin(NI_PRIMARY, ig_allhosts, TRUE);
}
```

Hginit creates a mutual exclusion semaphore with an initial value of zero before it starts the update process to insure that no other processes can access the host group table until hginit assigns the value HGS_FREE to field hg_state in each entry. Hginit also assigns global variable hgseed the IP address of the host's primary interface. Once the data structures have been initialized, hginit signals the mutual exclusion semaphore to allow access to the host group table.

Hginit calls rtadd to add a route to the IP routing table for the all-hosts multicast group. The route directs any outgoing datagram sent to that address to the primary interface. The call specifies a time to live of RT_INF, making the entry permanent.

As the final step of initialization, hginit calls hgjoin to place the host in the all-hosts group. Once a host has joined the all-hosts group, it will receive IGMP queries.

9.17 Summary

Hosts and gateways use IP multicast to deliver a datagram to a subset of all hosts. The set of hosts that communicate through a given IP multicast address is known as a host group. The IGMP protocol permits a host to join or leave a host group at any time.

To avoid unnecessary traffic, a multicast router periodically sends an IGMP query message to determine the host groups that have members on each network. When a query message arrives, a host sets a random timer for each host group to which it belongs. When the timer expires, the host sends an IGMP report to notify the gateways that at least one host on the local network retains its membership in the host group. All hosts in a given group receive a copy of a report for that group; a host cancels its timer if another host in the group reports first.

9.18 FOR FURTHER STUDY

Deering [RFC 1112] describes the IGMP protocol and specifies the message format. In addition, it specifies implementation requirements and provides the rationale for design decisions.



9.19 EXERCISES

1. What happens if an application generates an outgoing datagram for a host group before joining the group?
2. Assume the probability of datagram loss is π_i , where $0 \leq \pi_i \leq 1$. Derive a formula that specifies the number of cycles a multicast gateway should wait before declaring that all hosts have left a given host group.
3. Consider using conventional (unicast) IP addressing as a mechanism to communicate between two multicast gateways. When one gateway needs to send a multicast datagram to another, it encapsulates the multicast datagram in a unicast datagram and transfers it. What are the advantages of such an approach? Disadvantage?



10 *UDP: User Datagrams*

10.1 Introduction

The User Datagram Protocol (UDP) provides connectionless communication among application programs. It allows a program on one machine to send datagrams to program(s) on other machine(s) and to receive replies. This chapter discusses the implementation of UDP, concentrating on how UDP uses protocol port numbers to identify the endpoints of communication. It discusses two possible approaches to the problem of binding protocol port numbers, and shows the implementation of one approach in detail. Finally, it describes the UDP pseudo-header and examines how procedures that compute the UDP checksum use it.

10.2 UDP Ports And Demultiplexing

Conceptually, communication with UDP is quite simple. The protocol standard specifies an abstraction known as the protocol port number that application programs use to identify the endpoints of communication. When an application program on machine A wants to communicate with an application on machine B, each application must obtain a UDP protocol port number from its local operating system. Both must use these protocol port numbers when they communicate. Using protocol port numbers instead of system-specific identifiers like process, task, or job identifiers keeps the protocols independent of a specific system and allows communication between applications on a heterogeneous set of computer systems.

Although the idea of UDP protocol port numbers seems straightforward, there are two basic approaches to its implementation. Both approaches are consistent with the protocol standard, but they provide slightly different interfaces for application programs. The next sections describe how clients and servers use UDP, and show how the two approaches accommodate each.



10.2.1 Ports Used For Pairwise Communication

As Figure 10.1a illustrates, some applications use UDP for pairwise communication. To do so, each of the two applications obtains a UDP port number from its local operating system, and they both use the pair of port numbers when they exchange UDP messages. In such cases, the ideal interface between the application programs and the protocol software separates the address specification operation from the operations for sending and receiving datagrams. That is, the interface allows an application to specify the local and remote protocol port numbers to be used for communication once, and then sends and receives datagrams many times. Of course, when specifying a protocol port on another machine, an application must also specify the IP address of that machine. Once the protocol port numbers have been specified, the application can send and receive an arbitrary number of datagrams.

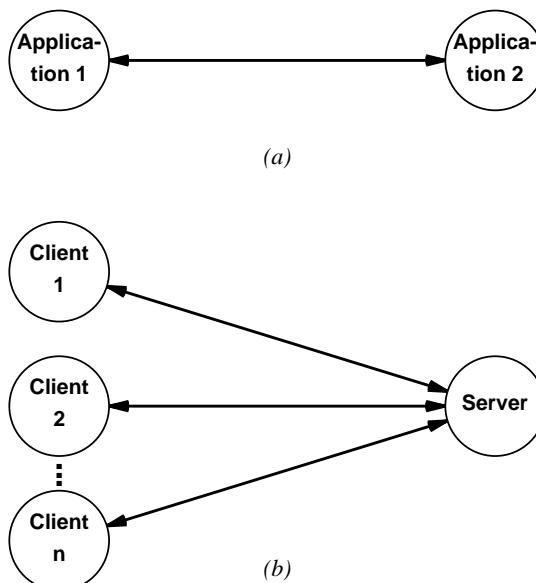


Figure 10.1 The two styles of interaction between programs using UDP. Clients and some other programs use pairwise interaction (a). Servers use many-one interaction (b), in which a single application may send datagrams to many destinations.

10.2.2 Ports Used For Many-One Communication

Most applications use the client-server model of interaction that Figure 10.1b illustrates. A single server application receives UDP messages from many clients. When the server begins, it cannot specify an IP address or a UDP port on another machine because it needs to allow arbitrary machines to send it messages. Instead, it specifies only a local UDP port number. Each message from a client to the server specifies the client's UDP port as well as the server's UDP port. The server extracts the source port



number from the incoming UDP datagram, and uses that number as the destination port number when sending a reply. Of course, the server must also obtain the IP address of the client machine when a UDP datagram arrives, so it can specify the IP address when sending a reply.

Because servers communicate with many clients, they cannot permanently assign a destination IP address or UDP protocol port number. Instead, the interface for many-one communication must allow the server to specify information about the destination each time it sends a datagram. Thus, unlike the ideal interface for pairwise communication, the ideal interface for servers does not separate address specification and datagram transmission.

10.2.3 Modes Of Operation

To accommodate both pairwise communication and many-one communication, most interfaces to UDP use parameters to control the mode of interaction. One mode accommodates the pairwise interaction typical of clients. It allows an application to specify both the local and foreign protocol port numbers once, and then send and receive UDP datagrams without specifying the port numbers each time. Another mode accommodates servers. It allows the server to specify only a local port and then receive from arbitrary clients. The system may require an application program to explicitly declare the mode of interaction, or it may deduce the mode from the port bindings that the application specifies.

10.2.4 The Subtle Issue Of Demultiplexing

In addition to the notion of an interaction mode, a UDP implementation provides an interpretation for protocol port demultiplexing. There are two possibilities:

- Demultiplex using only the destination protocol port number, or
- Demultiplex using source address as well as destination protocol port number.

The choice affects the way application programs interact with the protocol software in a subtle way. To understand the subtlety, consider the two styles of demultiplexing Figure 10.2 illustrates.

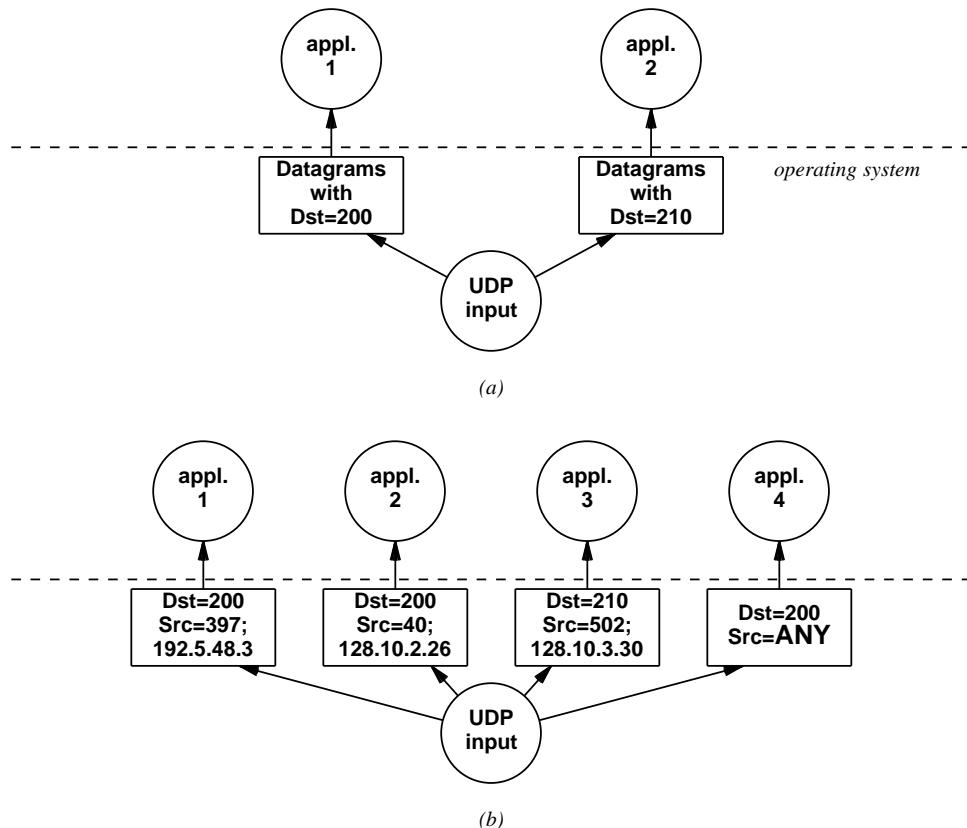


Figure 10.2 The two styles of UDP demultiplexing: (a) using only destination port, and (b) using (source, destination) port pairs. In style (a), an application receives all datagrams to a given destination port. In style (b) it only receives datagrams from the specified source.

In one style of demultiplexing, the system sends all datagrams for a given destination protocol port to the same queue. In the second style of demultiplexing, the system uses the source address (source protocol port number as well as the source IP address) when demultiplexing datagrams. Thus, in the second style, each queue contains datagrams from a given site.

Each style has advantages and disadvantages. For example, in the first style, creating a server is trivial because an application receives all datagrams sent to a given protocol port number, independent of their origin. However, because the system does not distinguish among multiple sources, the system cannot filter erroneously addressed datagrams. Thus, if a datagram arrives addressed to a given port, the application program using that port will receive it, even if it was sent in error. In the second style, creating a client is trivial because a given application receives only those datagrams from the application program with which it has elected to communicate. However, if a single application needs to communicate with two remote applications simultaneously, it must allocate two queues, one for each remote application. Furthermore, the system may



need to provide additional mechanisms that allow a program to wait for I/O activity on either queue^①.

Despite the apparent difficulties, it is possible to accommodate both clients and servers with both styles of demultiplexing. In the first style, a client that communicates with only one remote application must choose a local protocol port number not used by any other local program. In the second style, a server must use a wildcard^② facility as Figure 10.2 illustrates. The source specification labeled ANY represents a wildcard that matches any source (any IP address and any protocol port number). At a given time, the system allows at most one wildcard for a given destination port. When a datagram arrives, the implementation checks to see if the source and destination matches a specified source-destination pair before checking the wildcard. Thus, in the example, if a datagram arrives with destination port 200, source port 397, and source IP address 192.5.48.3, the system will place it in the queue for application 1. Similarly, the system will place datagrams with destination port 200, source port 40, and source IP address 128.10.2.26 in the queue for application 2. The system uses the wildcard specification to match other datagrams sent to port 200 and places them in the queue for application 4.

10.3 UDP

Our example implementation uses the style of demultiplexing that chooses a queue for incoming datagrams using only the destination protocol port. We selected this style because it keeps demultiplexing efficient and allows application programs to communicate with multiple remote sites simultaneously. After reviewing the definition of data structures used for UDP, we will examine how the software processes arriving datagrams, and how it sends outgoing datagrams.

10.3.1 UDP Declarations

Structure udp in file udp.h defines the UDP datagram format. In addition to the 16-bit source and destination protocol port numbers, the UDP header contains a 16-bit datagram length field and a 16-bit checksum.

```
/* udp.h */\n\n/* User Datagram Protocol (UDP) constants and formats */\n\n#define U_HLEN 8           /* UDP header length in bytes */\n\n
```

^① Berkeley UNIX providers a select system call to permit an application to await activity on any one of a set of I/O descriptors.

^② Adding a wildcard facility makes the second style functionally equivalent to the first style.



```
/* maximum data in UDP packet      */
#define U_MAXLEN (IP_MAXLEN-(IP_MINHLEN<<2)-U_HLEN)

struct udp {                                /* message format of DARPA UDP */
    unsigned short u_src;        /* source UDP port number */
    unsigned short u_dst;        /* destination UDP port number */
    unsigned short u_len;        /* length of UDP data */
    unsigned short u_cksum;     /* UDP checksum (0 => none) */
    char u_data[U_MAXLEN];     /* data in UDP message */
};

/* UDP constants */

#define ULPORT          2050 /* initial UDP local "port" number */

/* assigned UDP port numbers */

#define UP_ECHO          7   /* echo server */
#define UP_DISCARD        9   /* discard packet */
#define UP_USERS          11  /* users server */
#define UP_DAYTIME        13  /* day and time server */
#define UP_QOTD           17  /* quote of the day server */
#define UP_CHARGEN        19  /* character generator */
#define UP_TIME            37  /* time server */
#define UP_WHOIS          43  /* who is server (user information) */
#define UP_DNAME          53  /* domain name server */
#define UP_TFTP            69  /* trivial file transfer protocol server*/
#define UP_RWHO           513 /* remote who server (ruptime) */
#define UP_RIP             520 /* route information exchange (RIP) */

#ifndef Ndg
#define UPPS            1   /* number of xinu ports used to */
#else
/* demultiplex udp datagrams */
#define UPPS            Ndg
#endif
#define UPPLEN          50   /* size of a demux queue */

/* mapping of external network UDP "port" to internal Xinu port */

struct upq {                                /* UDP demultiplexing info */
    Bool up_valid; /* is this entry in use? */
}
```



```
unsigned short      up_port; /* local UDP port number      */
int      up_pid;        /* port for waiting reader   */
int      up_xport; /* corresponding Xinu port on   */
};                  /* which incoming pac. queued   */

extern  struct    upq  upqs[];
extern  int     udpmutex; /* for UDP port searching mutex      */
```

In addition to the declaration of the UDP datagram format, udp.h contains symbolic constants for values assigned to the most commonly used UDP protocol port numbers. For example, a TFTP server always operates on port 69, while RIP uses port 520.

10.3.2 Incoming Datagram Queue Declarations

UDP software divides the data structures that store incoming datagrams into two conceptual pieces: the first piece consists of queues for arriving datagrams, while the second piece contains mapping information that UDP uses to select a queue. The first piece is part of the interface between UDP and application programs that need to extract arriving datagrams. The second piece is part of the operating system — UDP software uses it to select a queue, but application programs cannot access it. File dgram.h contains the declaration of the queues used by application programs.

```
/* dgram.h */

/* datagram pseudo-device control block */

struct  dgblk  {           /* datagram device control block*/
    int dg_dnum;        /* device number of this device */
    int dg_state;       /* whether this device allocated*/
    u_short dg_lport;   /* local datagram port number */
    u_short dg_fport;   /* foreign datagram port number */
    int dg_xport;       /* incoming packet queue */
    int dg_upq;         /* index of our upq entry */
    int dg_mode;        /* mode of this interface */
};

/* datagram psuedo-device state constants */

#define DGS_FREE 0          /* this device is available */
#define DGS_INUSE1          /* this device is in use */

#define DG_TIME 30          /* read timeout (tenths of sec) */
```



```
/* constants for dg pseudo-device control functions */

#define DG_SETMODE    1          /* set mode of device      */
#define DG_CLEAR     2          /* clear all waiting datagrams   */

/* constants for dg pseudo-device mode bits */

#define DG_NMODE 001          /* normal (datagram) mode    */
#define DG_DMODE 002          /* data-only mode           */
#define DG_TMODE 004          /* timeout all reads        */
#define DG_CMODE 010          /* generate checksums (default) */

/* structure of xinugram as dg interface delivers it to user */

struct xgram {           /* Xinu datagram (not UDP)  */
    IPAddr xg_fip;        /* foreign host IP address */
    unsigned short xg_fport; /* foreign UDP port number */
    unsigned short xg_lport; /* local UDP port number   */
    u_char xg_data[U_MAXLEN]; /* maximum data to/from UDP */
};

#define XGHLEN 8 /* error in ( (sizeof(struct xgram)) - U_MAXLEN) */

/* constants for port specifications on UDP open call */

#define ANYFPORT 0 /* accept any foreign UDP port */
#define ANYLPORT 0 /* assign a fresh local port num */

extern struct dgb lk dgtab[Ndg];
extern int dgmutex;
```

Although the file contains many details beyond the scope of this chapter, two declarations are pertinent. The basic data structure used to store incoming datagrams consists of an array, dgtab. Each entry in the array is of type dgb lk. Think of dgtab as a set of queues; there will be one active entry in dgtab for each local UDP protocol port in use. Field dg_lport specifies the local UDP protocol port number, and field dg_xport defines the queue of datagrams that have arrived destined for that port, field dg_state specifies whether the entry is in use (DGS_INUSE) or currently unallocated (DGS_FREE).



In addition to defining the structure used for demultiplexing, dgram.h also specifies the format of datagrams transferred between an application program and the UDP protocol software, instead of passing the UDP datagram to applications, UDP software defines a new format in structure xgram. Recall that we use the style of demultiplexing where an application that opens a given protocol port number receives all datagrams sent to that port. The system passes datagrams to the application in xgram format, so the application can determine the sender's IP address as well as the sender's protocol port number.

10.3.3 Mapping UDP port numbers To Queues

UDP uses the destination port number on an incoming datagram to choose the correct entry in dgtab. It finds the mapping in array upqs, declared in file udp.h. Procedure udp_in, shown later, compares the destination protocol port number to field up_port in each entry of the upqs array until it finds a match. It then uses field up_xport to determine the identity of the Xinu port used to enqueue the datagram.

Separating the mapping in upqs from the queues in dgtab may seem wasteful because the current implementation uses a linear search for the mapping. However, linear search only suffices for systems that have few active UDP ports. Systems with many ports need to use a more efficient lookup scheme like hashing. Separating the data structure used to map ports from the data structure used for datagram queues makes it possible to modify the mapping algorithm without changing the data structures in the application interface. The separation also makes it possible for the operating system to use UDP directly, without relying on the same interface as application programs,

10.3.4 Allocating A Free Queue

Because our example code uses a sequential search of the upqs array, allocation of an entry is straightforward.

```
/* upalloc.c - upalloc */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*
 * upalloc - allocate a UDP port demultiplexing queue
 */
int upalloc(void)
{
```



```
    struct    upq  *pup;
    int       i;

    wait(udpmutex);
    for (i=0 ; i<UPPS ; i++) {
        pup = &upqs[i];
        if (!pup->up_valid) {
            pup->up_valid = TRUE;
            pup->up_port = -1;
            pup->up_pid = BADPID;
            pup->up_xport = pcreate(UPPLEN);
            signal(udpmutex);
            return i;
        }
    }
    signal(udpmutex);
    return SYSERR;
}

struct    upq  upqs[UPPS];
```

Procedure upalloc searches the array until it finds an entry not currently used, fills in the fields, creates a Xinu port to serve as the queue of incoming datagrams, and returns the index of the entry to the caller.

10.3.5 Converting To And From Network Byte Order

Two utility procedures handle conversion of UDP header fields between network byte order and local machine byte order, Procedure udpnet2h handles conversion to the local machine order for incoming datagrams. The code is self-explanatory.

```
/* udpnet2h.c - udpnet2h */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * udpnet2h - convert UDP header fields from net to host byte order
 */
udpnet2h(pudp)
```



```
    struct    udp *pudp;
{
    pudp->u_src = net2hs(pudp->u_src);
    pudp->u_dst = net2hs(pudp->u_dst);
    pudp->u_len = net2hs(pudp->u_len);
}
```

A related procedure, `udph2net`, converts header fields from the local host byte order to standard network byte order.

```
/* udph2net.c - udph2net */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * udph2net - convert UDP header fields from host to net byte order
 */
udp2net(pudp)
struct    udp *pudp;
{
    pudp->u_src = hs2net(pudp->u_src);
    pudp->u_dst = hs2net(pudp->u_dst);
    pudp->u_len = hs2net(pudp->u_len);
}
```

10.3.6 Processing An Arriving Datagram

A procedure in the pseudo-network interface calls procedure `udp_in` when UDP datagram arrives destined for the local machine. It passes arguments that specify the index of the network interface on which the packet arrived and the address of a buffer containing the packet.

```
/* udp_in.c - udp_in */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*
 * udp_in - handle an inbound UDP datagram

```



```
*-----  
*/  
  
int  
udp_in(pni, pep)  
struct netif *pni;  
struct ep *pep;  
{  
    struct ip *pip = (struct ip *)pep->ep_data;  
    struct udp *pudp = (struct udp *)pip->ip_data;  
    struct upq *pup;  
    unsigned short dst;  
    int i;  
  
    if (pudp->u_cksum && udpcksum(pip)) {  
        freebuf(pep);  
        return SYSERR; /* checksum error */  
    }  
    udpnet2h(pudp); /* convert UDP header to host order */  
    dst = pudp->u_dst;  
    wait(udpmutex);  
    for (i=0 ; i<UPPS ; i++) {  
        pup = &upqs[i];  
        if (pup->up_port == dst) {  
            /* drop instead of blocking on psend */  
            if (pcount(pup->up_xport) >= UPPLEN) {  
                signal(udpmutex);  
                freebuf(pep);  
                UdpInErrors++;  
                return SYSERR;  
            }  
            psend(pup->up_xport, (WORD)pep);  
            UdpInDatagrams++;  
            if (!isbadpid(pup->up_pid)) {  
                send(pup->up_pid, OK);  
                pup->up_pid = BADPID;  
            }  
            signal(udpmutex);  
            return OK;  
        }  
    }  
    signal(udpmutex);
```



```
    UdpNoPorts++;
    icmp(ICK_DESTUR, ICC_PORTUR, pip->ip_src, pep);
    return OK;
}

int udpmutex;
```

Udp_in first checks to see whether the sender supplied the optional checksum (by testing to see if the checksum field is nonzero). It calls udpcksum to verify the checksum if one is present. The call will result in zero if the packet contains a valid checksum. If the checksum is both nonzero and invalid, udp_in discards the UDP datagram without further processing. Udp_in also calls udpnet2h to convert the header fields to the local machine byte order.

After converting the header, udp_in demultiplexes the datagram, and it searches the set of datagram queues (array upqs) until it finds one for the destination UDP port. If the port is not full, udp_in calls psend to deposit the datagram and then calls send to send a message to whichever process is awaiting the arrival. If the queue is full, udp_in records an overflow error and discards the datagram.

If udp_in searches the entire set of datagram queues without finding one reserved for the destination port on the incoming datagram, it means that no application program has agreed to receive datagrams for that port. In such cases, udp_in must call icmp to send an ICMP destination unreachable message back to the original source.

10.3.7 UDP Checksum Computation

Procedure udpcksum computes the checksum of a UDP datagram. Like the procedure cksum described earlier, it can be used to generate a checksum (by setting the checksum header field to zero), or to verify an existing checksum. However, the UDP checksum differs from earlier checksums in one important way:

The UDP checksum covers the UDP datagram plus a pseudo-header that includes the IP source and destination addresses, UDP length, and UDP protocol type identifier.

When computing the checksum for an outgoing datagram, the protocol software must find out what values will be used when the UDP message is encapsulated in an IP datagram. When verifying the checksum for a message that has arrived, UDP extracts values from the IP datagram that carried the message. Including the IP source and destination addresses in the checksum provides protection against misrouted datagrams.

Procedure udpcksum does not assemble a pseudo-header in memory. Instead, it



picks up individual fields from the IP header and includes them in the checksum computation. For example, udpcksum assigns psh the address of the IP source field in the datagram and adds the four 16-bit quantities starting at that address, which include the IP source and destination addresses.

```
/* udpcksum.c - udpcksum */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#define UDP_ALEN      IP_ALEN      /* length of src+dst, in shorts */

/*-----
 * udpcksum - compute a UDP pseudo-header checksum
 *-----
 */
unsigned short udpcksum(pip)
struct ip *pip;
{
    struct udp *pudp = (struct udp *)pip->ip_data;
    unsigned short *psh;
    unsigned long sum;
    short len = net2hs(pudp->u_len);
    int i;

    sum = 0;

    psh = (unsigned short *)&pip->ip_src;
    for (i=0; i<IP_ALEN; ++i)
        sum += *psh++;

    psh = (unsigned short *)pudp;
    sum += hs2net(IPT_UDP + len);
    if (len & 0x1) {
        ((char *)pudp)[len] = 0; /* pad */
        len += 1; /* for the following division */
    }
    len /= 2; /* convert to length in shorts */

    for (i=0; i<len; ++i)
        sum += *psh++;

}
```



```
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);

    return (short)(~sum & 0xffff);
}
```

10.4 UDP Output Processing

Before an application program can communicate using UDP, it needs a local UDP port number. Servers, which use well-known ports, request a specific port assignment from the operating system. Usually, clients do not need a specific port — they can use an arbitrary port number. However, because our system demultiplexes using only destination port numbers, a client must be assigned a unique port number. Procedure `udpnntp` generates a UDP port number that is not in use.

```
/* udpnntp.c - udpnntp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * udpnntp - return the next available UDP local "port" number
 * N.B.: assumes udpmutex HELD
 */
unsigned short udpnntp()
{
    static unsigned short lastport = ULPORT;
    Bool inuse = TRUE;
    struct upq *pup;
    int i;

    while (inuse) {
        lastport++;
        if (lastport == 0)
            lastport = ULPORT;
        inuse = FALSE;
        for (i=0; !inuse && i<UPPS ; i++) {
            pup = &upqs[i];
            inuse = pup->up_valid && pup->up_port == lastport;
        }
    }
}
```



```
    }

    return lastport;
}
```

To generate an unused port number, `udpnntp` first increments the global counter `lastport`. It then iterates through the set of UDP input queues to see if any application program has already been assigned `lastport`. Usually, the iteration does not find a match, and `udpnntp` returns `lastport` to the caller. If it does find a match, `udpnntp` increments `lastport` and tries again.

10.4.1 Sending A UDP Datagram

When an application program generates UDP output, it transfers control to the operating system and eventually calls procedure `udpsend` to send the UDP datagram.

```
/* udpsend.c - udpsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/*
 * udpsend - send one UDP datagram to a given IP address
 */
int udpsend(IPAddr fip, u_short fport, u_short lport, struct ep *pep,
             unsigned datalen, Bool docksum)
{
    Ipaddr    fip;
    unsigned short    fport, lport;
    struct    ep    *pep;
    int    datalen;
    Bool docksum;
    {
        struct    ip    *pip = (struct ip *) pep->ep_data;
        struct    udp    *pudp = (struct udp *) pip->ip_data;
        struct    route    *prt, *rtget();
        struct    hg    *phg, hglookup();
        int    ttl;

        pudp->u_src = lport;
        pudp->u_dst = fport;
```



```
pudp->u_len = U_HLEN+datalen;
pudp->u_cksum = 0;
udph2net(pudp);
if (docksum) {
    prt = rtget(fip, RTF_LOCAL);
    if (prt == NULL) {
        IpOutNoRoutes++;
        freebuf(pep);
        return SYSERR;
    }
    blkcopy(pip->ip_dst, fip, IP_ALEN);
    if (prt->rt_ifnum == NI_LOCAL)
        blkcopy(pip->ip_src, pip->ip_dst, IP_ALEN);
    else
        blkcopy(pip->ip_src, nif[prt->rt_ifnum].ni_ip,
                IP_ALEN);
    rtfree(prt);
    pudp->u_cksum = udpcksum(pip);
    if (pudp->u_cksum == 0)
        pudp->u_cksum = ~0;
}
UdpOutDatagrams++;
if (IP_CLASSD(fip)) {
    wait(HostGroup.hi_mutex);
    phg = hglookup(NI_PRIMARY, fip);
    if (phg)
        ttl = phg->hg_ttl;
    else
        ttl = 1;
    signal(HostGroup.hi_mutex);
}
return ipsend(fip, pep, U_HLEN+datalen, IPT_UDP, IPP_NORMAL, ttl);
}
```

Because gateways have multiple network connections, they have multiple IP addresses. Before udpsend can compute the UDP checksum, it needs to know which address IP will use as the source address for the IP datagram that carries the message. To find out, udpsend calls procedure rtget, passing it the destination IP address as an argument. Once it determines a route, udpsend extracts the network interface, from which it obtains the IP datagram source address.



Once it has computed the source address for the IP datagram, `udpsend` fills in the remaining fields of the UDP header, calls `udpcksum` to compute the checksum, and calls `ipsend` to pass the resulting IP datagram to IP for routing and transmission.

10.5 Summary

UDP provides both pairwise communication between peer programs and many-one communication between clients and a server. While the two basic styles of demultiplexing both support clients and servers, each has advantages and disadvantages. The example code demultiplexes using only the destination protocol port number, and makes the creation of servers trivial. To help support clients, the system includes a procedure that generates a unique (unused) protocol port number on demand.

Both UDP input and UDP output are straightforward. The IP process executes the UDP input procedure, which demultiplexes datagrams and deposits each on a queue associated with the destination protocol port. Application programs allocate a port used for transmission and then call the output procedures to create and send UDP datagrams.

The UDP checksum includes fields from a pseudo-header that are used to verify that the IP datagram carrying UDP contained the correct IP source and destination addresses. For input, UDP can obtain values for pseudo-header fields from the IP datagram that carries the UDP message. For output, the pseudo-header processing complicates the UDP checksum computation because it forces UDP to determine which address IP will use as the source address for the datagram.

10.6 FOR FURTHER STUDY

Postel [RFC 768] defines the UDP protocol and specifies the message format. The host requirements document [RFC 1122] provides further clarification. Leffler, McKusick, Karels, and Quarterman [1989] presents details of the 4BSD UNIX implementation.

10.7 EXERCISES

1. Read the RFC carefully to determine whether all pseudo-header fields used to verify the UDP checksum must be taken from the IP datagram that carries the UDP datagram. Can constants ever be used? Explain.
2. Read the 4BSD UNIX documentation. Which style of demultiplexing does it use?
3. Does your local system allow you to specify the size of a UDP input queue? If so, how can you choose a reasonable size?



4. Look at `udpksum` carefully. What does it do if the computed checksum happens to be zero? Why?
5. Explain why `udpnntp` skips ports between 0 and ULPORT (2050) when it generates a local port number.
6. How many UDP ports does a typical timesharing system use simultaneously? How many does a typical scientific workstation use?
7. Our example code uses sequential search to find a UDP port for demultiplexing. Devise a hashing scheme that lowers lookup time. Will your scheme ever increase lookup time? (Hint: consider the previous exercise).



11 TCP: Data Structures And Input Processing

11.1 Introduction

TCP is the most complex of all protocols in the suite of Internet protocols. It provides reliable, flow-controlled, end-to-end, stream service between two machines of arbitrary processing speed using the unreliable IP mechanism for communication. Like most reliable transport protocols, TCP uses timeout with retransmission to achieve reliability. However, unlike most other transport protocols, TCP is carefully constructed to work correctly even if datagram are delayed, duplicated, lost, delivered out of order, or delivered with the data corrupted or truncated. Furthermore, TCP allows communication machines to reboot and reestablish connections at arbitrary times without causing confusion about which connections are open and which are new.

This chapter examines the global organization of TCP software and describes the data structures TCP uses to manage information about connections. Chapter 12 describes the details of connection management and implementation of the TCP finite state machine used for input. Chapter 13 discusses output and the finite state machine used to control it. Chapters 14 through 16 discuss the details of timer management, estimation of round trip times, retransmission, and miscellaneous details such as urgent data processing.

11.2 Overview Of TCP Software

Recall from Chapter 2 that our implementation of TCP uses three processes. One process handles incoming segments, another manages outgoing segments, and the third is a timer that manages delayed events such as retransmission timeout. In theory, using separate processes isolates the input, output, and event timing parts of TCP and permits us to design each piece independently. In practice, however, the processes interact closely. For example, the input and output processes must cooperate to match incoming



acknowledgements with outgoing segments and cancel the corresponding timer retransmission event. Similarly, the output and timer processes interact when the output process schedules a retransmission event or when the timer triggers a retransmission.

11.3 Transmission Control Blocks

TCP coordinates the activities of transmission, reception, and retransmission for each TCP connection through a data structure shared by all processes. The data structure is known as a transmission control block or TCB. TCP maintains one TCB for each active connection. The TCB contains all information about the TCP connection, including the addresses and port numbers of the connection endpoints, the current round-trip time estimate, data that has been sent or received, whether acknowledgement or retransmission is needed, and any statistics TCP gathers about the use of the connection.

Although the protocol standard defines the notion of the TCB and suggests some of the contents, it does not dictate all the details. Thus, a designer must choose the exact contents. Our example implementation places the information in structure tcb. In most cases field names match the names used in the protocol standard.

```
/* tcb.h */

/* TCP endpoint types */

#define TCPT_SERVER      1
#define TCPT_CONNECTION   2
#define TCPT_MASTER       3

/* TCP process info */

extern PROCESS      tcpinp();
#define TCPISTK        4096    /* stack size for TCP input */
#define TCPIPRI         100     /* TCP runs at high priority */
#define TCPINAM        "tcpinp" /* name of TCP input process */
#define TCPIARGC        0       /* count of args to tcpin */

extern PROCESS      tcpout();
#define TCPOSTK        4096    /* stack size for TCP output */
#define TCPOPRI         100     /* TCP runs at high priority */
#define TCPONAM        "tcpout" /* name of TCP output process */
#define TCPOARGC        0       /* count of args to tcpout */

#define TCPQLEN         20      /* TCP process port queue length */
```



```
/* TCP exceptional conditions */

#define TCPE_RESET      -1
#define TCPE_REFUSED    -2
#define TCPE_TOOBIG     -3
#define TCPE_TIMEDOUT   -4
#define TCPE_URGENTMODE -5
#define TCPE_NORMALMODE -6

/* string equivalents of TCPE_*, in "tcpswitch.c" */
extern char *tcperror[];

#define READERS        1
#define WRITERS        2

/* tcb_flags */

#define TCBF_NEEDOUT   0x01 /* we need output          */
#define TCBF_FIRSTSEND  0x02 /* no data to ACK           */
#define TCBF_GOTFIN    0x04 /* no more to receive       */
#define TCBF_RDONE     0x08 /* no more receive data to process */
#define TCBF_SDONE     0x10 /* no more send data allowed */
#define TCBF_DELACK    0x20 /* do delayed ACK's         */
#define TCBF_BUFFER    0x40 /* do TCP buffering (default no) */
#define TCBF_PUSH 0x80 /* got a push; deliver what we have */
#define TCBF_SNDFIN   0x100 /* user process has closed; send a FIN */
#define TCBF_RUPOK    0x200 /* receive urgent pointer is valid */
#define TCBF_SUPOK    0x400 /* send urgent pointer is valid */

/* aliases, for user programs */

#define TCP_BUFFER     TCBF_BUFFER
#define TCP_DELACK    TCBF_DELACK

/* receive segment reassembly data */

#define NTCPFRAG 10

struct tcb {
    short    tcb_state;    /* TCP state                  */
    short    tcb_ostate;   /* output state               */
}
```



```
short      tcb_type; /* TCP type (SERVER, CLIENT)          */
int       tcb_mutex;    /* tcb mutual exclusion           */
short      tcb_code;   /* TCP code for next packet      */
short      tcb_flags;   /* various TCB state flags      */
short      tcb_error;  /* return error for user side   */

IPAddr    tcb_rip;   /* remote IP address           */
u_short   tcb_rport;  /* remote TCP port             */
IPAddr    tcb_lip;   /* local IP address            */
u_short   tcb_lport;  /* local TCP port              */
struct    netif     *tcb_pni; /* pointer to our interface   */

tcpseq    tcb_suna;  /* send unacked                */
tcpseq    tcb_snxt;  /* send next                   */
tcpseq    tcb_slast; /* sequence of FIN, if TCBF_SNDFIN */
u_long    tcb_swindow; /* send window size (octets)   */
tcpseq    tcb_lwseq; /* sequence of last window update */
tcpseq    tcb_lwack; /* ack seq of last window update */
u_int     tcb_cwnd;  /* congestion window size (octets) */
u_int     tcb_ssthresh; /* slow start threshold (octets) */
u_int     tcb_smss;  /* send max segment size (octets) */
tcpseq    tcb_iss;   /* initial send sequence      */

int      tcb_srt;   /* smoothed Round Trip Time   */
int      tcb_rtde;  /* Round Trip deviation estimator */
int      tcb_persist; /* persist timeout value      */
int      tcb_keep;  /* keepalive timeout value    */
int      tcb_rexmt; /* retransmit timeout value   */
int      tcb_rexmtcount; /* number of rexmts sent     */

tcpseq    tcb_rnext; /* receive next                */
tcpseq    tcb_rupseq; /* receive urgent pointer     */
tcpseq    tcb_supseq; /* send urgent pointer        */

int      tcb_lqsize; /* listen queue size (SERVERs) */
int      tcb_listenq; /* listen queue port (SERVERs) */
struct tcb *tcb_pptcb; /* pointer to parent TCB (for ACCEPT) */
int      tcb_ocsem;  /* open/close semaphore        */
int      tcb_dvnum;  /* TCP slave pseudo device number */

int      tcb_ssema; /* send semaphore             */
```



```
    u_char    *tcb_sndbuf; /* send buffer */  
    u_int     tcb_sbstart; /* start of valid data */  
    u_int     tcb_sbcount; /* data character count */  
    u_int     tcb_sbsize; /* send buffer size (bytes) */  
  
    int      tcb_rsema; /* receive semaphore */  
    u_char    *tcb_rcvbuf; /* receive buffer (circular) */  
    u_int     tcb_rbstart; /* start of valid data */  
    u_int     tcb_rbcnt; /* data character count */  
    u_int     tcb_rbsize; /* receive buffer size (bytes) */  
    u_int     tcb_rmss; /* receive max segment size */  
    tcpseq   tcb_cwin; /* seq of currently advertised window */  
    int      tcb_rsegq; /* segment fragment queue */  
    tcpseq   tcb_finseq; /* FIN sequence number, or 0 */  
    tcpseq   tcb_pushseq; /* PUSH sequence number, or 0 */  
};  
/* TCP fragment structure */  
  
struct tcpfrag {  
    tcpseq   tf_seq;  
    int      tf_len;  
};  
/* TCP control() functions */  
  
#define  TCPC_LISTENQ 0x01 /* set the listen queue length */  
#define  TCPC_ACCEPT 0x02 /* wait for connect after passive open */  
#define  TCPC_STATUS 0x03 /* return status info (all, for master) */  
#define  TCPC_SOPT 0x04 /* set user-selectable options */  
#define  TCPC_COPT 0x05 /* clear user-selectable options */  
#define  TCPC_SENDURG 0x06 /* write urgent data */  
  
/* global state information */  
  
extern int tcps_oport; /* Xinu port to start TCP output */  
extern int tcps_iport; /* Xinu port to send TCP input packets */  
extern int tcps_lqsize; /* default SERVER queue size */  
extern int tcps_tmutex; /* tcb table mutex */  
  
extern int (*tcpswitch[])( ), (*tcposwitch[])( );  
  
#ifdef Ntcp
```



```
extern struct tcb  tcbtab[];  
#endif
```

While it is not possible to understand all fields in the TCB without looking at the procedures that use them, the meaning of some fields should be obvious. For example, in addition to fields that specify the current input and output states of the connection (tcb_state and tcb_ostate), the tcb structure includes fields that specify: a mutual exclusion semaphore (tcb_mutex), the local and remote IP addresses (tcb_lip and tcb_rip), the local and remote port numbers (tcb_lport and tcb_rport), and the network interface used (tcb_pni).

Of course, the tcb structure contains information used when sending segment: the receiver's current window size (tcb_swindow), the next sequence number to send (tcb_snext), the lowest unacknowledged byte in the sequence (tcb_sun), the congestion window size (tcb_cwnd), the slow-start threshold (tcb_ssthresh), and the maximum allowable segment size (tcb_smss).

For retransmission, the tcb structure maintains the smoothed round trip time estimate (tcb_srt), an estimate of the deviation in round trip times (tcb_rtde), the retransmission timeout value (tcb_rexmt), and a count of consecutive retransmissions (tcb_rexmtcount).

Additional fields in tcb contain values used for reception. In addition to the address of the receive buffer (tcb_rcvbuf), the tcb contains fields that specify the start of valid data (tcb_rbstart), a count of characters in the receive buffer (tcb_rbcnt), the allowable maximum segment size (tcb_rmss), and the sequence number of the last advertised window (tcb_cwin). We will discuss the remaining fields later.

Because segments can arrive out of order, TCP must store information about blocks of data as they arrive until it can assemble them into a contiguous stream. TCP keeps the information on a linked list, using structure tcpfrag, defined in tcb.h, to store the starting sequence number and length of each block.

11.4 TCP Segment Format

Structure tcp defines the TCP segment format. File tcp.h contains the declaration along with symbolic constants that define the meaning of bits in field tcp_code.

```
/* tcp.h - TCP_HLEN, SEQCMP */  
  
typedef  long tcpseq;  
  
/*  
 * SEQCMP - sequence space comparator
```



```
* This handles sequence space wrap-around. Overflow/Underflow makes
* the result below correct ( -, 0, + ) for any a, b in the sequence
* space. Results: result    implies
*
*          -   a < b
*
*          0   a = b
*
*          +   a > b
*/
#define SEQCMP(a, b) ((a) - (b))

/* tcp packet format */

struct tcp {
    unsigned short      tcp_sport;      /* source port */           */
    unsigned short      tcp_dport;      /* destination port */      */
    tcpseq      tcp_seq; /* sequence */           */
    tcpseq      tcp_ack; /* acknowledged sequence */ */
    unsigned char      tcp_offset;
    unsigned char      tcp_code; /* control flags */        */
    unsigned short      tcp_window; /* window advertisement */ */
    unsigned short      tcp_cksum; /* check sum */           */
    unsigned short      tcp_urp; /* urgent pointer */        */
    unsigned char      tcp_data[1];
};

/* TCP Control Bits */

#define TCPF_URG 0x20 /* urgent pointer is valid */           */
#define TCPF_ACK 0x10 /* acknowledgement field is valid */      */
#define TCPF_PSH 0x08 /* this segment requests a push */       */
#define TCPF_RST 0x04 /* reset the connection */           */
#define TCPF_SYN 0x02 /* synchronize sequence numbers */      */
#define TCPF_FIN 0x01 /* sender has reached end of its stream */ */

#define TCPMHLEN 20 /* minimum TCP header length */           */
#define TCPHOFFSET 0x50 /* tcp_offset value for TCPMHLEN */ */
#define TCP_HLEN(ptcp) (((ptcp)->tcp_offset & 0xf0)>>2)

/* TCP Options */

#define TPO_EOOL 0 /* end Of Option List */           */
#define TPO_NOOP 1 /* no Operation */           */
```



```
#define TPO_MSS 2 /* maximum Segment Size */
```

File `tcp.h` also defines the macro function `TCP_HLEN` that computes the length of a TCP segment header in octets. The header length, measured in 32-bit words, is stored in the high-order 4 bits of the 8-bit offset field. To compute the header length in bytes, the macro must multiply the length in 32-bit words by 4. To do so, it references the entire octet, computes a logical and to extract the length bits, and shifts them into position.

11.5 Sequence Space Comparison

TCP assigns integers, called sequence numbers (or sequence values) to octets in the data stream. When it sends data in a segment, TCP includes the sequence number in the segment header. The receiving TCP uses the sequence numbers to detect when segments arrive out of order, and to reorder them into the correct linear sequence. TCP chooses the initial starting sequence number for each connection at random to prevent delayed or duplicated packets from confusing the 3-way handshake.

The set of all possible sequence integers is known as the TCP sequence space. Because the sequence field in the TCP segment header has fixed size (32 bits), it is possible for sequence numbers to reach the end of the sequence space and wrap around to zero. TCP software needs to make comparisons between sequence values so it can determine whether the sequence number in one segment is greater or less than the sequence number in another. If one uses conventional comparisons, small values like zero will always compare less than large values, even though zero "follows" the largest possible integer when sequence numbers wrap around the end of the sequence space.

Surprisingly, conventional computer arithmetic can be used to establish a correct relationship between two sequence values, as long as the sequence space size equals the range of integer values on the machine, and the values being compared do not differ by more than one-half the integer space. With current computers and networks, TCP never needs to compare two sequence numbers that differ by more than one-half the largest integer because computers cannot generate output fast enough to wrap around the sequence space before datagrams timeout.

If the two sequence numbers compared are close together, simple integer subtraction yields the desired result. Integer underflow takes care of the case where a very large number is subtracted from a very small number. That is, if a and b are two sequence numbers that differ by no more than one-half the largest possible integer value, the following is true:



result of a b	relationship in Sequence Space
0	a precedes b
+	a equals b a follows b

Figure 11.1 The result of subtracting two sequence values that differ by no more than one-half the largest sequence space value.

We can summarize:

TCP uses integer subtraction to compare TWO sequence values because it can assume they never differ by more than one-half of the sequence space. In such cases, integer underflow produces the desired result when comparing a very large integer with a very small one.

Macro SEQCMP in file tcp.h implements sequence space comparison. The TCP code uses SEQCMP when doing comparisons to help the reader clearly distinguish between conventional subtraction and sequence space comparison.

11.6 TCP Finite State Machine

Conceptually, TCP uses a finite state machine to control all interactions. Each end of a TCP connection implements a copy of the state machine and uses it to control actions taken when a segment arrives. Figure 11.2 shows the TCP finite state machine and transitions among states.

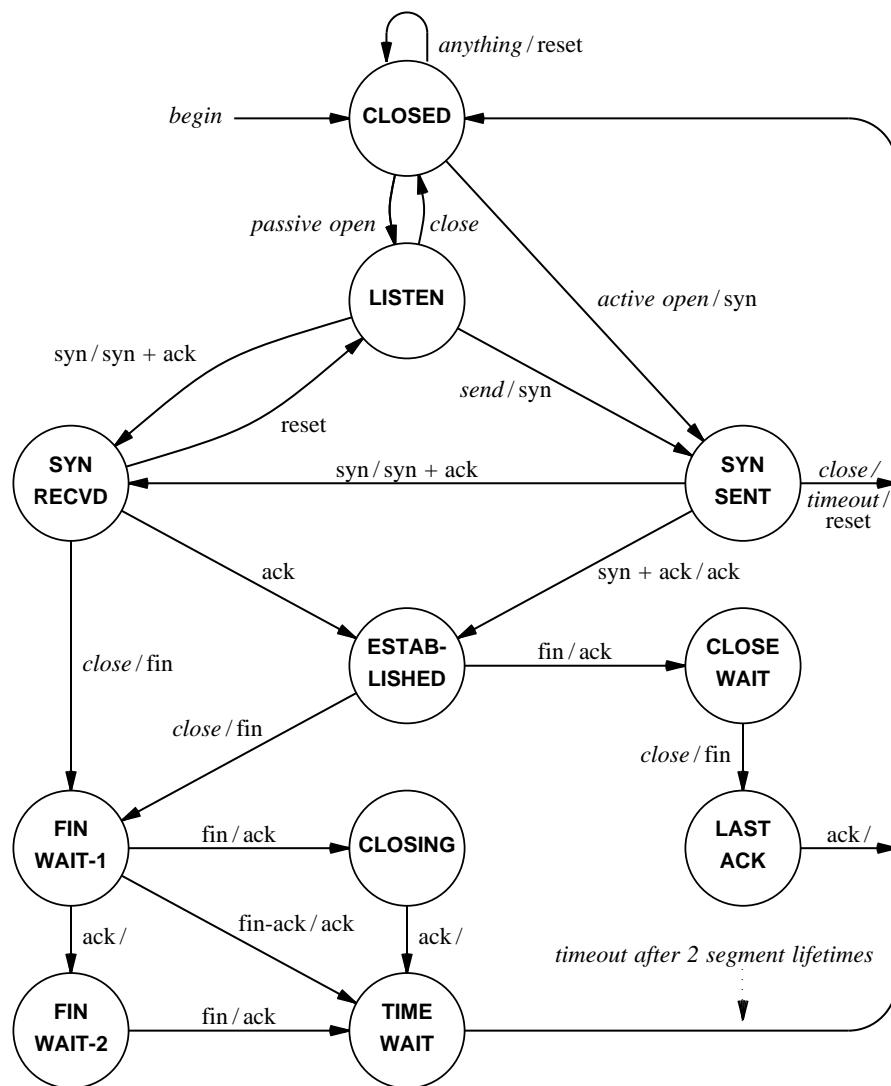


Figure 11.2 The TCP Finite State Machine that controls processing.

In theory, the finite state machine completely specifies how TCP on one machine interacts with TCP on another. In practice, however, the finite state machine does not fully specify interactions. Instead, the machine specifies only the macroscopic state of TCP, while additional variables further specify the details or microscopic state. More important, because the macroscopic transitions specified by the state machine do not control output or retransmission, such events must be handled separately. We can summarize:

The finite state machine specifies how TCP makes macroscopic state



transitions in response to input or user commands; an implementation contains a separate mechanism that makes microscopic state transitions to control output and retransmission.

11.7 Example State Transition

To understand the TCP finite state machine, consider an example of the three-way handshake used to establish a connection between a client and a server. Both the client and server will create an endpoint for communication, and both will have a copy of the finite state machine. The server begins first by issuing a passive open operation, which causes the server's finite state machine to enter the listen state. The server waits in the LISTEN state until a client contacts it. When a client issues an active open, it causes TCP software on its machine to send a SYN segment to the server and to enter the SYN-SENT state.

When the server, which is waiting in the LISTEN state, receives the SYN segment, it replies with a SYN plus an ACK segment, creates a new TCB, and places the new TCB in the SYN-RECEIVED state. When the SYN plus ACK segment arrives at the client, the client TCP replies with an ACK, and moves from the SYN-SENT state to the ESTABLISHED state. Finally, when the client's ACK arrives at the newly created TCB, it also moves to the ESTABLISHED state, which allows data transfer to proceed.

11.8 Declaration Of The Finite State Machine

File tcpfsm.h contains the symbolic constants for states in the TCP finite state machine.

```
/* tcpfsm.h - TCB, EVENT, MKEVENT */

/* TCP states */

#define TCPS_FREE          0
#define TCPS_CLOSED         1
#define TCPS_LISTEN         2
#define TCPS_SYNSENT        3
#define TCPS_SYNRCVD        4
#define TCPS_ESTABLISHED    5
#define TCPS_FINWAIT1       6
#define TCPS_FINWAIT2       7
#define TCPS_CLOSEWAIT      8
#define TCPS_LASTACK        9
#define TCPS_CLOSING        10
```



```
#define TCPS_TIMEWAIT      11

#define NTCPSTATES          12

/* Output States */

#define TCPO_IDLE           0
#define TCPO_PERSIST         1
#define TCPO_XMIT            2
#define TCPO_REXMT           3

#define NTCPOSTATES          4

/* event processing */

#define SEND                0x1
#define PERSIST              0x2
#define RETRANSMIT           0x3
#define DELETE               0x4
#define TMASK                0x7

#define EVENT(x)  ((int)(x) & TMASK)
#define TCB(x)       ((int)(x) >> 3)
#define MKEVENT(timer, tcb)   ((tcb<<3) | (timer & TMASK))

/* implementation parameters */

#define TCP_MAXRETRIES        12 /* max retransmissions before giving up */
#define TCP_TWOMSL             12000 /* 2 minutes (2 * Max Segment Lifetime) */
#define TCP_MAXRXT              2000 /* 20 seconds max rexmt time */
#define TCP_MINRXT              50 /* 1/2 second min rexmt time */
#define TCP_ACKDELAY             20 /* 1/5 sec ACK delay, if TCBF_DELACK */

#define TCP_MAXPRS              6000 /* 1 minute max persist time */

/* second argument to tcpsend(): */

#define TSF_NEWDATA            0 /* send all new data */
#define TSF_REXMT1              /* retransmit the first pending segment */

/* third argument to tcpwr(): */
```



```
#define TWF_NORMAL 0 /* normal data write */
#define TWF_URGENT 1 /* urgent data write */
```

11.9 TCB Allocation And Initialization

11.9.1 Allocating A TCB

Procedures that implement the TCP finite state machine must allocate and initialize a TCB when TCP establishes a connection. To do so they call procedure tcballoc.

```
/* tcballoc.c - tcballoc */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 * tcballoc - allocate a Transmission Control Block
 *-----*
 */
struct tcb *tcballoc()
{
    struct tcb *ptcb;
    int slot;

    wait(tcps_tmutex);
    /* look for a free TCB */

    for (ptcb=&tcbtab[0], slot=0; slot<Ntcp; ++slot, ++ptcb)
        if (ptcb->tcb_state == TCPS_FREE)
            break;
    if (slot < Ntcp) {
        ptcb->tcb_state = TCPS_CLOSED;
        ptcb->tcb_mutex = screate(0);
    } else
        ptcb = (struct tcb *)SYSERR;
    signal(tcps_tmutex);
    return ptcb;
}
```



Tcballoc searches array tcbtab until it finds an unused entry (i.e., an entry with state equal to TCPS_FREE). If such an entry exists, tcballoc changes the state to CLOSED (the initial state of a connection), creates a mutual exclusion semaphore for the TCB, and returns the address of the newly allocated entry to the caller. The call returns with tcb_mutex held. That is, the call returns with exclusive access to the new TCB. If no unused TCB exists, tcballoc returns SYSERR to indicate that an error occurred.

11.9.2 Deallocating A TCB

When a connection terminates, TCP software calls procedure tcbdealloc to free the TCB and allow it to be used again,

```
/* tcbdealloc.c - tcbdealloc */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcbdealloc - deallocate a TCB and free its resources
 * ASSUMES ptcb->tcb_mutex HELD
 */
int tcbdealloc(ptcb)
struct    tcb  *ptcb;
{
    if (ptcb->tcb_state == TCPS_FREE)
        return OK;
    switch (ptcb->tcb_type) {
    case TCPT_CONNECTION:
        tcpkilltimers(ptcb);
        sdelete(ptcb->tcb_ocsem);
        sdelete(ptcb->tcb_ssema);
        sdelete(ptcb->tcb_rsema);
        freemem(ptcb->tcb_sndbuf, ptcb->tcb_sbsize);
        freemem(ptcb->tcb_rcvbuf, ptcb->tcb_rbsize);
        if (ptcb->tcb_rsegq >= 0)
            freeq(ptcb->tcb_rsegq);
        break;
    case TCPT_SERVER:
        pdelete(ptcb->tcb_listenq, 0);
        break;
    default:
```



```
    signal(ptcb->tcb_mutex);
    return SYSERR;
}
ptcb->tcb_state = TCPS_FREE;
sdelete(ptcb->tcb_mutex);
return OK;
}
```

Two special cases complicate the deallocation. As we will see, if a connection has been in progress, tcbdealloc must first call tcpkilltimers to delete any outstanding timer events. It must then delete the send and receive semaphores as well as the memory used to buffer incoming or outgoing data. For a server, tcbdealloc must delete the queue of incoming connection requests. Finally, in all cases, tcbdealloc must delete the mutual exclusion semaphore. We will see how TCP software allocates and uses the semaphores and buffers later.

11.10 Implementation Of The Finite State Machine

A designer must choose between two basic implementations of the TCP finite state machine:

- Table-Driven
- Procedure-Driven

A purely table-driven approach uses a two-dimensional array in which each row corresponds to one state, and each column corresponds to one possible input event or operation that causes transition. Thus, each table entry corresponds to an input event in a particular state. The entry contains the address of a procedure to call to process the event, as well as the integer value of the state to which a transition should occur. A state field in the TCB specifies the current state. When an input event occurs, TCP translates it into one of the possible columns and uses the translated input event and current state to select an entry from the table. TCP uses the entry to select and invoke a procedure, and then updates the state variable.

A procedure-driver approach uses one procedure for each input state. When an event occurs, TCP uses the current state to choose the correct procedure. The procedure processes the input event and updates the state variable,

The table-driven approach works well for implementing a finite state machine that has regular structure, simple semantics, and a relatively complex transition graph. The procedure-driven approach works well for implementing a finite state machine that has few transitions and complex semantics. We have chosen the latter.



Because the TCP state machine contains few states, specifies few transitions among the states, provides complex operations, and includes many exceptions to handle errors, our example implementation uses a procedure-driven approach.

Thus, our implementation has one procedure for each of the states shown in Figure 11.2, and it has a field in the TCB that specifies the current state. TCP calls the procedure for the current state whenever an input segment arrives. In addition, our implementation provides a separate procedure for each local operation (e.g., a server uses a separate procedure to issue a passive open).

11.11 Handling An Input Segment

When IP receives a TCP segment destined for the local machine, it eventually calls `tcp_in` to deliver the segment.

```
/* tcp_in.c - tcp_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 *  tcp_in - deliver an inbound TCP packet to the TCP process
 *-----*
 */
int tcp_in(pni, pep)
struct    netif      *pni;
struct    ep       *pep;
{
    /* drop instead of blocking on psend */

    TcpInSegs++;
    if (pcount(tcps_iport) >= TCPQLEN) {
        freebuf(pep);
        return SYSERR;
    }
    psend(tcps_iport, (int)pep);
    return OK;
}
```



As the code shows, `tcp_in` sends the incoming segment to the TCP input port, from which the TCP input process extracts it. The TCP input process executes procedure `tcpinp`.

```
/* tcpinp.c - tcpinp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpinp - handle TCP segment coming in from IP
 */
PROCESS tcpinp()
{
    struct ep *pep;
    struct ip *pip;
    struct tcp *ptcp;
    struct tcb *ptcb, *tcpdemux();

    tcps_iport = pcreate(TCPQLEN);
    signal(Net.sema);
    while (TRUE) {
        pep = (struct ep *)preceive(tcps_iport);
        if ((int)pep == SYSERR)
            break;
        pip = (struct ip *)pep->ep_data;
        if (tcpcksum(pep)) {
            freebuf(pep);
            continue;
        }
        ptcp = (struct tcp *)pip->ip_data;
        tcpnet2h(ptcp); /* convert all fields to host order */
        ptcb = tcpdemux(pep);
        if (ptcb == 0) {
            tcpreset(pep);
            freebuf(pep);
            continue;
        }
        if (!tcpok(ptcb, pep))
            tcpackit(ptcb, pep);
    }
}
```



```
        else {
            tcpopts(ptcb, pep);
            tcpswitch[ptcb->tcb_state](ptcb, pep);
        }
        if (ptcb->tcb_state != TCPS_FREE)
            signal(ptcb->tcb_mutex);
        freebuf(pep);
    }
}

int tcps_oport, tcps_iprot, tcps_lqsize, tcps_tmutex;
```

Tcpinp repeatedly extracts a segment from the input port, calls tcpksum to verify the checksum, and calls tcpnet2h to convert header fields to local byte order. It uses tcpdemux to find the correct TCB for the segment (calling tcprreset to send a RESET if no TCB exists). It then calls tcopok to verify that the segment is acceptable for the current window, and calls tcopackit to send an acknowledgment if it is not.^① Finally, tcpinp uses tcpopts to handle options in the segment, and then uses array tcpswitch to choose a procedure corresponding to the current input state. The next sections review individual procedures that tcpinp uses.

11.11.1 Converting A TCP Header To Local Byte Order

Procedure tcpnet2h converts integer fields in the TCP header from network standard byte order to local machine byte order.

```
/* tcpnet2h.c - tcpnet2h */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpnet2h - convert TCP header fields from net to host byte order
 */
struct tcp *tcpnet2h(struct tcp *ptcp)
struct    tcp  *ptcp;
{
```

^① According to the protocol standard, the acknowledgement does not confirm receipt of the unacceptable segment; it merely reports the correctly received sequence and the current window size.



```
/* NOTE: does not include TCP options */

ptcp->tcp_sport = net2hs(ptcp->tcp_sport);
ptcp->tcp_dport = net2hs(ptcp->tcp_dport);
ptcp->tcp_seq = net2hl(ptcp->tcp_seq);
ptcp->tcp_ack = net2hl(ptcp->tcp_ack);
ptcp->tcp_window = net2hs(ptcp->tcp_window);
ptcp->tcp_urp = net2hs(ptcp->tcp_urp);
return ptcp;
}
```

11.11.2 Computing The TCP Checksum

TCP computes a checksum the same as UDP. Initially, tcpcksum computes the checksum of a pseudo-header that includes the source and destination IP addresses, segment length, and protocol type value used by IP (the value used in field ip_proto). It then treats the segment as an array of 16-bit values and adds each of them to the checksum. Finally, tcpcksum handles overflow and returns the complement of the checksum to the caller.

```
/* tcpcksum.c - tcpcksum */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 *  tcpcksum - compute a TCP pseudo-header checksum
 *-----
 */
unsigned short tcpcksum(pip)
struct ip *pip;
{
    struct tcp *ptcp = (struct tcp *)pip->ip_data;
    unsigned short *sptr, len;
    unsigned long tcksum;
    unsigned i;

    tcksum = 0;

    sptr = (unsigned short *)&pip->ip_src;
    /* 2*IP_ALEN octets = IP_ALEN shorts... */
    /* they are in net order.           */
}
```



```
for (i=0; i<IP_ALEN; ++i)
    tcksum += *sptr++;
sptr = (unsigned short *)ptcp;
len = pip->ip_len - IP_HLEN(pip);
tcksum += hs2net(IPT_TCP + len);
if (len % 2) {
    ((char *)ptcp)[len] = 0; /* pad */
    len += 1; /* for the following division */
}
len >= 1; /* convert to length in shorts */

for (i=0; i<len; ++i)
    tcksum += *sptr++;
tcksum = (tcksum >> 16) + (tcksum & 0xffff);
tcksum += (tcksum >> 16);

return (short)(~tcksum & 0xffff);
}
```

11.11.3 Finding The TCB For A Segment

Procedure tcpdemux finds the correct TCB for an incoming segment. The code searches array tcbtab sequentially. For TCBs that correspond to established connections, tcpdemux makes four comparisons to check both connection endpoints. In addition to comparing the source and destination protocol port numbers in the segment to those in the entry, it compares the source and destination IP addresses in the IP datagram to those in the entry. However, because servers do not specify a foreign IP address or protocol port number, tcpdemux cannot compare the source addresses on these entries. Thus, for TCBs in the LISTEN state, tcpdemux compares only the destination protocol port number.

If a connection exists for the incoming segment, tcpdemux returns a pointer to the entry for the segment after acquiring its mutual exclusion semaphore. If no connection exists, tcpdemux examines the segment type. For most segment types, tcpdemux returns an error code (0). However, if the incoming segment contains a synchronization (SYN) request and a server has issued a passive open, tcpdemux returns a pointer to the TCB entry for the server. Of course, if no server has created a TCB for the specified destination address, tcpdemux returns an error for the SYN request.

To make searching efficient, tcpdemux searches the set of possible connections once. During the search, it looks for an exact match (i.e., a connection for which both endpoints in the TCB match both endpoints in the incoming segment) and also keeps a record of partial matches (server connections for which the destination matches). After



completing the search, it tests to see if the segment consisted of a SYN request. If so, it returns any partial match that may have been found.

```
/* tcpdemux.c - tcpdemux */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 * tcpdemux - do TCP port demultiplexing
 *-----
 */

struct tcb *tcpdemux(pep)
struct    ep    *pep;
{
    struct    ip    *pip = (struct ip *)pep->ep_data;
    struct    tcp   *ptcp = (struct tcp *)pip->ip_data;
    struct    tcb   *ptcb;
    int       tcbn, lstcgn;

    wait(tcps_tmutex);
    for (tcbn=0, lstcgn = -1; tcbn<Ntcp; ++tcbn) {
        if (tcbtab[tcbn].tcb_state == TCPS_FREE)
            continue;
        if (ptcp->tcp_dport == tcbtab[tcbn].tcb_lport &&
            ptcp->tcp_sport == tcbtab[tcbn].tcb_rport &&
            blkequ(pip->ip_src, tcbtab[tcbn].tcb_rip, IP_ALEN) &&
            blkque(pip->ip_dst, tcbtab[tcbn].tcb_lip, IP_ALEN)) {
            break;
        }
        if (tcbtab[tcbn].tcb_state == TCPS_LISTEN &&
            ptcp->tcp_dport == tcbtab[tcbn].tcb_lport)
            lstcgn = tcbn;
    }
    if (tcbn >= Ntcp)
        if (ptcp->tcp_code & TCPF_SYN)
            tcbn = lstcgn;
        else
            tcbn = -1;
    signal(tcps_tmutex);
    if (tcbn < 0)
```



```
        return 0;

    wait(tcbtab[tcbn].tcb_mutex);

    if (tcbtab[tcbn].tcb_state == TCPS_FREE)

        return 0;           /* OOPS! Lost it... */

    return &tcbtab[tcbn];

}
```

11.11.4 Checking Segment Validity

We saw that tcpinp calls function tcopok to check the validity of a segment before following transitions of the finite state machine. Tcpok compares the incoming segment to information in the TCB to see whether data in the segment lies in the receive window.

```
/* tcopok.c - tcopok */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcopok - determine if a received segment is acceptable
 */
Bool tcopok(ptcb, pep)
struct tcb *ptcb;
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct tcp *ptcp = (struct tcp *) pip->ip_data;
    int seglen, rwindow;
    tcpseq wlast, slast, sup;
    Bool rv;

    if (ptcb->tcb_state < TCPS_SYNRCVD)
        return TRUE;

    seglen = pip->ip_len - IP_HLEN(pip) - TCP_HLEN(ptcp);

    /* add SYN and FIN */
    if (ptcp->tcp_code & TCPF_SYN)
        ++seglen;
    if (ptcp->tcp_code & TCPF_FIN)
        ++seglen;
    rwindow = ptcb->tcb_rbsize - ptcb->tcb_rbcnt;
```



```
    if (rwindow == 0 && seglen == 0)
        return ptcp->tcp_seq == ptcb->tcb_rnext;
    wlast = ptcb->tcb_rnext + rwindow - 1;
    rv = (ptcp->tcp_seq - ptcb->tcb_rnext) >= 0 &&
        (ptcp->tcp_seq - wlast) <= 0;
    if (seglen == 0)
        return rv;
    slast = ptcp->tcp_seq + seglen - 1;
    rv |= (slast - ptcb->tcb_rnext) >= 0 && (slast - wlast) <= 0;

    /* If no window, strip data but keep ACK, RST and URG */
    if (rwindow == 0)
        pip->ip_len = IP_HLEN(pip) + TCP_HLEN(ptcp);
    return rv;
}
```

Tcpok allows all segments in the unsynchronized states (CLOSED, LISTEN, and SYN-SENT). For others, it compiles the segment length. Conceptually, SYN and FIN occupy one position in the sequence space, so tcpok adds one to the length if either the SYN or FIN bits are set. Once it has determined the segment length, tcpok computes the receiver window size (rwindow) and the highest possible sequence number that lies in the window (wlast). If data in the segment lies in the acceptable range (i.e., lies below or within the window), tcpok returns TRUE. Even if the window size is zero, some segment processing should still occur. Therefore, tcpok changes the IP header length when the window size is zero to make it appear that the segment arrived without data. Tcpinp has already verified the checksum, so it need not be recomputed.

11.11.5 Choosing A Procedure For the Current State

Once tcpinp has found a TCB for an incoming segment and verified that data in the segment is within the advertised window, it uses the current connection state (found in ptcb->tcb_state) to select a procedure to handle the segment. Array tcpswitch merely contains the addresses of procedures in one-to-one correspondence with states. As we will see, each procedure takes two arguments: a pointer to the TCB entry and a pointer to an incoming packet. File tcpswitch.c contains a declaration of the tcpswitch array.

```
/* tcpswitch.c */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```



```
char *tcperror[] = {
    "no error",
    "connection reset",           /* TCPE_RESET          */
    "connection refused",         /* TCPE_REFUSED        */
    "not enough buffer space",   /* TCPE_TOOBIG         */
    "connection timed out",       /* TCPE_TIMEDOUT      */
    "urgent data pending",        /* TCPE_URGENTMODE    */
    "end of urgent data",         /* TCPE_NORMALMODE    */
};

/* SEGMENT ARRIVES state processing */

int
tcpclosed(), tcplisten(), tcpsynsent(), tcpsynrcvd(),
tcpestablished, tcpfin1(), tcpfin2(), tcpclosewait(),
tcpclosing(), tcplastack(), tcptimewait();

int {*tcpswitch[NTCPSTATES]}() = {
    ioerr,                      /* TCPS_FREE           */
    tcpclosed,                   /* TCPS_CLOSED         */
    tcplisten,                  /* TCPS_LISTEN         */
    tcpsynsent,                 /* TCPS_SYNSENT        */
    tcpsynrcvd,                 /* TCPS_SYNRCVD       */
    tcpestablished,              /* TCPS_ESTABLISHED   */
    tcpfin1,                     /* TCPS_FINWAIT1      */
    tcpfin2,                     /* TCPS_FINWAIT2      */
    tcpclosewait,                /* TCPS_CLOSEWAIT     */
    tcplastack,                 /* TCPS_LASTACK       */
    tcpclosing,                  /* TCPS_CLOSING       */
    tcptimewait,                 /* TCPS_TIMEWAIT      */
};

/* Output event processing */

int tcpidle(), tcppersist(), tcpxmit(), tcprexmt();

int {*tcposwitch[NTCPOSTATES]}() = {
    tcpidle,                    /* TCPO_IDLE           */
    tcppersist,                 /* TCPO_PERSIST        */
    tcpxmit,                    /* TCPO_XMIT           */
    tcprexmt,                  /* TCPO_REXMT          */
};

}
```



11.12 Summary

TCP uses three separate processes to handle input, output, and timer functions. The processes coordinate through a data structure known as the transmission control block (TCB). TCP maintains a separate TCB for each active connection.

Our example implementation uses a procedure-driven implementation of the finite state machine in which one procedure corresponds to each state. This chapter showed how the TCP input process handles an incoming segment, using the connection endpoints to demultiplex it among, active TCBs and using a table to switch it to the appropriate state procedure.

11.13 FOR FURTHER STUDY

Postel [RFC 793] outlines the general idea underlying the TCB structure and describes many of the fields. The host requirements document [RFC 1122] contains further refinements. Many of the remaining fields in the TCB have been derived from RFCs discussed in the next chapters.

11.14 EXERCISES

1. Procedure tcballoc uses a sequential search to find a free TCB, which means the overhead of searching is proportional to the number of concurrent active TCP connections. Describe an implementation that can allocate a TCB in constant time.
2. Consider the order of fields in the example TCB. Have they been grouped according to Function? Explain.
3. The code declares array tcpswitch to be an array of pointers to functions that return integers. How could one implement tcpswitch in a language like Pascal that does not provide pointers to procedures? What are the advantages and disadvantages of each implementation?
4. The code declares some integer fields in the TCP header to be short and others to be long. Will this declaration work on all machines? Explain why or why not.
5. File tcpfsm.h defines 12 to be the maximum number of retransmissions TCP makes before giving up. Discuss whether this is a reasonable limit.
6. File tcpfsm.h defines two times the maximum segment lifetime to be two minutes. Can you imagine an internet where datagrams survive more than two minutes? Explain.



7. If applications use TCP to carry a sequence of 16-bit integers from one machine to another, will they have the same value when they arrive? Why or why not?
8. Array `tcperror` (declared in file `tcpswitch.c`) contains pointers to strings that give an explanation of each possible TCP error message. What is the advantage of collecting all the messages into one array?



12 TCP: Finite State Machine Implementation

12.1 Introduction

Chapter 11 discussed the general organization of TCP software in which individual procedures correspond to states of the TCP finite state machine. It showed how the TCP input process demultiplexes the incoming segment among TCBs, and how it uses the state variable from a TCB to select one of the procedures that correspond to machine states. This chapter examines each of the state procedures in detail.

12.2 CLOSED State Processing

The CLOSED state represents a TCB that has been allocated but not used in any way. In particular, the application program that allocated the TCB has neither completed an active open operation nor has it completed a passive open operation. As a result, any incoming segment generates a TCP RESET. Procedure `tcpclosed` implements the CLOSED state. It calls one of the output procedures, `tcpreset`, to generate and send the RESET message.

```
/* tcpclosed.c - tcpclosed */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpclosed - do CLOSED state processing
 */
int tcpclosed(ptcb, pep)
```



```
struct    tcb  *ptcb;
struct    ep   *pep;
{
    tcprreset(pep);
    return    SYSERR;
}
```

12.3 Graceful Shutdown

TCP uses a modified 3-way handshake to shut down connections. One side, call it A, initiates the shutdown by issuing a close operation. TCP on side A sends a FIN segment and moves to the FIN-WAIT-1 state. When it receives the FIN, the other side, call it B, sends an ACK, moves to the CLOSE-WAIT state, and waits for the application to close the connection. Back at side A, receipt of the ACK causes TCP to move to the FIN-WAIT-2 state.

When the application on side B executes a close operation, TCP sends a FIN and moves to the LAST-ACK state. Side A receives the FIN, moves to the TIME-WAIT state, sends the final ACK, and shuts down the connection. When the last ACK arrives on side B, that side shuts down as well. The next sections examine the procedures that handle graceful shutdown.

12.4 Timed Delay After Closing

Because the Internet Protocol is a best-effort delivery system, datagrams can be duplicated, delayed, or delivered out of order. Duplication and delay pose a potential problem for protocols like TCP that use IP for delivery because TCP allows applications to reuse protocol port numbers. In particular, a lost acknowledgement will cause a RESET, and will lead the sender to believe its last packet (including the FIN and data) was not delivered. Furthermore, if TCP allowed immediate reuse of port numbers after a connection terminated, a duplicated FIN request from the previous connection could cause termination of a later one that used the same ports.

To prevent duplicated segments from interfering with later connections, TCP does not delete a TCB immediately after a connection closes. Instead, it leaves the TCB in place for a short time. The standard specifies that TCP should wait twice the maximum segment lifetime^① before deleting the record of a connection. In our implementation, procedure tcpwait schedules the delayed deletion of a TCB. Unlike most procedures in this chapter, tcpwait does not correspond to an input state. Instead, other input state

^① The maximum segment lifetime is defined to be the maximum time a segment can survive in the underlying delivery system before it must be discarded.



procedures call it to schedule delayed deletion.

```
/* tcpwait.c - tcpwait */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpwait - (re)schedule a DELETE event for 2MSL from now
 */
int tcpwait(ptcb)
struct    tcb  *ptcb;
{
    int  tcbnum = ptcb - &tcbtab[0];

    tcpkilltimers(ptcb);
    tmset(tcps_oport, TCPQLEN, MKEVENT(DELETE, tcbnum), TCP_TWOMSL);
    return OK;
}
```

Tcpwait uses the timer process described in Chapter 14. It calls `tcpkilltimers` to delete any pending events associated with the TCB (e.g., retransmission events), and `tmset` to create a deletion event that will occur `TCP_TWOMSL` time units in the future. When the deletion event occurs, it causes the timer process to delete the TCB.

12.5 TIME-WAIT State Processing

TCP leaves a connection in the TIME-WAIT state after successful completion of graceful shutdown. Procedure `tcptimewait` implements TIME-WAIT state processing.

```
/* tcptimewait.c - tcptimewait */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcptimewait - do TIME_WAIT state input processing
 */

```



```
int tcptimewait(ptcb, pep)
{
    struct tcb *ptcb;
    struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct tcp *ptcp = (struct tcp *)pip->ip_data;

    if (ptcp->tcp_code & TCPF_RST)
        return tcbdealloc(ptcb);

    if (ptcp->tcp_code & TCPF_SYN) {
        tcpreset(pep);
        return tcbdealloc(ptcb);
    }

    tcpacked(ptcb, pep);
    tcpdata(ptcb, pep);           /* just ACK any packets */
    tcpwait(ptcb);
    return OK;
}
```

If a RESET arrives, the other side of the connection must have reinitialized, so tcptimewait deallocates the TCB. To prevent delayed SYN requests from causing a new connection, tcptimewait sends a RESET if a SYN segment arrives. Finally, if any other segment arrives, it could mean that an acknowledgment was lost, so TCP responds to the segment as usual. It calls repacked to handle acknowledgements, and tcpdata to process data in the segment. Finally, it calls tcpwait to remove the old deletion event and schedule a new one. The consequence of restarting the timer for each new segment can be surprising.

Because TCP restarts the TCB deletion timeout after each non-SYN segment, the TCB will not expire as long as the other side continues to send segments.

The advantage of leaving the TCB in place is that TCP will correctly handle delayed messages. The disadvantage is that an implementation that never stops sending segments can keep resources reserved in another machine indefinitely.

12.6 CLOSING State Processing

TCP reaches the CLOSING state after receiving a FIN in response to a FIN. Thus, both sides have agreed to shut down, and TCP has entered the CLOSING state to await



an acknowledgment of its FIN. Procedure `tcpclosing` implements the CLOSING state.

```
/* tcpclosing.c - tcpclosing */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpclosing - do CLOSING state input processing
 */
int tcpclosing(ptcb, pep)
struct tcb *ptcb;
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct tcp *ptcp = (struct tcp *)pip->ip_data;

    if (ptcp->tcp_code & TCPF_RST)
        return tcbdealloc(ptcb);
    if (ptcp->tcp_code & TCPF_SYN) {
        tcpreset(pep);
        return tcbdealloc(ptcb);
    }
    tcpacked(ptcb, pep);
    if ((ptcb->tcb_code & TCPF_FIN) == 0) {
        ptcb->tcb_state = TCPS_TIMEWAIT;
        signal(ptcb->tcb_ocsem); /* wake closer */
        tcpwait(ptcb);
    }
    return OK;
}
```

If a RESET arrives, `tcpclosing` deallocates the TCB. If a SYN request arrives, `tcpclosing` responds by sending a RESET and deallocating the TCB. For other segments, `tcpclosing` calls `tcpacked` to handle acknowledgements. Bit TCPF_FIN in the code field of the TCB records whether an acknowledgment has arrived for the FIN that was sent. When a segment arrives acknowledging the FIN, `tcpacked` clears the bit. `Tcpclosing` checks the bit and causes a transition to the TIME-WATT state if the bit has been cleared. When it makes the transition, `tcpclosing` calls `tcpwait` to erase any pending events and start the TCB deletion timer.



12.7 FIN-WAIT-2 State Processing

Usually, when one side sends a FIN, the other side acknowledges it immediately and delays before sending the second FIN. The state machine handles the delay with state FIN-WAIT-2, implemented by procedure tcpfin2.

```
/* tcpfin2.c - tcpfin2 */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 *  tcpfin2 - do FIN_WAIT_2 state input processing
 *-----
 */
int tcpfin2(ptcb, pep)
struct tcb *ptcb;
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct tcp *ptcp = (struct tcp *)pip->ip_data;

    if (ptcp->tcp_code & TCPF_RST)
        return tcpabort(ptcb, TCPE_RESET);
    if (ptcp->tcp_code & TCPF_SYN) {
        tcpreset(pep);
        return tcpabort(ptcb, TCPE_RESET);
    }
    if (tcpacked(ptcb, pep) == SYSERR)
        return OK;
    tcpdata(ptcb, pep); /* for data + FIN ACKing */

    if (ptcb->tcb_flags & TCBF_RDONE) {
        ptcb->tcb_state = TCPS_TIMEWAIT;
        tcpwait(ptcb);
    }
    return OK;
}
```

If a RESET arrives, tcpfin2 calls tcpabort to abort the connection and deallocate the TCB. It sends a RESET in response to an arriving SYN. If the other side sent data or a



FIN in the segment, tcpfin2 calls tcpacked to acknowledge the input and tcpdata to process it.

The finite state machine specifies that TCP should change to the TIME-WAIT state when a FIN arrives. However, it is important to understand that TCP does not follow such state transitions merely because a segment arrives with the FIN bit set. Instead, to accommodate datagrams that arrive out of order, it waits until the entire sequence of data has been received up to and including the FIN. That is,

Because TCP must handle out-of-order delivery, it does not make all state transitions instantly. In particular, it delays transitions that occur for a FIN segment until all data has been received and acknowledged.

In terms, of the implementation, if all the data plus a FIN arrives, the call to tcpdata sets bit TCBF_RDONE in the TCB. Thus, when checking to see whether it should move to the TIME-WAIT state, tcpfin2 checks the TCBF_RDONE bit in the TCB instead of the FIN bit in the segment. When making a transition to TIME-WAIT, tcpfin2 calls tcpwait to remove existing timer events and create a TCB deletion event,

12.8 FIN-WAIT-1 State Processing

TCP enters state FIN-WATT-1 when the user issuer a close operation, causing TCP to send a FIN. The other side can respond with an ACK of the FIN or with its own FIN or both. If a FIN arrives alone, the other side must have started to close the connection, so TCP responds with an ACK and moves to the CLOSING state. If an ACK arrives alone, TCP moves to the FIN-WAIT-2 state to await the FIN. Finally, if both a FIN and an ACK arrive, TCP moves to the TIME-WAIT state. Procedure tcpfin1 implements these transitions.

```
/* tcpfin1.c - tcpfin1 */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 *  tcpfin1 - do FIN_WAIT_1 state input processing
 *-----*
 */
int tcpfin1(ptcb, pep)
struct tcb *ptcb;
struct ep *pep;
```



```
{  
    struct ip *pip = (struct ip *)pep->ep_data;  
    struct tcp *ptcp = (struct tcp *)pip->ip_data;  
  
    if (ptcp->tcp_code & TCPF_RST)  
        return tcpabort(ptcb, TCPE_RESET);  
    if (ptcp->tcp_code & TCPF_SYN) {  
        tcpreset(pep);  
        return tcpabort(ptcb, TCPE_RESET);  
    }  
    if (tcpacked(ptcb, pep) == SYSERR)  
        return OK;  
    tcpdata(ptcb, pep);  
    tcpswindow(ptcb, pep);  
  
    if (ptcb->tcb_flags & TCBF_RDONE) {  
        if (ptcb->tcb_code & TCPF_FIN) /* FIN not ACKed */  
            ptcb->tcb_state = TCPS_CLOSING;  
        else {  
            ptcb->tcb_state = TCPS_TIMEWAIT;  
            signal(ptcb->tcb_ocsem); /* wake closer */  
            tcpwait(ptcb);  
        }  
    } else if ((ptcb->tcb_code & TCPF_FIN) == 0) {  
        signal(ptcb->tcb_ocsem); /* wake closer */  
        ptcb->tcb_state = TCPS_FINWAIT2;  
    }  
    return OK;  
}
```

As expected, `tcpfin1` aborts the connection immediately if a RESET arrives, and sends a RESET in response to a SYN. In general, TCP must still process incoming data, so it calls `tcpacked` to handle incoming acknowledgements, `tcpdata` to process data in the segment, and `tcpswindow` to adjust its sending window size. Once the input has been processed, `tcpfin1` checks to see if it should make a state transition. If the `TCBF_RDONE` bit is set, a FIN has arrived and so has all data in the sequence up to the FIN. If the `TCPF_FIN` bit is cleared, an ACK has arrived for the outgoing FIN. `Tcpfin1` uses these two bits to determine whether to make a transition to the CLOSING, FIN-WAIT-2, or TIME-WAIT states. When making the transition to TIME-WAIT, it must call `tcpwait` to schedule a TCB deletion event. When the outgoing FIN has been acknowledged, `tcpfin1` signals semaphore `tcb_ocsem`, allowing the application program



that has closed the connection to complete the close operation. If multiple application programs have access to the TCB, it will be deleted when the last one issues a close.

12.9 CLOSE-WAIT State Processing

The shutdown states we have seen so far handle transitions when an application program initiates shutdown with a close operation. By contrast, when a FIN arrives before the application issues a close, TCP enters the CLOSE-WAIT state. It uses end-of-file to inform the application program that the other side has shut down the connection, and waits for the application to issue a close operation before moving to the LAST-ACK state.

TCP uses procedure `tcpclosewait` to process incoming segments while it waits in the CLOSE-WAIT state.

```
/* tcpclosewait.c - tcpclosewait */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpclosewait - do CLOSE_WAIT state input processing
 */
int tcpclosewait(ptcb, pep)
struct tcb *ptcb;
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct tcp *ptcp = (struct tcp *)pip->ip_data;

    if (ptcp->tcp_code & TCPF_RST) {
        TcpEstabResets++;
        TcpCurrEstab--;
        return tcpabort(ptcb, TCPE_RESET);
    }
    if (ptcp->tcp_code & TCPF_SYN) {
        TcpEstabResets++;
        TcpCurrEstab--;
        tcpreset(pep);
        return tcpabort(ptcb, TCPE_RESET);
    }
}
```



```
tcpacked(ptcb, pep);
tcpwindow(ptcb, pep);
return OK;
}
```

If a RESET arrives, `tcpclosewait` calls `tcpabort` to abort the connection and remove the TCB. If a SYN arrives, `tcpclosewait` generates a RESET and aborts the connection. Finally, it calls `tcpacked` to handle acknowledgements and `tcpwindow` to update the sending window size.

12.10 LAST-ACK State Processing

The transition from CLOSE-WAIT to LAST-ACK occurs when an application issues a close operation. During the transition, TCP schedules a FIN to be sent and enters the LAST-ACK state to await acknowledgement. The FIN will be sent after remaining data, which may be delayed if the receiver has closed its window. Once it sends the FIN, TCP schedules retransmission. If an acknowledgement does not arrive within the normal retransmission timeout, TCP will retransmit the FIN. Procedure `tcplastack` implements LAST-ACK state processing.

```
/* tcplastack.c - tcplastack */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcplastack - do LAST_ACK state input processing
 */
int tcplastack(ptcb, pep)
struct tcb *ptcb;
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct tcp *ptcp = (struct tcp *)pip->ip_data;

    if (ptcp->tcp_code & TCPF_RST)
        return tcpabort(ptcb, TCPE_RESET);
    if (ptcp->tcp_code & TCPF_SYN) {
        tcpreset(pep);
```



```
        return tcpabort(ptcb, TCPE_RESET);
    }

    tcpacked(ptcb, pep);

    if ((ptcb->tcb_code & TCPF_FIN) == 0)
        signal(ptcb->tcb_ocsem); /* close() deallocs */
    return OK;
}
```

If a RESET arrives, tcplastack calls `tcpabort` to abort the connection. If a SYN arrives, it sends a RESET and then aborts the connection. For other cases, tcplastack calls `tcpacked` to handle incoming acknowledgements, and signals the open/close semaphore to allow applications to complete their close operations once the outgoing FIN has been acknowledged.

12.11 ESTABLISHED State Processing

Once a connection has been established, both sides remain in the ESTABLISHED state while they exchange data and acknowledgement. TCP calls procedure `tcpestablished` to handle any segment that arrives while in the ESTABLISHED state.

```
/* tcpestablished.c - tcpestablished */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpestablished - do ESTABLISHED state input processing
 */
int tcpestablished(ptcb, pep)
struct tcb *ptcb;
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct tcp *ptcp = (struct tcp *)pip->ip_data;

    if (ptcp->tcp_code & TCPF_RST) {
        TcpEstabResets++;
        TcpCurrEstab--;
        return tcpabort(ptcb, TCPE_RESET);
    }
}
```



```
    }

    if (ptcp->tcp_code & TCPF_SYN) {
        TcpEstabResets++;
        TcpCurrEstab--;
        tcpriset(pep);
        return tcpabort(ptcb, TCPE_RESET);
    }

    if (tcpacked(ptcb, pep) == SYSERR)
        return OK;
    tcpdata(ptcb, pep);
    tcpswindow(ptcb, pep);
    if (ptcb->tcb_flags & TCBF_RDONE)
        ptcb->tcb_state = TCPS_CLOSEWAIT;
    return OK;
}
```

If a RESET arrives, it means the other endpoint must have restarted and has no knowledge of the connection. Therefore, tcpestablished calls tcpabort to abort the connection immediately. If a SYN segment arrives, tcpestablished sends a RESET and aborts the connection. Otherwise, it calls tcpacked to handle incoming acknowledgements, tcpdata to check the FIN bit and extract data from the segment, and tcpswindow to update the sending window size if the segment contains a new window advertisement. If a FIN has arrived and all data up through the FIN has been received, the call to tcpdata will set bit TCBF_RDONE of the TCB flags field. Tcpestablished uses this bit to determine whether it should move to the CLOSE-WATT state.

12.12 Processing Urgent Data In A Segment

In the ESTABLISHED state, TCP must accept data from incoming segments, use it to fill in the receive buffer, compute a new window size, and send an acknowledgement. Procedure tcpdata handles the details of receiving data.

```
/* tcpdata.c - tcpdata */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 *  tcpdata - process an input segment's data section
 *-----
```



```
*/  
  
int tcpdata(ptcb, pep)  
struct    tcb  *ptcb;  
struct    ep   *pep;  
{  
    struct    ip   *pip = (struct ip *)pep->ep_data;  
    struct    tcp  *ptcp = (struct tcp *)pip->ip_data;  
    tcpseq      first, last, wlast;  
    int        datalen, rwindow, i, pp, pb;  
  
    if (ptcp->tcp_code & TCPF_URG) {  
        int rup = ptcp->tcp_seq + ptcp->tcp_urgptr;  
#ifdef BSDURG  
        rup--;  
#endif  
        if (!(ptcb->tcb_flags & TCBF_RUPOK) ||  
            SEQCMP(rup, ptcbrupseq) > 0) {  
            ptcbrupseq = rup;  
            ptcbrflags |= TCBF_RUPOK;  
        }  
    }  
    if (ptcp->tcp_code & TCPF_SYN) {  
        ptcbrnext++;  
        ptcbrflags |= TCBF_NEEDOUT;  
        ++ptcp->tcp_seq; /* so we start with data */  
    }  
    datalen = pip->ip_len - IP_HLEN(pip) - TCP_HLEN(ptcp);  
    rwindow = ptcbrbsize - ptcbrbcount;  
    wlast = ptcbrnext + rwindow-1;  
    first = ptcp->tcp_seq;  
    last = first + datalen - 1;  
    if (SEQCMP(ptcbrnext, first) > 0) {  
        datalen -= ptcbrnext - first;  
        first = ptcbrnext;  
    }  
    if (SEQCMP(last, wlast) > 0) {  
        datalen -= last - wlast;  
        ptcp->tcp_code &= ~TCPF_FIN; /* cutting it off */  
    }  
    pb = ptcbrbstart + ptcbrbcount; /* == rnext, in buf */  
    pb += first - ptcbrnext; /* distance in buf */
```



```
pb %= ptcb->tcb_rbsize;           /* may wrap          */
pp = first - ptcp->tcp_seq;        /* distance in packet */
for (i=0; i<datalen; ++i) {
    ptcb->tcb_rcvbuf[pb] = ptcp->tcp_data[pp++];
    if (++pb >= ptcb->tcb_rbsize)
        pb = 0;
}
tcpdodat(ptcb, ptcp, first, datalen); /* deal with it      */
if (ptcb->tcb_flags & TCBF_NEEDOUT)
    tcpkick(ptcb);
return OK;
}
```

Tcpdata begins by checking for urgent data. It uses mask TCPF_URG to examine the urgent bit of the code field. If the bit is set, the segment contains a valid urgent pointer in field `tcp_urgrptr`. Tcpdata extracts the urgent pointer and computes the location in the sequence to which it refers. Although the standard specifies that the urgent pointer gives the location of the end of urgent data, implementations derived from BSD UNIX interpret the pointer as giving a location one beyond the urgent data. To insure compatibility with such implementations, our code includes a configuration constant, `BSDURG`. When using the BSD interpretation of the urgent pointer, `tcpdata` decrements the value of the receive urgent pointer, `rup`, by one.

After it computes a sequence value for urgent data, `tcpdata` records the information in the TCB. It uses the mask `TCBF_RUPOK` to see if the receive urgent pointer has already been set for the TCB. The standard specifies that if multiple segments arrive carrying urgent data, the application must receive all urgent data immediately. Therefore, if the TCB has no outstanding urgent data pending or the new urgent pointer specifies a larger sequence than the existing one, `tcpdata` records the sequence space value of the urgent pointer that arrived and sets flag `TCBF_RUPOK` to indicate that the pointer is valid.

12.13 Processing Other Data In A Segment

After handling urgent data, `tcpdata` checks the SYN bit in the incoming segment. Conceptually, a SYN occupies one position in the arriving data sequence, so if the segment contains a SYN, `tcpdata` adds one to the sequence number in the TCB.

To handle data in the incoming segment, `tcpdata` computes its length (`datalen`) as well as the space remaining in the buffer (`rwindow`). It then computer an index in the receive buffer where the data starts (`pb`) and an index in the data area of the segment (`pp`). It treats the receive buffer as a circular array and copies `datalen` octets from the



segment into the buffer, wrapping around if the buffer index exceeds the buffer size. After copying data into the receive buffer, tcpdata calls procedure tcpdodat to finish processing, and procedure tcpkick to start output if output is needed (e.g., to return an acknowledgement).

Procedure tcpdodat handles several details.

```
/* tcpdodat.c - tcpdodat */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpdodat - do input data processing
 */
int tcpdodat(ptcb, ptcp, first, datalen)
struct    tcb  *ptcb;
struct    tcp  *ptcp;
tcpseq     first;
int       datalen;
{
    int   wakeup = 0;

    if (ptcb->tcb_rnext == first) {
        if (datalen > 0) {
            tfcoalesce(ptcb, datalen, ptcp);
            ptcb->tcb_flags |= TCBF_NEEDOUT;
            wakeup++;
        }
        if (ptcp->tcp_code & TCPF_FIN) {
            ptcb->tcb_flags |= TCBF_RDONE|TCBF_NEEDOUT;
            ptcb->tcb_rnext++;
            wakeup++;
        }
        if (ptcp->tcp_code & (TCPF_PSH | TCPF_URG)) {
            ptcb->tcb_flags |= TCBF_PUSH;
            wakeup++;
        }
        if (wakeup)
            tcpwakeups(READERS, ptcb);
    } else {
```



```
/* process delayed controls */
if (ptcp->tcp_code & TCPF_FIN)
    ptcb->tcb_finseq = ptcp->tcp_seq + datalen;
if (ptcp->tcp_code & (TCPF_PSH | TCPF_URG))
    ptcb->tcb_pushseq = ptcp->tcp_seq + datalen;
ptcp->tcp_code &= ~(TCPF_FIN|TCPF_PSH);
tfinsert(ptcb, first, datalen);
}
return OK;
}
```

For the case where data in the incoming segment extends the sequence of contiguous data that has been received successfully, tcpdodat must process control flags immediately. First, it sets the output flags field in the TCB (tcb_flags) so an acknowledgement will be generated. It calls procedure tfcoalesce to determine whether the data fills in holes in the sequence space that were formed when segments arrived out of order. Second, if the segment contains a FIN in addition to data, tcpdodat counts the FIN as an item in the sequence space, and sets the output flags to show that a FIN has arrived and an ACK is needed. Third, if the incoming segment has the push bit set, tcpdodat sets a flag to show that push has been requested. In any case, if tcpdodat determines that new data is available, it calls tcpwakeup to awaken any application processes that may be blocked awaiting data arrival.

If a segment arrives out of order, tcpdodat must handle delayed controls. For example, it could happen that a segment carrying a FIN arrives before the last segment carrying data. In such cases, TCP has stored information about the FIN in the TCB, so if the missing data arrives, tcpdodat can perform processing that was delayed. Tcpdodat calls tfinsert to record the octets received.

12.14 Keeping Track Of Received Octets

Recall that TCP must accommodate out-of-order delivery. Because the window advertisement limits incoming data to the buffer that has been allocated, TCP can always copy the arriving data directly into the buffer. However, TCP must also keep a record of which octets from the sequence have been received. To do so, it maintains a list of (sequence, length) pairs received for each active TCB.

Borrowing terminology used by IP, our example implementation calls items on the list fragments. Each item on the TCP fragment list represents a single segment of data that has been received. The entry contains the sequence number of the first octet and a length as defined by structure tcpfrag in file tcb.h. Whenever data arrives out of order, tcpdodat calls procedure tfinsert to insert an entry on the fragment list.



```
/* tfinsert.c - tfinsert */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>
#include <q.h>

/*
 * tfinsert - add a new TCP segment fragment to a TCB sequence queue
 */
int tfinsert(struct tcb *ptcb, tcpseq seq, unsigned datalen)
{
    struct tcb *ptcb;
    tcpseq seq;
    int datalen;
    {
        struct tcpfrag *tf;

        if (datalen == 0)
            return OK;
        tf = (struct tcpfrag *)getmem(sizeof(struct tcpfrag));
        tf->tf_seq = seq;
        tf->tf_len = datalen;
        if (ptcb->tcb_rsegq < 0)
            ptcb->tcb_rsegq = newq(NTCPFRAG, QF_WAIT);
        if (enq(ptcb->tcb_rsegq, tf, -tf->tf_seq) < 0)
            freemem(tf, sizeof(struct tcpfrag));
        return OK;
    }
}
```

Tfinsert takes four arguments that describe the sequence of data that has arrived. It allocates a new node for the data and links the node into the fragment list. The arguments consist of a pointer to a TCB, a starting sequence number, data length, and a Boolean that specifies whether a FIN has arrived. When data first arrives out of order for a connection, no queue exists, so tfinsert calls newq to create one. In any case, it calls enq to enqueue the new node.

In addition to adding entries that record the sequence numbers of data received, TCP must advance the counter that tells how many contiguous octets of the sequence space have been received successfully. In essence, it moves a pointer along the sequence



space until it finds the next "hole." Procedure tfcoalesce implements the operation.

```
/* tfcoalesce.c - tfcoalesce */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 * tfcoalesce - join TCP fragments
 *-----
 */

int tfcoalesce(ptcb, datalen, ptcp)
struct tcb *ptcb;
int datalen;
struct tcp *ptcp;
{
    struct tcpfrag *tf;
    int new;

    ptcb->tcb_rnext += datalen;
    ptcb->tcb_rbcnt += datalen;
    if (ptcb->tcb_rnext == ptcb->tcb_finseq)
        goto alldone;
    if ((ptcb->tcb_rnext - ptcb->tcb_pushseq) >= 0) {
        ptcp->tcp_code |= TCPF_PSH;
        ptcb->tcb_pushseq = 0;
    }
    if (ptcb->tcb_rsegq < 0) /* see if this closed a hole */
        return OK;
    tf = (struct tcpfrag *)deq(ptcb->tcb_rsegq);
    while ((tf->tf_seq - ptcb->tcb_rnext) <= 0) {
        new = tf->tf_len - (ptcb->tcb_rnext - tf->tf_seq);
        if (new > 0) {
            ptcb->tcb_rnext += new;
            ptcb->tcb_rbcnt += new;
        }
        if (ptcb->tcb_rnext == ptcb->tcb_finseq)
            goto alldone;
        if ((ptcb->tcb_rnext - ptcb->tcb_pushseq) >= 0) {
            ptcp->tcp_code |= TCPF_PSH;
            ptcb->tcb_pushseq = 0;
        }
    }
}
```



```
        }

        freemem(tf, sizeof(struct tcpfrag));
        tf = (struct tcpfrag *)deq(ptcb->tcb_rsegq);
        if (tf == 0) {
            freeq(ptcb->tcb_rsegq);
            ptcb->tcb_rsegq = EMPTY;
            return OK;
        }
    }

    enq(ptcb->tcb_rsegq, tf, -tf->tf_seq); /* got one too many */
    return OK;

alldone:
do
    freemem(tf, sizeof(struct tcpfrag));
    while (tf = (struct tcpfrag *)deq(ptcb->tcb_rsegq))
        freemem(tf, sizeof(struct tcpfrag));
    freeq(ptcb->tcb_rsegq);
    ptcb->tcb_rsegq = EMPTY;
    ptcp->tcp_code |= TCPF_FIN;
    return OK;
}
```

The central loop in `tfcoalesce` iterates through the entire TCP fragment list. It removes the first entry before starting, and then removes another entry each time the loop iterates. On each iteration, `tfcoalesce` checks to see if the entry it removed extends the currently received sequence space. The test is straightforward: a new entry only extends the sequence space if its starting sequence lies within or exactly adjacent to the existing sequence (field `tcb_rnext`).

During the iteration, if field `tcb_rnext` reaches the sequence number of the FIN, `tfcoalesce` declares that input is complete and branches to label `alldone` to remove the list. If the loop completes without exhausting the list, `tfcoalesce` must reinsert the last unlinked entry back into the list.

12.15 Aborting A TCP Connection

We have seen that several of the state procedures need to abort a TCP connection immediately. To do so, they call procedure `tcpabort`, passing as an argument a pointer to the TCB that must be deallocated, as well as an integer that encodes the cause of the abort.

```
/* tcpabort.c - tcpabort */
```



```
#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *  tcpabort - abort an active TCP connection
 */
int tcpabort(ptcb, error)
struct    tcb  *ptcb;
int       error;
{
    tcpkilltimers(ptcb);
    ptcb->tcb_flags |= TCBF_RDONE|TCBF_SDONE;
    ptcb->tcb_error = error;
    tcpwakeup(READERS|WRITERS, ptcb);
    return OK;
}
```

Tcpabort uses tcpkilltimers to delete all pending events for the connection. In addition, it sets bits in the flags field to show that both reception and transmission have completed. It stores the argument error in the TCB to show what kind of error caused the problem. Finally, tcpabort calls tcpwakeup to awaken any readers or writers that may be blocked awaiting I/O. Each of the application programs waiting to read or write will awaken and find the error type stored in the TCB.

12.16 Establishing A TCP Connection

Recall that TCP uses a 3-way handshake to establish a connection. A server issues a passive open and waits in the LISTEN state, while a client issues an active open and enters the SYN-SENT state. The server moves to the SYN-RECEIVED state. Eventually, both client and server enter the ESTABLISHED state. The next sections present the procedures associated with the states used to establish a connection.

12.17 Initializing A TCB

TCP calls procedure tcpsync to initialize a TCB whenever an application issues an active or passive open operation.

```
/* tcpsync.c - tcpsync */
```



```
#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>

/*
 * tcpsync - initialize TCB for a new connection request
 */
int tcpsync(ptcb)
struct    tcb  *ptcb;
{
    ptcb->tcb_state = TCPS_CLOSED;
    ptcb->tcb_type = TCPT_CONNECTION;

    ptcb->tcb_iss = ptcb->tcb_sunsa = ptcb->tcb_snext = tcpiss();
    ptcb->tcb_lwack = ptcb->tcb_iss;

    ptcb->tcb_sndbuf = (char *)getmem(TCPSBS);
    ptcb->tcb_sbsize = TCPSBS;
    ptcb->tcb_sbstart = ptcb->tcb_sbcount = 0;
    ptcb->tcb_ssema = screate(1);

    ptcb->tcb_rcvbuf = (char *)getmem(TCPRBS);
    ptcb->tcb_rbsize = TCPRBS;
    ptcb->tcb_rbstart = ptcb->tcb_rbcnt = 0;
    ptcb->tcb_rsegq = EMPTY;
    ptcb->tcb_rsema = screate(0);
    ptcb->tcb_ocsem = screate(0);

/* timer stuff */

    ptcb->tcb_srt = 0;      /* in sec/100 */
    ptcb->tcb_rtde = 0;      /* in sec/100 */
    ptcb->tcb_rexmt = 50;      /* in sec/100 */
    ptcb->tcb_rexmtcount = 0;
    ptcb->tcb_keep = 12000;      /* in sec/100 */

    ptcb->tcb_code = TCPF_SYN;
    ptcb->tcb_flags = 0;
```



```
    return OK;  
}
```

Tcpsync initializes the connection to the CLOSED state, allocates send and receive buffers, creates send and receive semaphores, and initializes miscellaneous counters and retransmission estimates. Finally, it sets the TCPF_SYN bit in the tcb_code field to indicate that a SYN should be sent.

12.18 SYN-SENT State Processing

Once TCP has sent a SYN request, it moves to the SYN-SENT state. Procedure tcpsynsent implements SYN-SENT state processing,

```
/* tcpsynsent.c - tcpsynsent */  
  
#include <conf.h>  
#include <kernel.h>  
#include <network.h>  
  
/*-----  
 * tcpsynsent - do SYN_SENT state processing  
 *-----  
 */  
  
int tcpsynsent(ptcb, pep)  
struct tcb *ptcb;  
struct ep *pep;  
{  
    struct ip *pip = (struct ip *)pep->ep_data;  
    struct tcp *ptcp = (struct tcp *)pip->ip_data;  
  
    if ((ptcp->tcp_code & TCPF_ACK) &&  
        ((ptcp->tcp_ack - ptcb->tcb_iss <= 0) ||  
         (ptcp->tcp_ack - ptcb->tcb_snext) > 0))  
        return tcpreset(pep);  
    if (ptcp->tcp_code & TCPF_RST) {  
        ptcb->tcb_state = TCPS_CLOSED;  
        ptcb->tcb_error = TCPE_RESET;  
        TcpAttemptFails++;  
        tcpkilltimers(ptcb);  
        signal(ptcb->tcb_ocsem);  
        return OK;
```



```
    }

    if ((ptcp->tcp_code & TCPF_SYN) == 0)
        return OK;

    ptcb->tcb_swindow = ptcp->tcp_window;
    ptcb->tcb_lwseq = ptcp->tcp_seq;
    ptcb->tcb_rnext = ptcp->tcp_seq;
    ptcb->tcb_cwin = ptcb->tcb_rnext + ptcb->tcb_rbsize;
    tcpacked(ptcb, pep);
    tcpdata(ptcb, pep);

    ptcp->tcp_code &= ~TCPF_FIN;
    if (ptcb->tcb_code & TCPF_SYN)           /* our SYN not ACKed */
        ptcb->tcb_state = TCPS_SYNRCVD;
    else {
        TcpCurrEstab++;
        ptcb->tcb_state = TCPS_ESTABLISHED;
        signal(ptcb->tcb_ocsem);      /* return in open */
    }
    return OK;
}
```

If an ACK arrives, tcpsynsent checks to insure the ACK specifies the correct sequence number and sends a RESET if it does not. If a RESET arrives, tcpsynsent moves to the CLOSED state and calls tcckilltimers to delete any pending events. If the incoming segment contains a SYN, it can also carry data or an acknowledgement for a SYN that was sent previously, so tcpsynsent calls repacked and tcpdata to process the segment. Finally, tcpsynsent examines the TCPF_SYN bit in the TCB to see if the SYN for this connection has been acknowledged. If the SYN has been acknowledged, tcpsynsent moves the connection to the ESTABLISHED state. Otherwise, it moves to the SYN-RECEIVED state.

12.19 SYN-RECEIVED State Processing

TCP places a connection in the SYN-RECEIVED state either when a SYN arrives from the other end to initiate a 3-way handshake, or when a SYN arrives without an ACK and the connection is in the SYN-SENT state. Procedure tcpsynrcvd handles incoming segments for the SYN-RECEIVED state.

```
/* tcpsynrcvd.c - tcpsynrcvd */
```

```
#include <conf.h>
#include <kernel.h>
```



```
#include <network.h>
#include <ports.h>

/*
 * tcpsynrcvd - do SYN_RCVD state input processing
 */
int tcpsynrcvd(ptcb, pep)
struct tcb *ptcb;
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct tcp *ptcp = (struct tcp *)pip->ip_data;
    struct tcb *pptcb;

    if (ptcp->tcp_code & TCPF_RST) {
        TcpAttemptFails++;
        if (ptcb->tcb_pptcb != 0)
            return tcbdealloc(ptcb);
        else
            return tcpabort(ptcb, TCPE_REFUSED);
    }
    if (ptcp->tcp_code & TCPF_SYN) {
        TcpAttemptFails++;
        tcpreset(pep);
        return tcpabort(ptcb, TCPE_RESET);
    }
    if (tcpacked(ptcb, pep) == SYSERR)
        return OK;
    if (ptcb->tcb_pptcb != 0) { /* from a passive open */
        pptcb = ptcb->tcb_pptcb;
        if (wait(pptcb->tcb_mutex) != OK) {
            TcpAttemptFails++;
            tcpreset(pep);
            return tcbdealloc(ptcb);
        }
        if (pptcb->tcb_state != TCPS_LISTEN) {
            TcpAttemptFails++;
            tcpreset(pep);
            signal(pptcb->tcb_mutex);
            return tcbdealloc(ptcb);
        }
    }
}
```



```
        }

        if (pcount(pptcb->tcb_listenq) >= pptcb->tcb_lqsize) {
            TcpAttemptFails++;
            signal(pptcb->tcb_mutex);
            return tcbdealloc(ptcb);
        }

        psend(pptcb->tcb_listenq, ptcb->tcb_dvnum);
        signal(pptcb->tcb_mutex);

    } else /* from an active open */
        signal(ptcb->tcb_ocsem);
    TcpCurrEstab++;
    ptcb->tcb_state = TCPS_ESTABLISHED;
    tcpdata(ptcb, pep);
    if (ptcb->tcb_flags & TCBF_RDONE)
        ptcb->tcb_state = TCPS_CLOSEWAIT;
    return OK;
}
```

If a RESET arrives, tcpsynrcvd aborts the connection and deallocates the TCB. For passively opened connections, the TCB is a separate copy of the parent TCB, so it merely calls tcpdealloc to remove the orphan TCB. For actively opened connections, however, tcpsynrcvd calls tcpabort, which records the error in the TCB. It also aborts the connection if a SYN arrives.

Because TCP only enters the SYN-RECEIVED state after responding to a SYN, any incoming segment other than RESET or SYN means the other side views the connection as established. Thus, when a segment arrives, tcpsynrcvd calls tcpacked to handle acknowledgements, moves to the ESTABLISHED state, and calls tcpdata to extract data from the segment.

Tcpsynrcvd also handles part of the transition between a server and the process that executes for a particular connection. As we will see, when a server issues a passive open, it creates a listen queue. The server then enters a Loop, extracting the next connection from the listen queue and creating a process to handle the connection. We can summarize:

Using a passive open, a server creates a queue of connections for a given TCP port. TCP allocates a new TCB for each new connection, places a connection identifier in the listen queue for the port, and awakens the server process so it can handle the new connection.

For passive opens, tcpsynrcvd enqueues tcb_dvnum, the descriptor the server will use



for the connection, on the listen queue for the server. To avoid blocking the input process, it uses pcount to insure that space remains on the queue, and psend to enqueue the connection identifier on the listen queue. For active connection tcpsynrcvd signals the open-close semaphore, allowing the active open to proceed. In either case, tcpsynrcvd moves the state of the connection to the ESTABLISHED state, and calls tcpdata to extract data from the segment, if it exists. Finally, it checks the tcb_flags field to see if a FIN has arrived (possibly out of sequence or possibly in the same segment that carried the SYN) and transfers to the CLOSE-WAIT state if it has.

12.20 LISTEN State Processing

The LISTEN state, used by servers to await connections from clients, is among the most complex because it creates a new TCB for each incoming connection. Procedure tcplisten provides the implementation.

```
/* tcplisten.c - tcplisten */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcplisten - do LISTEN state processing
 */
int tcplisten(ptcb, pep)
struct tcb *ptcb;
struct ep *pep;
{
    struct tcb *newptcb, *tcballoc();
    struct ip *pip = (struct ip *)pep->ep_data;
    struct tcp *ptcp = (struct tcp *)pip->ip_data;

    if (ptcp->tcp_code & TCPF_RST)
        return OK; /* "parent" TCB still in LISTEN */
    if ((ptcp->tcp_code & TCPF_ACK) ||
        (ptcp->tcp_code & TCPF_SYN) == 0)
        return tcpreset(pep);
    newptcb = tcballoc();
    if ((int)newptcb == SYSERR || tcpsync(newptcb) == SYSERR)
        return SYSERR;
    newptcb->tcb_state = TCPS_SYNRCVD;
```



```
newptcb->tcb_ostate = TCPO_IDLE;
newptcb->tcb_error = 0;
newptcb->tcb_pptcb = ptcb;           /* for ACCEPT */

blkcopy(newptcb->tcb_rip, pip->ip_src, IP_ALEN);
newptcb->tcb_rport = ptcp->tcp_sport;
blkcopy(newptcb->tcb_lip, pip->ip_dst, IP_ALEN);
newptcb->tcb_lport = ptcp->tcp_dport;

tcpwinit(ptcb, newptcb, pep);      /* initialize window data */

newptcb->tcb_finseq = newptcb->tcb_pushseq = 0;
newptcb->tcb_flags = TCBF_NEEDOUT;
TcpPassiveOpens++;
ptcp->tcp_code &= ~TCPF_FIN;      /* don't process FINs in LISTEN */
tcpdata(newptcb, pep);
signal(newptcb->tcb_mutex);
return OK;
}
```

Tcplisten begins as expected. It ignores RESET requests because no connection exists. It sends a RESET for any incoming segment other than a SYN segment.

Once it receives a SYN, tcplisten must call tcballoc to create a TCB for the new connection. We call the original TCB the parent and the new TCB the child. Tcplisten calls tcpsync to initialize fields in the child TCB and then places the child in the SYN-RECEIVED state. Meanwhile, the parent TCB remains in the LISTEN state. An important step in initializing the new TCB consists of copying the sender's IP address and protocol port number from the arriving segment into the newly created TCB. Afterward, tcplisten calls tcpwinit to finish initializing window information for the new TCB.

Tcplisten does not explicitly generate the SYN and ACK response for the new connection. Instead, it sets bit TCBF_NEEDOUT in the tcb_flags field to indicate that output is needed, and calls tcpdata to process any data from the segment and to start output. Finally, tcplisten signals the mutual exclusion semaphore for the new TCB, allowing other parts of TCP to begin using it (e.g., to accept input or generate acknowledgements).

12.21 Initializing Window Variables For A New TCB

Procedure tcpwinit initializes variables used to control window and segment sizes.



```
/* tcpwinit.c - tcpwinit */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 *  tcpwinit - initialize window data for a new connection
 *-----*
 */

int tcpwinit(ptcb, newptcb, pep)
struct    tcb  *ptcb;
struct    tcb  *newptcb;
struct    ep   *pep;
{
    struct    ip   *pip = (struct ip *)pep->ep_data;
    struct    tcp  *ptcp = (struct tcp *)pip->ip_data;
    struct    route     *prt, *rtget();
    Bool      local;
    int       mss;

    newptcb->tcb_swindow = ptcp->tcp_window;
    newptcb->tcb_lwseq = ptcp->tcp_seq;
    newptcb->tcb_lwack = newptcb->tcb_iss; /* set in tcpsync() */ */

    prt = (struct route *)rtget(pip->ip_src, RTF_REMOTE);
    local = prt && prt->rt_metric == 0;
    newptcb->tcb_pni = &nif[prt->rt_ifnum];
    rtfree(prt);
    if (local)
        mss = newptcb->tcb_pni->ni_mtu-IPMHLEN-TCPMHLEN;
    else
        mss = 536; /* RFC 1122 */
    if (ptcb->tcb_smss) {
        newptcb->tcb_smss = min(ptcb->tcb_smss, mss);
        ptcb->tcb_smss = 0; /* reset server smss */ }
    else
        newptcb->tcb_smss = mss;
    newptcb->tcb_rmss = mss; /* receive mss */ }
    newptcb->tcb_cwnd = newptcb->tcb_smss; /* 1 segment */ }
    newptcb->tcb_ssthresh = 65535; /* IP max window */ }
```



```
newptcb->tcb_rnext = ptcp->tcp_seq;
newptcb->tcb_cwin = newptcb->tcb_rnext + newptcb->tcb_rbsize;
}
```

Because Chapter 15 discusses congestion control and the window size limits used to implement it, most of the initialization cannot be understood before reading that chapter. However, the maximum segment size selection is interesting and can be understood easily.

The TCP standard specifies that TCP should use a default maximum segment size (MSS) of 536 octets^① when communicating with destinations that do not lie on a directly connected network. For destinations on directly connected networks, however, TCP can use the network MTU to compute an optimal MSS. To do so, tcpwinit calls rtget to find a route to the remote endpoint. If the route has a metric of zero, the destination lies on a directly connected network, so TCP computes the MSS by subtracting the TCP and IP header sizes from the network MTU. If TCP on the remote machine specifies an MSS, tcpwinit uses the smaller of the specified MSS and the MSS computed from the network MTU. In any case, tcpwinit uses the MSS computed from the network MTU for input.

12.22 Summary

We examined an implementation that uses a single procedure to represent each state of the TCP finite state machine. This chapter reviewed the eleven state procedures as well as the utility procedures they use. Each state procedure handles incoming segments. It must accommodate requests to abort (e.g., RESET) and special requests to start (SYN) and shutdown (FIN).

The requirement that TCP accept segments out of order complicates most state procedures. For example, TCP may receive a request for shutdown (FIN) before all data arrives. Or it may receive a segment carrying data before a segment that completes the 3-way handshake used to establish the connection. Our implementation accommodates out-of-order delivery by recording startup and shutdown events in the TCB and checking them as each segment arrives.

12.23 FOR FURTHER STUDY

The TCP standard [RFC 793] specifies the finite state machine and gives details

^① The maximum segment size is computed by subtracting the minimum size of an IP header (20 octets), and the minimum size of a TCP header (20 octets) from the default IP datagram size (576 octets).



about making transitions. The host requirements document [RFC 1122] discusses changes and clarifications,

12.24 EXERCISES

1. The state diagram in Figure 11.2 shows a direct transition from SYN-RECEIVED to FIN-WAIT-1 if the user issues a close. Explain why our implementation of TCP cannot make such a transition.
2. Suppose an underlying network exhibits extremely bad behavior and reorders datagrams completely. Can a FIN for a connection ever arrive before the SYN for that connection? Why or why not?
3. Read the protocol standard carefully. How should TCP respond if an acknowledgement arrives for data that has not been sent yet?
4. Describe what happens if TCP enters state FIN-WAIT-2 and then the remote site crashes. Hint: Are there any timers running locally? Explain how the problem can be resolved.



13 TCP: Output Processing

13.1 Introduction

Chapters 11 and 12 discussed the use of a finite state machine to control TCP input processing. Although the standard specifies finite state machine transitions used for input, an implementation is much more complex than the simple diagram implies. This chapter discusses the output side of TCP and shows how it also uses a finite state machine to control processing. It discusses output of data segments that originate when an application program on the local machine sends information, output of acknowledgements sent in response to arriving segments, and output triggered when retransmission timers expire.

13.2 Controlling TCP Output Complexity

TCP output is complex because it interacts closely with TCP input and timer events, all of which occur concurrently. For example, when the output process sends a segment, it must schedule a retransmission event. Later, if the retransmission timer expires, the timer process must send the segment. Meanwhile, the application program may generate new data, causing TCP to send more segments, or acknowledgements may arrive, causing TCP to cancel previous retransmission events. However, because the underlying IP protocol may drop, delay, or deliver segments out of order, events may not occur in the expected order. Even if data arrives at the remote site, an acknowledgement may be lost. Because the remote site may receive data out of order, a single ACK may acknowledge receipt of many segments. Furthermore, a site may receive the FIN for a connection before it has received all data segments, so retransmission may be necessary even after an application closes a connection. Thus, the correct response to an input or output event depends on the history of previous event and cannot easily be specified in isolation.

To help control the complexity of interactions among the TCP input, output, and timer processes, our implementation uses a simple finite state machine to control output



operations. Unlike the finite state machine used for input, the output state machine is not part of the TCP standard. Instead, it is part of our design, and other implementations may use slightly different strategies to control output. In general, all implementations need some technique to handle the details of output because the input state machine does not distinguish among output operations.

We think of the output state machine as defining microscopic transitions that occur within a single state of the input state machine.

Thus, once the input state machine reaches its ESTABLISHED state, the output state machine makes transitions that control transmission, retransmission, and idling when there is nothing to send.

13.3 The Four TCP Output States

In principle, the output state machine is simpler than the input machine. Conceptually, it contains only four possible states and the transitions among them are quite simple, as Figure 13.1 shows. For example, when an application program produces new data and needs TCP to form and send a segment, it places the data in a buffer, moves the output state to TRANSMIT, and signals the output process, allowing it to execute. The output process calls an appropriate procedure to generate and send a segment, and then moves the output state machine back to the IDLE state.

The state diagram only provides a model from which the designer builds software. As with the input side, exceptions and special cases complicate the implementation, and no simple state transition diagram can explain all the subtleties.

13.4 TCP Output As A Process

Using a separate TCP output process helps separate execution of the input, timer, and output functions, and allows them to operate concurrently. For example, a retransmission timer may expire and trigger retransmission, while the input process is sending an acknowledgement in response to an incoming segment. The interaction can be especially complex because a TCP segment can carry acknowledgements along with data. If each procedure that needs output acts independently, TCP generates unnecessary traffic. To coordinate output, our example implementation uses a single process to handle output, and makes all interaction message-driven. When a procedure needs to generate output it places information in the TCB and sends a message to the TCP output process. Thus, there is little interaction among processes generating output, and little need for mutual exclusion.

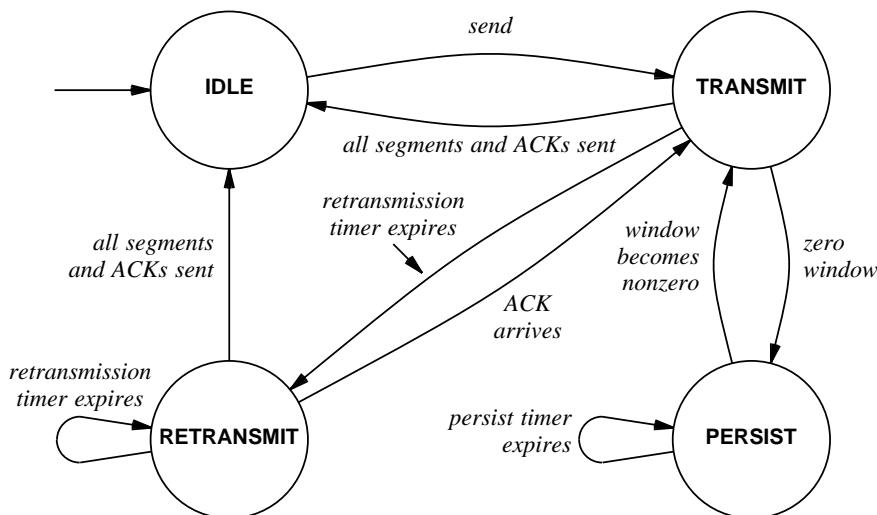


Figure 13.1 Conceptual transitions among the four TCP output states. Unlike the finite state machine used for input, the output state machine is not specified by the TCP protocol standard — it was defined for our implementation.

13.5 TCP Output Messages

Figure 13.2 lists the message types that can be sent to the TCP output process.

Number	Message	Meaning
1	SEND	Send data and/or ACK
2	PERSIST	Send probe to test receiver's zero window
3	RETRANSMIT	Retransmit data segment
4	DELETE	Delete a TCB that has expired

Figure 13.2 The four message types that can be sent to the TCP output process.

Although two of the message types have the same names as states in the finite state machine used for output, they should not be confused. A message specifies an action that is required, while a state specifies the current status of the connection. For example, a message that specifies RETRANSMIT may occur while the connection is in the TRANSMIT state.

13.6 Encoding Output States And TCB Numbers

Conceptually, whenever a process passes a message to the TCP output process it must send two items: the index of the TCB to which the message applies and a value that



identifies the message type. In our example code, the operating system only provides message passing facilities for passing a single integer value. To accommodate the restriction on message passing, our example TCP encodes both the TCB and message to be delivered into a single integer.

In addition to symbolic constants for the output states, file tcpfsm.h (shown in Chapter 11) contains declarations of three in-line macro functions used to encode and decode messages. Function MKEVENT takes a TCB number and message type (called a timer event), and encodes them in an integer by using the low-order 3 bits to represent the event, and the higher-order bits to store the TCB index. Function TCB takes an encoded integer value and extracts the TCB index; function EVENT takes an encoded integer value and extracts the event.

13.7 Implementation Of The TCP Output Process

Our implementation of the finite state machine used for output follows the pattern used for input. A single procedure handles each state; the output process uses the current output state, found in the TCB, to choose the appropriate procedure. The code can be found in procedure tcpout.

```
/* tcpout.c - tcpout */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *  tcpout - handle events affecting TCP output processing
 */
PROCESS tcpout()
{
    struct    tcb  *ptcb;
    int       i;

    tcps_oport = pcreate(TCPQLEN);
    signal(Net.sema);      /* synchronize on startup */

    while (TRUE) {
        i = preceive(tcps_oport);
        ptcb = &tcbtab[TCB(i)];
        if (ptcb->tcb_state <= TCPS_CLOSED)
            continue;      /* a rogue; ignore it */
    }
}
```



```
    wait(ptcb->tcb_mutex);

    if (ptcb->tcb_state <= TCPS_CLOSED)
        continue;      /* TCB deallocated */

    if (EVENT(i) == DELETE)      /* same for all states */
        tcbdealloc(ptcb);

    else
        tcposwitch[ptcb->tcb_ostate](TCB(i), EVENT(i));

    if (ptcb->tcb_state != TCPS_FREE)
        signal(ptcb->tcb_mutex);

}
}
```

Tcpout begins by calling pcreate to create a port on which messages can be queued. It records the port identifier in global variable tcps_oport, so other processes can know where to send messages. Tcpout then enters an infinite loop, waiting for the next message to arrive at the port, extracting the message, and handling it.

After receiving a message from the port, tcpout uses functions TCB and EVENT to decode the TCB number and event type. It then uses the output state variable from the TCB (tcb_ostate) and array tcposwitch to select the procedure for the current output.

Tcpout contains two optimizations. First, because state processing does not make sense if the TCB is closed, tcpout tests explicitly for a closed TCB. If it is closed, tcpout continues processing without calling any state procedures. Second, because all states deallocate the TCB in response to a DELETE event, tcpout tests for the DELETE event explicitly, and calls tcbdealloc directly whenever it arrives.

13.8 Mutual Exclusion

To guarantee that it has exclusive use of the TCB, tcpout waits on the mutual exclusion semaphore, tcb_mutex. Thus, each state procedure is called with exclusive access to the TCB. As a consequence, the state procedures should not wait on the mutual exclusion semaphore again, or deadlock will result:

The TCP output process obtains exclusive use of a TCB before calling a state procedure. The state procedure must not wait on the mutual exclusion semaphore again, or deadlock will result.

13.9 Implementation Of The IDLE State

Procedure tcpidle implements IDLE state processing, it is called whenever an event



occurs for an idle connection.

```
/* tcpidle.c - tcpidle */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 *  tcpidle - handle events while a connection is idle
 *-----*
 */

int tcpidle(tcbnum, event)
int tcbnum;
int event;
{
    if (event == SEND)
        tcpxmit(tcbnum, event);
    return OK;
}
```

Remember that tcpout explicitly tests for a DELETE event. Of the remaining events, PERSIST and RETRANSMIT cannot occur for an idle connection. Therefore, only SEND messages make sense in the IDLE state. Tcpidle calls tcpxmit to send data or an acknowledgement.

13.10 Implementation Of The PERSIST State

The PERSIST state handles events when the remote receiver has advertised a zero window. To avoid having a lost window update prevent TCP from ever sending, the protocol standard requires a sender to probe the receiver periodically by sending a segment. The receiver will return its latest window size in the ACK.

Procedure tcppersist implements the PERSIST state.

```
/* tcppersist.c - tcppersist */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 *  tcppersist - handle events while the send window is closed
 */
```



```
*-----  
*/  
  
int tcppersist(tcbnum, event)  
int tcbnum;  
int event;  
{  
    struct    tcb  *ptcb = &tcbtab[tcbnum];  
  
    if (event != PERSIST && event != SEND)  
        return OK;      /* ignore everything else */  
    tcpsend(tcbnum, TSF_REXMT);  
    ptcb->tcb_persist = min(ptcb->tcb_persist<<1, TCP_MAXPRS);  
    tmset(tcps_oport, TCPQLEN, MKEVENT(PERSIST, tcbnum),  
          ptcb->tcb_persist);  
    return OK;  
}
```

While in the PERSIST state, only the periodic events to test the window are allowed. Therefore, tcppersist discards all other events. It uses tcpsend to send a segment and calls tmset to reschedule another PERSIST event in the future.

13.11 Implementation Of The TRANSMIT State

Procedure tcpxmit handles the details of transmission.

```
/* tcpxmit.c - tcpxmit */  
  
#include <conf.h>  
#include <kernel.h>  
#include <network.h>  
  
/*-----  
 * tcpxmit - handle TCP output events while we are transmitting  
 *-----  
 */  
  
int tcpxmit(tcbnum, event)  
int tcbnum;  
int event;  
{  
    struct    tcb  *ptcb = &tcbtab[tcbnum];  
    int       tosend, tv, pending, window;
```



```
if (event == RETRANSMIT) {
    tmclear(tcbs_oport, MKEVENT(SEND, tcbnum));
    tcpxmit(tcbnum, event);
    ptcb->tcb_ostate = TCPO_REXMT;
    return OK;
} /* else SEND */
tosend = tcphowmuch(ptcb);
if (tosend == 0) {
    if (ptcb->tcb_flags & TCBF_NEEDOUT)
        tcpsend(tcbnum, TSF_NEWDATA); /* just an ACK */
    if (ptcb->tcb_snext == ptcb->tcb_suna)
        return OK;
    /* still unacked data; restart transmit timer */
    tv = MKEVENT(RETRANSMIT, tcbnum);
    if (!tmleft(tcbs_oport, tv))
        tmset(tcbs_oport, TCPQLEN, tv, ptcb->tcb_rexmt);
    return OK;
} else if (ptcb->tcb_swindow == 0) {
    ptcb->tcb_ostate = TCPO_PERSIST;
    ptcb->tcb_persist = ptcb->tcb_rexmt;
    tcpsend(tcbnum, TSF_NEWDATA);
    tmset(tcbs_oport, TCPQLEN, MKEVENT(PERSIST,tcbnum),
          ptcb->tcb_persist);
    return OK;
} /* else, we have data and window */
ptcb->tcb_ostate = TCPO_XMIT;
window = min(ptcb->tcb_swindow, ptcb->tcb_cwnd);
pending = ptcb->tcb_snext - ptcb->tcb_suna;
while (tcphowmuch(ptcb) > 0 && pending < window) {
    tcpsend(tcbnum, TSF_NEWDATA);
    pending = ptcb->tcb_snext - ptcb->tcb_suna;
}
tv = MKEVENT(RETRANSMIT, tcbnum);
if (!tmleft(tcbs_oport, tv))
    tmset(tcbs_oport, TCPQLEN, tv, ptcb->tcb_rexmt);
return OK;
}
```

If a retransmission event caused the call, tcpxmit moves the connection to the RETRANSMIT state and calls tcpxmit to send the segment.



For normal transmissions, `tcpxmit` checks several possibilities. First, it calls `tcpflowmuch` to compute the number of octets of data that should be sent. If no more data remains to be sent, `tcpxmit` checks the `TCBF_NEEDOUT` bit in field `tcb_flags` to see if output is needed (e.g., an acknowledgement, pushed data, or a window update), and calls `tcpsend` if output is pending.

If the output buffer contains data that is ready for transmission, `tcpxmit` checks the send window (`tcb_swindow`). If the receiver has specified a zero window size, `tcpxmit` moves to the PERSIST state, uses the current retransmission timer period (`tcb_rexmt`) as the persist period, and schedules a PERSIST event for that time interval.

Finally, `tcpxmit` handles the case where data is ready and the receiver has advertised a nonzero window. It moves to the TRANSMIT state, and uses `tcpflowmuch` to determine how much data can be sent. It repeatedly calls `tcpsend` to transmit a segment. To handle the case where one or more of the transmitted segments are lost, `tcpxmit` schedules a single retransmission event for the first segment in the window.

13.12 Implementation Of The RETRANSMIT State

Because the implementation of the RETRANSMIT state involves estimation of round-trip delays and backoff heuristics, the code appears in Chapter 15.

13.13 Sending A Segment

When `tcpxmit` needs to send a segment, it calls `tcpsend` to perform the task. `Tcpsend` allocates a buffer, assembles a segment, and sends it in an IP datagram.

```
/* tcpsend.c - tcpsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *  tcpsend - compute and send a TCP segment for the given TCB
 */
int
tcpsend(int tcbnum, Bool rexmt)
{
    struct    tcb  *ptcb = &tcbtab[tcbnum];
    struct    ep   *pep;
    struct    ip   *pip;
```



```
struct      tcp *ptcp;
u_char       *pch;
unsigned i, datalen, tocopy, off;
int         newdata;

pep = (struct ep *)getbuf(Net.netpool);
if ((int)pep == SYSERR)
    return SYSERR;
pep->ep_order = ~0;
pip = (struct ip *)pep->ep_data;
pip->ip_src = ptcb->tcb_lip;
pip->ip_dst = ptcb->tcb_rip;
datalen = tcpsndlen(ptcb, rexmt, &off); /* get length & offset */
ptcp = (struct tcp *)pip->ip_data;
ptcp->tcp_sport = ptcb->tcb_lport;
ptcp->tcp_dport = ptcb->tcb_rport;
if (!rexmt)
    ptcp->tcp_seq = ptcb->tcb_snext;
else
    ptcp->tcp_seq = ptcb->tcb_sunseq;
ptcp->tcp_ack = ptcb->tcb_rnext;

if ((ptcb->tcb_flags & TCBF_SNDFIN) &&
    SEQCMP(ptcp->tcp_seq+datalen, ptcb->tcb_slast) == 0)
    ptcb->tcb_code |= TCPF_FIN;
ptcp->tcp_code = ptcb->tcb_code;
ptcp->tcp_offset = TCPHOFFSET;
if ((ptcb->tcb_flags & TCBF_FIRSTSEND) == 0)
    ptcp->tcp_code |= TCPF_ACK;
if (ptcp->tcp_code & TCPF_SYN)
    tcprmss(ptcb, pip);
if (datalen > 0)
    ptcp->tcp_code |= TCPF_PSH;
ptcp->tcp_window = tcprwindow(ptcb);
if (ptcb->tcb_flags & TCBF_SUPOK) {
    short up = ptcb->tcb_supseq - ptcp->tcp_seq;

    if (up >= 0) {
#endif      BSDURG
        ptcp->tcp_urgptr = up + 1; /* 1 past end */
#else      /* BSDURG */

```



```
        ptcp->tcp_urgptr = up;
#endif /* BSDURG */
        ptcp->tcp_code |= TCPF_URG;
    } else
        ptcp->tcp_urgptr = 0;
} else
    ptcp->tcp_urgptr = 0;
pch = &pip->ip_data[TCP_HLEN(ptcp)];
i = (ptcb->tcb_sbstart+off) % ptcb->tcb_sbsize;
for (tocopy=datalen; tocopy > 0; --tocopy) {
    *pch++ = ptcb->tcb_sndbuf[i];
    if (++i >= ptcb->tcb_sbsize)
        i = 0;
}
ptcb->tcb_flags &= ~TCBF_NEEDOUT; /* we're doing it */
if (rexmt) {
    newdata = ptcb->tcb_suna + datalen - ptcb->tcb_snext;
    if (newdata < 0)
        newdata = 0;
    TcpRetransSegs++;
} else {
    newdata = datalen;
    if (ptcb->tcb_code & TCPF_SYN)
        newdata++; /* SYN is part of the sequence */
    if (ptcb->tcb_code & TCPF_FIN)
        newdata++; /* FIN is part of the sequence */
}
ptcb->tcb_snext += newdata;
if (newdata >= 0)
    TcpOutSegs++;
if (ptcb->tcb_state == TCPS_TIMEWAIT) /* final ACK */
    tcpwait(ptcb);
datalen += TCP_HLEN(ptcp);
tcpiph2net(ptcp);
pep->ep_order &= ~EPO_TCP;
ptcp->tcp_cksum = 0;
ptcp->tcp_cksum = tcpcksum(pep, datalen);
return ipsend(ptcb->tcb_rip, pep, datalen, IPT_TCP, IPP_NORMAL,
IP_TTL);
}
```



Although the idea behind sending a segment is straightforward, many details make the code complex. Conceptually, TCP maintains the sequence space as Figure 13.3 illustrates.

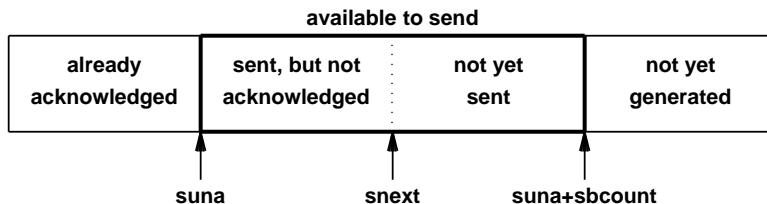


Figure 13.3 The conceptual sequence space and fields in the TCB that maintain pointers into it. Sequence numbers increase from left to right.

Because TCP uses a circular output buffer to hold the data, `tcpsend` must translate the sequence space computation into corresponding buffer addresses when it accesses data. Figure 13.4 explains how the available data maps into a circular buffer.

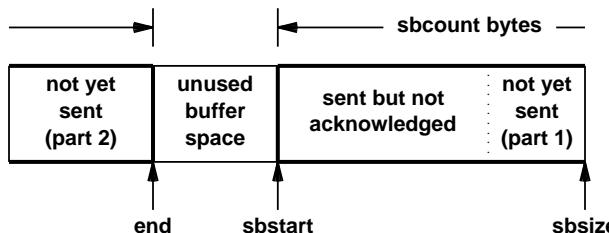


Figure 13.4 Available data wrapped around a circular TCP output buffer. Variable `sbcnt` tells the number of available bytes and `end` gives the location of the last byte of data. `End` can be computed as $(\text{sbstart} + \text{sbcnt}) \bmod \text{sbsize}$.

`TcpSend` begins by allocating a buffer that will hold an IP datagram as well as a complete TCP segment. It copies the local and remote IP addresses into the IP datagram, and calls `tcpSendLen` to compute the length of data to send as well as the sequence number of the first octet being sent. If the call occurred because retransmission is needed, `tcpSend` uses the sequence number of the first unacknowledged data octet in place of the computed value.

`TcpSend` places the sequence number of the next expected incoming octet (`tcb_rnext`) in the acknowledgement field of the segment. After filling in other values in the header, `tcpSend` calls `tcpRwindow` to compute a window advertisement and checks for urgent data. Finally, it copies data octets into the segment from the sending buffer, fills in the remaining header fields, calls `tcpH2Net` to convert integers to network byte order, calls `tcpChecksum` to compute the segment checksum, and passes the resulting IP datagram to `ipSend` for transmission.



13.14 Computing The TCP Data Length

Procedure `tcpsndlen` computes the amount of data to be sent.

```
/* tcpsndlen.c - tcpsndlen */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <q.h>

struct uqe *upalloc();

/*-----
 * tcpsndlen - compute the packet length and offset in sndbuf
 *-----
 */

int tcpsndlen(ptcb, rexmt, poff)
struct tcb *ptcb;
Bool rexmt;
int *poff;
{
    struct uqe *puqe, *puqe2;
    unsigned datalen;

    if (rexmt || (ptcb->tcb_code & TCPF_SYN))
        *poff = 0;
    else
        *poff = ptcb->tcb_snext - ptcb->tcb_sunap;
    datalen = ptcb->tcb_sbcount - *poff;
    datalen = min(datalen, ptcb->tcb_swindow);
    return min(datalen, ptcb->tcb_smss);
}
```

The normal case is straightforward. If the length is needed for retransmission of a segment or for the first segment in a stream, `tcpsndlen` starts the offset at zero (i.e., the offset from the first unacknowledged byte of data), and sets the data length equal to the count of octets in the output buffer. For other cases, `tcpsndlen` computes the offset of the first unsent byte of data, and computes the length of data to be sent by finding the difference between that sequence and the highest sequence of octets in the sending buffer. Of course, for non-retransmitted data, `tcpsndlen` must honor the receiver's advertised window, so if the window is smaller, it limits the data length.



13.15 Computing Sequence Counts

Procedure `tcpxmit` uses function `tcpflowmuch` to determine whether it needs to generate a segment. `Tcpflowmuch` determines how much data is waiting.

```
/* tcpflowmuch.c.c - tcpflowmuch */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpflowmuch.c - compute how much data is available to send
 */
int tcpflowmuch(ptcb)
struct    tcb  *ptcb;
{
    int  tosend;

    tosend = ptcb->tcb_sunap + ptcb->tcb_sbcount - ptcb->tcb_snext;
    if (ptcb->tcb_code & TCPF_SYN)
        ++tosend;
    if (ptcb->tcb_flags & TCBF_SNDFIN)
        ++tosend;
    return tosend;
}
```

13.16 Other TCP Procedures

13.16.1 Sending A Reset

The input procedures call `tcpreset` to generate and send a RESET segment whenever segments arrive unexpectedly (e.g., when no connection exists). The argument is a pointer to the input packet that caused the error.

```
/* tcprereset.c - tcprereset */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*

```



```
* tcpreset - generate a reset in response to a bad packet
*-----
*/
int tcpreset(pepin)
struct ep *pepin
{
    struct ep *pepout;
    struct ip *pipin = (struct ip *)pepin->ep_data, *pipout;
    struct tcp *ptcpin = (struct tcp *)pipin->ip_data, *ptcpout;
    int datalen;

    if (ptcpin->tcp_code & TCPF_RST)
        return OK;           /* no RESETs on RESETs */
    pepout = (struct ep *)getbuf(Net.netpool);
    if ((int)pepout == SYSERR)
        return SYSERR;
    pipout = (struct ip *)pepout->ep_data;
    blkcopy(pipout->ip_src, pipin->ip_dst, IP_ALEN);
    blkcopy(pipout->ip_dst, pipin->ip_src, IP_ALEN);

    ptcpout = (struct tcp *)pipout->ip_data;
    ptcpout->tcp_sport = ptcpin->tcp_dport;
    ptcpout->tcp_dport = ptcpin->tcp_sport;
    if (ptcpin->tcp_code & TCPF_ACK) {
        ptcpout->tcp_seq = ptcpin->tcp_ack;
        ptcpout->tcp_code = TCPF_RST;
    } else {
        ptcpout->tcp_seq = 0;
        ptcpout->tcp_code = TCPF_RST|TCPF_ACK;
    }
    datalen = pipin->ip_len - IP_HLEN(pipin) - TCP_HLEN(ptcpin);
    if (ptcpin->tcp_code & TCPF_SYN)
        datalen++;
    if (ptcpin->tcp_code & TCPF_FIN)
        datalen++;
    ptcpout->tcp_ack = ptcpin->tcp_seq + datalen;
    ptcpout->tcp_offset = TCPH_OFFSET;
    ptcpout->tcp_window = ptcpout->tcp_urp = 0;
    tcp2net(ptcpout);
    ptcpout->tcp_cksum = 0;
    ptcpout->tcp_cksum = tcpcksum(pepout);
```



```
    TcpOutSegs++;

    return ipsend(pipin->ip_src, pepout, TCPMHLEN, IPT_TCP,
                  IPP_NORMAL, IP_TTL);

}
```

Tcpreset rests the TCPF_RST bit in the segment that caused the problem to avoid generating RESET messages in response to RESET messages. It then proceeds to allocate a buffer that will hold an IP datagram and a RESET segment, and fills in the IP header.

When filling in the TCP header, tcpreset checks to see if the segment that caused the problem contained an ACK. If it does, tcpreset takes the sequence number for the RESET from the incoming acknowledgement field. Otherwise, it uses zero for the sequence number.

After filling in all the header fields, tcpreset calls tcph2net to convert integers in the TCP header to network byte order. It then calls tcpcksum to compute the checksum, and ipsend to send the resulting datagram.

13.16.2 Converting To Network Byte Order

Procedure tcph2net converts fields in the TCP header to network byte order. The code is straightforward.

```
/* tcph2net.c - tcph2net */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcph2net - convert TCP header fields from host to net byte order
 */
struct tcp *tcph2net(ptcp)
struct    tcp  *ptcp;
{
    /* NOTE: does not include TCP options */

    ptcp->tcp_sport = hs2net(ptcp->tcp_sport);
    ptcp->tcp_dport = hs2net(ptcp->tcp_dport);
    ptcp->tcp_seq = hl2net(ptcp->tcp_seq);
    ptcp->tcp_ack = hl2net(ptcp->tcp_ack);
    ptcp->tcp_window = hs2net(ptcp->tcp_window);
```



```
    ptcp->tcp_urgptr = hs2net(ptcp->tcp_urgptr);
    return ptcp;
}
```

13.16.3 Waiting For Space In The Output Buffer

Application programs that generate output may need to block if insufficient space remains in the buffer associated with a given TCB. To allocate space, they call procedure `tcpgetspace`.

```
/* tcpgetspace.c - tcpgetspace */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*-----
 *  tcpgetspace - wait for space in the send buffer
 *  N.B. - returns with tcb_mutex HELD
 *-----
 */
int tcpgetspace(ptcb, len)
struct tcb *ptcb;
int len;
{
    if (len > ptcb->tcb_sbsize)
        return TCPE_TOOBIG; /* we'll never have this much */
    while (1) {
        wait(ptcb->tcb_ssema);
        wait(ptcb->tcb_mutex);
        if (ptcb->tcb_state == TCPS_FREE)
            return SYSERR; /* gone */
        if (ptcb->tcb_error) {
            tcpwakeup(WRITERS, ptcb); /* propagate it */
            signal(ptcb->tcb_mutex);
            return ptcb->tcb_error;
        }
        if (len <= ptcb->tcb_sbsize - ptcb->tcb_sbcount)
            return len;
        signal(ptcb->tcb_mutex);
    }
}
```



If an application needs more space than the entire buffer can hold, `tcpgetspace` returns an error code. Otherwise, it signals the mutual exclusion semaphore and waits on the "send" semaphore again. `Tcpgetspace` tests field `tcb_error` to see if an error has occurred (e.g., a RESET caused TCP to abort the connection). If so, `tcpgetspace` calls `tcpwakeup` to awaken other processes that are waiting to write, signals the mutual exclusion semaphore, and returns the error to its caller.

If no error has occurred, `tcpgetspace` computes the available space by subtracting the count of used bytes from the buffer size. If the available space is sufficient to satisfy the request, `tcpgetspace` returns to its caller. Otherwise, it signals the mutual exclusion semaphore, and waits on the send semaphore again. Note that when `tcpgetspace` finds sufficient space, it returns to its caller with the mutual exclusion semaphore held. Thus, no other process can take space in the buffer until the caller uses the space it requested and signals the semaphore.

13.16.4 Awakening Processes Waiting For A TCB

Application programs block while waiting to transfer data through a TCP connection. If an abnormal condition causes TCP to break the connection, it must unblock all waiting processes before it can deallocate the TCB. Each process will resume execution, usually in a read or write procedure, which will find the error condition recorded in the TCB and report the error to its caller. Procedure `tcpwakeup` unblocks waiting processes. It takes two arguments: the first specifies the type of process that should be awakened (either READERS or WRITERS), and the second gives a pointer to the appropriate TCB.

```
/* tcpwakeup.c - tcpwakeup */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 *  tcpwakeup - wake up processes sleeping for TCP, if necessary
 *  NB: Called with tcb_mutex HELD
 *-----*
 */
int tcpwakeup(type, ptcb)
int      type;
struct   tcb  *ptcb;
{
```



```
int      freelen;
STATWORD ps;
disable(ps);
if (type & READERS) {
    if (((ptcb->tcb_flags & TCBF_RDONE) ||
        ptcb->tcb_rbcnt > 0 ||
        (ptcb->tcb_flags & TCBF_RUPOK)) &&
        scount(ptcb->tcb_rsema) <= 0)
        signal(ptcb->tcb_rsema);
}
if (type & WRITERS) {
    freelen = ptcb->tcb_sbsize - ptcb->tcb_sbcount;
    if (((ptcb->tcb_flags & TCBF_SDONE) || freelen > 0) &&
        scount(ptcb->tcb_ssema) <= 0)
        signal(ptcb->tcb_ssema);
    /* special for abort */
    if (ptcb->tcb_error && ptcb->tcb_ocsem > 0)
        signal(ptcb->tcb_ocsem);
}
restore(ps);
return OK;
}
```

Two semaphores control reading and writing. When no data remains for processes to read and no processes are waiting to read, the count of the reader's semaphore remains zero. Thus, any new process that attempts to read will be blocked. Tcpwakeups examines the input buffer, and if data is available, signals the reader's semaphore. If one or more processes remain blocked, one of them will proceed. If not, the call to signal will increment the semaphore count, which will allow the next process that issues a read to continue. Thus, the name wakeup is a slight misnomer because it might not awaken any processes when called.

Tcpwakeups examines the TCB to decide whether readers or writers should be allowed to proceed. If the remote side has sent all data, tcpwakeups will find bit TCBF_RDONE set. It also examines the count of bytes in the receive buffer to see if data has arrived, and bit TCBF_RUPOK to see if urgent data is present. In such cases, tcpwakeups checks to see if the semaphore currently allows access, and calls signal if it does not.

Tcpwakeups also participates in error propagation. If processes remain blocked waiting on the semaphore when an error occurs, the call to signal will allow the first process to proceed. When that process finds the error code, it signals the semaphore



again, allowing the next process to read. Each process executes, and then signals the semaphore to allow one more process to execute, until all waiting processes have resumed and found the error code.

Tcpwakeup behaves similarly when awakening processes waiting to write. It signals the writers' semaphore to unblock the first one, which will execute and unblock the next, and so on. Tcpwakeup also checks for the special case where the connection has been aborted and either an open or close is pending. In that case, tcpwakeup signals the open-close semaphore. The call to open or close returns an error code and deletes the TCB.

13.16.5 Choosing An Initial Sequence Number

To make TCP work in an environment where segments can be lost, duplicated, or delivered out of order, the code must choose a unique starting sequence number each time an application attempts to create a new connection. Procedure tcpiss generates an initial starting sequence by using the current time-of-day clock.

```
/* tcpiss.c - tcpiss */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#define    TCPINCR      904

/*-----
 *  tcpiss - set the ISS for a new connection
 *-----
 */
int tcpiss()
{
    static    int    seq = 0;
    extern    long   clktime;      /* the system ticker */

    if (seq == 0)
        seq = clktime;
    seq += TCPINCR;
    return seq;
}
```

Tcpiss maintains a static variable and uses the clock to initialize the variable only once. After initialization, tcpiss merely increments the starting sequence by a small



amount (904) for each new connection.

13.17 Summary

TCP output uses an extremely simple finite state machine that can be thought of as controlling macroscopic transitions within a single state of the input finite state machine. The output machine has four states that correspond to an idle connection, a connection on which data is being transmitted, a connection for which data is waiting but the receiver has closed its window, and a connection on which data is being retransmitted.

To help separate the interactions between input, output, and timer functions, our example implementation uses a separate process for each. All normal TCP output occurs from the output process, which performs only one operation at any time. Thus, there is never a problem controlling the concurrent interaction of transmission, retransmission, and acknowledgements. Other processes use message passing to inform the output process that output is needed; the output process uses a single message queue in which each message includes both a TCB number and a request for that TCB. Requests can specify transmission of data or acknowledgement, retransmission, deletion of the TCB, or the probe of a closed window.

The example implementation uses a separate procedure to implement each state of the output finite state machine. We saw that although the ideas are straightforward, details, exceptions, and special cases complicate the code.

13.18 FOR FURTHER STUDY

Pastel [RFC 793] specifies the TCP protocol, and [RFC 1122] contains further refinements.

13.19 EXERCISES

1. Explain what happens to the TCP checksum if routes change after a TCP connection has been opened. (Hint: consider the pseudo-header and the IP addresses used by `tcpsend`.)
2. Suppose a low priority process is waiting to write to a TCP connection, a medium priority process is executing, and a high priority process needs to create a new connection. Explain how `tcpwakeup` and the TCB deallocation scheme can allow the medium priority process to prevent the high priority process from forming a connection.
3. Ask David Stevens (`dls@cs.purdue.edu`) why the example code increments the



starting sequence number by 904.

4. Read the specification to find out how long TCP should persist in attempting to probe a closed window. How long does our example implementation persist?



14 TCP: Timer Management

14.1 Introduction

Real-time delay processing forms an essential part of TCP. In addition to the obvious need for timers that handle retransmissions, TCP uses timers for the 2 MSL delay following connection close, for probing after a receiver advertises a zero-size window, and, in some implementations, to delay acknowledgements.

This chapter considers an implementation of software that handles real-time delays. It shows how a single data structure can efficiently store a variety of delay requests, and how a single timer process can manage all the delays TCP requires. Earlier chapters have already shown how input and output software calls the timer routines to schedule delays and how the output process manages events when they occur. Later chapters will complete the investigation by showing how TCP estimates round trip delays and uses the round-trip estimates to compute retransmission delays.

14.2 A General Data Structure For Timed Events

The key to efficient management of timed events lies in a data structure known as a delta list. Each item on a delta list corresponds to an event scheduled to occur in the future. Because each scheduled event may occur at a different time, each item on a delta list has a field that gives the time at which the event should occur. To make updates efficient, a delta list stores events ordered by the time at which they will occur, and uses relative times, not absolute times. For example, Figure 14.1 shows a delta list that contains four items scheduled to occur 16, 20, 21, and 30 time units in the future. Items on the list have time values of 16, 4, 1, and 9 because the first item occurs 16 time units from the present, the second occurs 4 time units after the first, the third occurs 1 time unit after the second, and the fourth occurs 9 time units beyond the third.

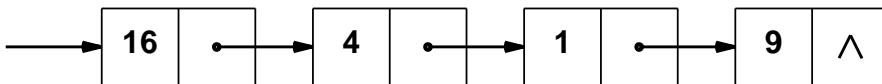


Figure 14.1 A delta list of events that occur 16, 20, 21, and 30 time units in the future. Stored values give times relative to the previous event.

The chief advantage of a delta list lies in its efficiency. Because all times are relatives, a periodic timer process only needs to decrement the time in the first item on the list. When the time in the first item reaches zero, the event occurs and the timer removes the item from the list. It then begins periodically decrementing the count in the next item on the list. The simplicity and efficiency will become clear as we consider software that manages a delta list.

14.3 A Data Structure For TCP Events

Items on the TCP delta list must contain more than a time field and a pointer to the next item. They must identify the action to be taken when the event expires. Structure tqent specifies the exact format of items on the TCP delta list. The declaration can be found in file tcptimer.h.

```
/* tcptimer.h */

/* A timer delta list entry */

struct tqent {
    int tq_timeleft;      /* time to expire (1/100 secs) */
    long tq_time;         /* time this entry was queued */
    int tq_port;          /* port to send the event */
    int tq_portlen;       /* length of "tq_port" */
    void *tq_msg;         /* data to send when expired */
    struct tqent *tq_next; /* next in the list */
};

/* timer process declarations and definitions */

extern PROCESS tcptimer();
#define TMSTK 512        /* stack size for fast timer */
#define TMPRI 100         /* timer process priority */
#define TMNAM "tcptimer"  /* name of fast timer process */
#define TMARGC 0           /* count of args to TCP timer */

extern long ctr100;        /* 1/100th of a second clock */
```



Field tq_next contains a pointer to the next item on the list. Field tq_timeleft specifies the time at which the item should occur. Values in tq_timeleft are relative times measured in hundredths of seconds, and follow the rule for delta lists:

Time in the first item on a delta list is measured relative to the current time, while time in other items is measured relative to the previous item on the list.

14.4 Timers, Events, And Messages

TCP timer management software can follow one of two basic designs: items on the timer delta list can store commands that the timer process interprets when the event occurs, or items on the list can store messages that the timer process delivers when the event occurs. The chief advantage of the former design lies in its ability to permit each timer event to trigger an arbitrarily complex operation. The chief advantage of the latter design lies in its simplicity. The timer process does not need to know the meaning of each message — it can take the same action whenever an event occurs.

To keep the timer process simple and efficient, we have chosen the latter design. Each event on the delta list includes a message (field tq_msg) and the identifier of a port to which the message should be sent when the event occurs (field tq_port). The timer mechanism sends the message in tq_msg to the port given by tq_port. To summarize:

The example TCP timer mechanism does not understand or interpret the messages stored in events. When the event occurs, the timer merely sends the specified message to the specified port.

14.5 The TCP Timer Process

When the system first starts, protocol initialization software creates a TCP timer process that executes procedure tcptimer.

```
/* tcptimer.c - tcptimer */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <tcptimer.h>

int tqmutex;
int tqpid;
```



```
struct tqent *tqhead;

/*-----
 * tcptimer - TCP timer process
 *-----
 */

PROCESS tcptimer()
{
    long now, lastrun;      /* times from system clock */
    int delta;              /* time since last iteration */
    struct tqent *tq;       /* temporary delta list ptr */

    lastrun = ctr100;        /* initialize to "now" */
    tqmutex = screate(1);    /* mutual exclusion semaphore */
    tqpid = getpid();        /* record timer process id */
    signal(Net.sema);        /* start other network processes */

    while (TRUE) {
        sleep10(TIMERGRAN);    /* real-time delay */
        if (tqhead == 0) /* block timer process if delta */
            suspend(tqid);   /* list is empty */
        wait(tqmutex);
        now = ctr100;
        delta = now - lastrun; /* compute elapsed time */

        /* Note: check for possible clock reset (time moved)
         * backward or delay was over an order of magnitude too */
        /* long) */

        if (delta < 0 || delta > TIMERGRAN*100)
            delta = TIMERGRAN*10; /* estimate the delay */
        lastrun = now;
        while (tqhead != 0 && tqhead->tq_timeleft <= delta) {
            delta -= tqhead->tq_timeleft;
            if (pcount(tqhead->tq_port) <= tqhead->tq_portlen)
                psend(tqhead->tq_port, tqhead->tq_msg);
            tq = tqhead;
            tqhead = tqhead->tq_next;
            freemem(tq, sizeof(struct tqent));
        }
    }
}
```



```
    if (tqhead)
        tqhead->tq_timeleft -=delta;
    signal(tqmutex);
}
}
```

The timer process begins by creating a mutual exclusion semaphore and storing its id in variable tqmutex. Tcptimer also stores its own process id in variable tqpid, and signals the network semaphore to allow packets to flow. Finally, tcptimer enters an infinite loop.

In each iteration of the main loop, tcptimer calls sleep10 to delay for TIMERGRAN tenths of seconds. It then checks the head of the delta list. If no item remains on the list, the timer calls suspend to block itself until some other process deposits an item. Although the call to suspend is not necessary, it eliminates having the timer process continue periodic execution when there is nothing for it to do.

As long as the delta list remains nonempty, tcptimer continues to iterate. On each iteration, it waits on the mutual exclusion semaphore (tqmutex) to obtain exclusive use of the delta list, processes items on the list, and then signals the mutual exclusion semaphore to allow other processes to access the list again. Note that the list is always available while tcptimer is blocked in the call to sleep10.

In the Xinu operating system, global variable ctr100 contains the value of the real-time clock expressed as hundredths of seconds past an epoch date. Tcptimer references variable ctr100 to obtain the current time, and uses variable lastrun to record the time of each iteration. Therefore, tcptimer can compute the elapsed time between iterations by subtracting the value of lastrun from the current time. The code checks to see if the system clock has been reset (e.g., time has moved backward or time has moved forward by more than ten times the expected delay). If it has, tcptimer substitutes a reasonable estimate for the delay and proceeds.

To process items on the delta list, tcptimer compares the time remaining for the item to the time that has expired between iterations. If the event should have occurred during the interval between the last iteration and the current iteration, tcptimer sends the message that the event contains (tq_msg) to the port that the message specifies (tq_port). It then removes the event from the delta list.

When removing an event, tcptimer updates the value of delta by decrementing the time for the event. Thus, like items on the list, delta always contains a relative time, making it possible to compare it directly to the time value stored in an individual item.

When tcptimer finishes removing items that have occurred, two possibilities exist: the list can be empty or nonempty. If the list is empty, no further processing is needed. However, if the list is nonempty, it must be true that the time remaining before the next



item should occur is greater than delta. In such cases, tcptimer reduces the time of the remaining item by delta before beginning the next cycle of delay.

14.6 Deleting A TCP Timer Event

TCP software may need to cancel an event before it expires. For example, when it receives an acknowledgement, TCP might cancel a retransmission event. To cancel an event, TCP needs to remove the corresponding item from the timer delta list. Procedure tmclear provides the necessary function.

```
/* tmclear.c - tmclear */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <tcptimer.h>

/*
 * tmclear - clear the indicated timer
 */
int tmclear(port, msg)
int port, msg;
{
    struct tqent     *prev, *ptq;
    int      timespent;

    wait(tqmutex);
    prev = 0;
    for (ptq = tqhead; ptq != NULL; ptq = ptq->tq_next) {
        if (ptq->tq_port == port && ptq->tq_msg == msg) {
            timespent = ctrl00 - ptq->tq_time;
            if (prev)
                prev->tq_next = ptq->tq_next;
            else
                tqhead = ptq->tq_next;
            if (ptq->tq_next)
                ptq->tq_next->tq_timeleft +=
                    ptq->tq_timeleft;
            signal(tqmutex);
            freemem(ptq, sizeof(struct tqent));
        }
    }
    return timespent;
}
```



```
        }
        prev = ptq;
    }
    signal(tqmutex);
    return SYSERR;
}
```

Tmclear takes a message (msg) and a port identifier (port) as arguments, and deletes a timer event with that message and port pair. The code is straightforward. Tmclear searches the delta list until it finds the item that matches the arguments. At each step, it keeps a pointer to an item on the list (ptq) and a pointer to the previous item (prev). When it finds a match, tmclear removes the item by unlinking it from the list and calling freemem to return the storage to the system's free memory pool.

Recall that times stored in items on the delta list are relative. Thus, whenever tmclear deletes an event it must be careful to adjust the time remaining for events that follow it. To make the adjustment, tmclear checks field tq_next to see if any items follow the one being deleted. If so, tmclear adds the delay for the deleted item to the delay for the one following.

14.7 Deleting All Events For A TCB

We saw in Chapters 11 and 12 that before TCP can remove a TCB, it must delete all timer events associated with that TCB. Procedure tcpkilltimers performs the task. Because our TCP software only allows three possible message types, and only arranges to send messages to the TCP output port, tcpkilltimers can use three calls to tmclear to delete all TCP events for a given TCB.

```
/* tcpkilltimers.c - tcpkilltimers */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpkilltimers - kill all outstanding timers for a TCB
 */
int tcpkilltimers(ptcb)
struct tcb *ptcb;
{
    int tcbnum = ptcb - &tcbtab[0];
```



```
/* clear all possible pending timers */

tmclear(tcps_oport, MKEVENT(SEND, tcbnum));
tmclear(tcps_oport, MKEVENT(RETRANSMIT, tcbnum));
tmclear(tcps_oport, MKEVENT(PERSIST, tcbnum));
return OK;
}
```

Most software that calls `tcpkilltimers` uses a pointer to refer to a TCB instead of the array index. To accommodate such software, argument `ptcb` is declared to be a pointer to the TCB. However, `tcpkilltimers` needs to use a TCB index number in the call to `MKEVENT`. To compute the index, `tcpkilltimers` uses pointer arithmetic, subtracting the address of the start of the TCB array (`tcbtab`) from the pointer to a given entry. Thus, variable `tcbnum` contains an integer index of the entry.

14.8 Determining The Time Remaining For An Event

Procedure `tmleft` determines the amount of time left before an event occurs. It returns zero if no such event exists.

```
/* tmleft.c - tmleft */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <tcoptimer.h>

/*
 * tmleft - how much time left for this timer?
 */
int tmleft(port, msg)
int port, msg;
{
    struct tqent *tq;
    int timeleft = 0;

    if (tqhead == NULL)
        return 0;
    wait(tqmutex);
```



```
for (tq = tqhead; tq != NULL; tq = tq->tq_next) {  
    timeleft += tq->tq_timeleft;  
    if (tq->tq_port == port && tq->tq_msg == msg) {  
        signal(tqmutex);  
        return timeleft;  
    }  
}  
signal(tqmutex);  
return 0;  
}
```

To determine the time remaining, tmleft must sum the relative times in all events up to and including the time on the specified event. It uses pointer tq to walk the linked list, starting at tqhead and fallowing pointer tq_next in each item. As it moves along the list, tmleft accumulates the time delay in local variable timeleft. When it reaches the item for which the time was requested, it signals the mutual exclusion semaphore and returns the computed total to the caller. If the specified event does not exist, tmleft returns zero.

14.9 Inserting A TCP Timer Event

TCP software calls procedure tmset to create an event and insert it on the TCP delta list. Tmset takes arguments that specify a desired delay (time), a message to be sent when the event occurs (msg}, the port to which the message should be sent (port) and the length of the port (portlen).

```
/* tmset.c - tmset */  
  
#include <conf.h>  
#include <kernel.h>  
#include <network.h>  
#include <tcptimer.h>  
  
/*-----  
 * tmset - set a fast timer  
 *-----  
 */  
int tmset(port, portlen, msg, time)  
int port, portlen, msg, time;  
{  
    struct tqent     *ptq, *newtq, *tq;
```



```
newtq = (struct tqent *)getmem(sizeof(struct tqent));
newtq->tq_timeleft = time;
newtq->tq_time = ctrl00;
newtq->tq_port = port;
newtq->tq_portlen = portlen;
newtq->tq_msg = msg;
newtq->tq_next = NULL;

/* clear duplicates */
(void) tmclear(port, msg);

wait(tqmutex);
if (tqhead == NULL) {
    tqhead = newtq;
    resume(tqid);
    signal(tqmutex);
    return OK;
}
/* search the list for our spot */

for (ptq=0, tq=tqhead; tq; tq=tq->tq_next) {
    if (newtq->tq_timeleft < tq->tq_timeleft)
        break;
    newtq->tq_timeleft -= tq->tq_timeleft;
    ptq = tq;
}
newtq->tq_next = tq;
if (ptq)
    ptq->tq_next = newtq;
else
    tqhead = newtq;
if (tq)
    tq->tq_timeleft -= newtq->tq_timeleft;
signal(tqmutex);
return OK;
}
```

Tmset calls getmem to allocate free memory for an event list item, and then fills in fields of the item from the arguments. It calls tmclear to remove the message from the list if it already exists. Finally, tmset waits on the mutual exclusion semaphore, inserts the new item in the list, and signals the mutual exclusion semaphore before returning.



Although the list insertion code in tmset is straightforward, a few details make it appear complicated. The timer process remains suspended as long as no events are pending. When tmset inserts an item into an empty list, it calls resume to restart the timer process. When it inserts into a nonempty list, tmset must search the list to find the correct insertion point.

During the search, tmset uses two variables that point to a node on the list (tq) and its predecessor (ptq). As it passes items on the list, tmset subtracts their delay from the delay for the new item to keep its delay relative to the current position in the list. When the while loop terminates, the new item belongs between the items to which ptq and tq point. Tmset links the new item into the list and decrements the time on the successor by the added delay,

14.10 Starting TCP Output Without Delay

The example timer software has been constructed to work correctly, even if the caller specifies a delay of zero clock ticks. Tmset will correctly add the new request to the beginning of the delta list. When the tcptimer process awakens, it will remove the item from the delta list and deposit the message on the TCP output port. When the TCP output process receives the message, it will proceed to handle it.

Although the mechanism works correctly, scheduling an event with zero delay is inefficient because it forces the operating system to context switch between the calling process, the TCP timer process, and the TCP output process in rapid succession. Furthermore, scheduling a SEND event with zero delay occurs often (whenever the input process needs to send an ACK or whenever an application program generates output). To eliminate the unnecessary context switch, our example software provides procedure tcpkick that can be used to schedule a SEND without delay.

```
/* tcpkick.c - tcpkick */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpkick - make sure we send a packet soon
 */
int tcpkick(ptcb)
struct    tcb  *ptcb;
{
    int tcbnum = ptcb - &tcbtab[0]; /* for MKEVENT() */
}
```



```
int tv;

tv = MKEVENT(SEND, tcbnum);
if (ptcb->tcb_flags & TCBF_DELACK && !tmleft(tcps_oport, tv))
    tmset(tcps_oport, TCPQLEN, tv, TCP_ACKDELAY);
else if (pcount(tcps_oport) < TCPQLEN)
    psend(tcps_oport, tv); /* send now */
return OK;
}
```

After creating a needed event, `tcpkick` checks the TCB to see if it is using delayed ACKs (bit `TCBF_DELACK`). As long as ACKs are not delayed, `tcpkick` sends the message directly to the TCP output process. If it finds that ACKs should be delayed, `tcpkick` calls `tmset` to schedule the event in a short time.

14.11 Summary

TCP requires real-time processing to handle events like retransmission that must be scheduled to occur in the future. Our sample implementation stores delayed events on a delta list because it makes periodic updates efficient. Items on a delta list each correspond to a single event. The delta list arranges items by the time they will occur and stores time relative to the previous item on the list.

A single TCP process manages the delta list. It periodically decrements the remaining time in the first item on the list, and schedules the event when the time reaches zero. When an event occurs, the TCP timer process extracts an integer message and a port identifier from the event, and sends the message to that port. Thus, the timer process does not understand or interpret the messages stored in events,

14.12 FOR FURTHER STUDY

Comer [1987] describes delta list processing in more detail and gives invariants for maintaining times in relative form during the search.

14.13 EXERCISES

1. Devise a slightly different data structure that eliminates some or all of the special cases in `tmset`.
2. Step through the insertion of a new item on a delta list to see if you understand how the relative time is maintained during the search.



3. Rewrite `tcpkilltimers` to search the delta list and remove all items for a given TCB. How much more efficient is it than the current implementation?
4. How does the modification suggested in the previous exercise reduce the generality of the timing mechanism?
5. Would it be helpful to modify `tcpkick` to allow it to handle messages other than SEND? Why or why not?



15 TCP: Flow Control And Adaptive Retransmission

15.1 Introduction

TCP accommodates an extraordinary diversity of underlying physical networks by tolerating a wide range of delay, throughput, and packet loss rates, it handles each connection independently, allowing multiple connections from a single machine to each traverse a path with different underlying characteristics. More important, TCP adapts to changes in the round trip delay on a given connection, making it reliable even when the underlying packet switching system experiences congestion or temporary failures.

Adaptive retransmission lies at the heart of TCP and accounts for its success. In essence, adaptive retransmission uses recent past behavior to predict future behavior. It requires TCP to measure the round trip delay for each transmission, and to use statistical techniques to combine the individual measurements into a smoothed estimate of the mean round trip delay. Furthermore, TCP continually updates its round trip delay estimate as it acquires new measurements.

This chapter considers the implementation of software that provides adaptive retransmission. It discusses measurement of round trip times, statistical smoothing, retransmission timing, generation and processing of acknowledgements, and window-based flow control. It includes congestion-control and slow-start techniques as well as timer backoff and other optimizations.

Although the techniques discussed in this chapter require only a few lines of code to implement, their effect on TCP performance is dramatic. More important, they have arisen after much experimentation and careful analysis, so an average programmer is not likely to invent them independently. Finally, most of these techniques are now part of the TCP standard, so they must not be considered optional.



15.2 The Difficulties With Adaptive Retransmission

In principle, round trip estimation should be easy. However, problems in a practical internet impose several difficulties. Segments or acknowledgements can be lost or delayed making individual round trip measurements inaccurate. Bursty traffic from multiple sources can cause delays to fluctuate wildly. Furthermore, the load imposed by even a single connection can congest a network or gateway. Finally, retransmission after segment loss can cause congestion, or add to it.

Although the original TCP specification contained many subtle weaknesses and omissions, most of the adaptive retransmission problems have been solved either through improvements in statistical smoothing methods or through the use of practical heuristics.

15.3 Tuning Adaptive Retransmission

To achieve efficiency and robustness, TCP adaptive retransmission must be tuned in five principle areas:

- Retransmission timer and backoff
- Window-based flow control
- Maximum segment size computation
- Congestion avoidance and control
- Round trip estimation

The next sections examine each of these areas in detail, and show the implementation of techniques to resolve these subtle problems.

15.4 Retransmission Timer And Backoff

TCP uses a cumulative acknowledgement scheme in which each acknowledgement carries a sequence number. The sequence number specifies how many contiguous octets from the data stream the remote site has received correctly. Because acknowledgements do not specify individual segments and because acknowledgements can be lost, the sender cannot distinguish whether a given acknowledgement arose from an original transmission or the retransmission of a segment. Thus, the sender cannot accurately measure the round trip delay for retransmitted segments.

15.4.1 Karn's Algorithm

The standard specifies that TCP should use a technique known as Karn's algorithm to control the retransmission timer value. During normal data transfer, acknowledgements arrive for each segment before the retransmission timer expires. In



such cases, Karn's algorithm does not interfere with the usual process of measuring the round trip delay and computing a retransmission timeout for the next segment to be sent. However, because TCP cannot correctly associate acknowledgements with individual transmissions of a segment, Karn's algorithm specifies that TCP should ignore round trip measurements for all retransmitted segments. Furthermore, once retransmissions begin, Karn's algorithm separates the computation of retransmission timeouts from the previous estimate of round trip delay, doubling the timeout for each retransmission.

To implement Karn's algorithm, the software needs to store three pieces of information. First, it needs to store a value for retransmission timeout, which it computes from the current round trip estimate. Second, it needs to store an indication of whether TCP has begun retransmitting. Third, it needs to store a count of retransmissions. Our example code keeps all these values in fields of the TCB. Field tcb_rexmt stores the current value for the retransmission timer. If retransmission has begun, field tcb_ostate contains the value TCPO_REXMT. Finally, field tcb_rexmtcount records the current count of retransmissions.

15.4.2 Retransmit Output State Processing

Procedure tcprexmt implements the retransmission computation specified by Karn's algorithm.

```
/* tcprexmt.c - tcprexmt */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcprexmt - handle TCP output events while we are retransmitting
 */
int tcprexmt(tcbnum, event)
{
    struct    tcb  *ptcb = &tcbtab[tcbnum];

    if (event != RETRANSMIT)
        return OK;      /* ignore others while retransmitting */
    if (++ptcb->tcb_rexmtcount > TCP_MAXRETRIES) {
        tcpabort(ptcb, TCPE_TIMEDOUT);
        return OK;
    }
    tcpsend(tcbnum, TSF_REXMT);
```



```
tmset(tcbs_oport, TCPQLEN, MKEVENT(RETRANSMIT, tcbnum),
      min(ptcb->tcb_rexmt<<ptcb->tcb_rexmtcount, TCP_MAXRXT));
if (ptcb->tcb_ostate != TCPO_REXMT)
    ptcb->tcb_ssthresh = ptcb->tcb_cwnd; /* first drop */
ptcb->tcb_ssthresh = min(ptcb->tcb_swindow,ptcb->tcb_ssthresh)/2;
if (ptcb->tcb_ssthresh < ptcb->tcb_smss)
    ptcb->tcb_ssthresh = ptcb->tcb_smss;
ptcb->tcb_cwnd = ptcb->tcb_smss;
return OK;
}
```

Tcprexmt corresponds to the RETRANSMIT output state, and will be called by the TCP output process whenever a timer event occurs during retransmission. Because the connection has begun retransmission, events like SEND cannot be processed, so tcprexmt ignores all events except the RETRANSMIT event.

Tcprexmt increments the retransmission count in field tcb_rexmtcount and enforces a maximum retransmission count by comparing it to the constant TCP_MAXRETRIES. When it reaches the maximum allowed count, tcprexmt calls tcpabort to abort the connection, passing it the error code TCPE_TIMEDOUT. After tcprexmt has checked for errors, it calls tcpsend to retransmit the unacknowledged data that remains in the output buffer. The second argument to tcpsend specifies that this call is for retransmission.

Once tcprexmt retransmits the data, it needs to schedule another retransmission timeout in the future. The call to tmset implements timer control according to Karn's algorithm. It shifts the timeout in tcb_rexmt left tcb_rexmtcount bits to double the delay for each retransmission that has occurred. It then passes the computed delay as an argument to tmset, causing it to schedule a new RETRANSMIT event.

For small values of TCP_MAXRETRIES, doubling the timeout on each retransmission works well. However, if the system allows a large number of retries, doubling the timeout on each can result in severe delays before TCP decides to abort a connection. To prevent the timeout from becoming arbitrarily large, tcprexmt enforces a maximum timeout by choosing the minimum of the computed timeout and constant TCP_MAXRXT.

Section 15.7.1 discusses the final few statements in tcprexmt, which handle congestion control.

15.5 Window-Based Flow Control

When TCP on the receiving machine sends an acknowledgement, it includes a



window advertisement in the segment to tell the sender how much buffer space the receiver has available for additional data. The window advertisement always specifies the data the receiver can accept beyond the data being acknowledged, and TCP mandates that once a receiver advertises a given window, it may never advertise a subset of that window (i.e., the window never shrinks). Of course, as the sender fills the advertised window, the value in the acknowledgement field increases and the value in the window field may become smaller until it reaches zero. However, the receiver may never decrease the point in the sequence space through which it has agreed to accept data. Thus, the window advertisement can only decrease if the sender supplies data and the acknowledgement number increases; it cannot decrease merely because the receiver decides to decrease its buffer size.

TCP uses window advertisements to control the flow of data across a connection. A receiver advertises small window sizes to limit the data a sender can generate. In the extreme case, advertising a window size of zero halts transmission altogether^①.

15.5.1 Silly Window Syndrome

If a receiver advertises buffer space as soon as it becomes available, it may cause behavior known as the silly window syndrome. Silly window behavior is characterized as a situation in which the receiver's window oscillates between zero and a small positive value, while the sender transmits small segments to fill the window as soon as it opens. Such behavior leads to low network utilization because each segment transmitted contains little data compared to the overhead for TCP and IP headers.

To prevent a TCP peer from falling victim to the silly window syndrome when transmitting, TCP uses a technique known as receiver-side silly window avoidance. The silly window avoidance rule states that once a receiver advertises a zero window, it should delay advertising a nonzero window until it has a nontrivial amount of space in its buffer. A nontrivial amount of buffer space is defined to be the space sufficient for one maximum-sized segment or the space equivalent to one quarter of the buffer, whichever is larger.

15.5.2 Receiver-Side Silly Window Avoidance

Procedure `tcpwindow` implements receiver-side silly window avoidance when it computes a window advertisement.

```
/* tcprwindow.c - tcprwindow */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

^① We say that the receiver "closes" the window.



```
/*
 * tcprwindow - do receive window processing for a TCB
 */
int tcprwindow(ptcb)
struct tcb *ptcb;
{
    int window;

    window = ptcb->tcb_rbsize - ptcb->tcb_rbcnt;
    if (ptcb->tcb_state < TCPS_ESTABLISHED)
        return window;
    /*
     * Receiver-Side Silly Window Syndrome Avoidance:
     * Never shrink an already-advertised window, but wait for at
     * least 1/4 receiver buffer and 1 max-sized segment before
     * opening a zero window.
    */
    if (window*4 < ptcb->tcb_rbsize || window < ptcb->tcb_rmss)
        window = 0;
    window = max(window, ptcb->tcb_cwin - ptcb->tcb_rnext);
    ptcb->tcb_cwin = ptcb->tcb_rnext + window;
    return window;
}
```

Tcprwindow begins by computing a window size equal to the available buffer space (i.e., the size of the receive buffer minus the current count of characters in the buffer). If TCP has just begun a three-way handshake, but has not yet established a connection (the state is less than TCPS_ESTABLISHED), the receiver maximum segment size has not been initialized. Therefore, rcpwindow cannot apply receiver-side silly window avoidance — it merely stores the value computed for the window in field tcb_window of the TCB and returns to its caller. Once a connection has been established, tcprwindow applies the rule for receiver-side silly window avoidance, by reducing the window to zero unless a nontrivial amount of space is available.

The final statements of tcprwindow apply congestion avoidance to the window advertisement as discussed below.

15.5.3 Optimizing Performance After A Zero Window

Once a receiver advertises a zero window, the sender enters the PERSIST output



state and begins to probe the receiver^①. The receiver responds to each probe by sending an acknowledgement. As long as the window remains closed, the probes continue, and the acknowledgements contain a window advertisement of zero. Eventually, when sufficient space becomes available, the acknowledgements will carry a nonzero window, and the sender will start to transmit new data.

Although the sender bears ultimate responsibility for probing a zero window, a minor optimization can improve performance. The optimization consists of arranging for the receiver to generate a gratuitous acknowledgement that contains the new window size, without waiting for the next probe. Thus, in our implementation, whenever an application program extracts data from a TCP input buffer, it checks to see if the additional space causes the window to open, and sends a gratuitous acknowledgement if it does. As the sender processes the acknowledgement, it finds the nonzero window advertisement, moves back to the TRANSMIT state, and resumes transmission of data.

15.5.4 Adjusting The Sender's Window

Procedure `tcpwindow` computes the size of the sender's window. It handles window advertisements in incoming segments, and keeps track of the amount of data the peer TCP is willing to receive.

```
/* tcpwindow.c - tcpwindow */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpwindow - handle send window updates from remote
 */
int tcpwindow(ptcb, pep)
struct tcb *ptcb;
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct tcp     *ptcp = (struct tcp *)pip->ip_data;
    tcpseq        wlast, owlast;

    if (SEQCMP(ptcp->tcp_seq, ptcb->tcb_lwseq) < 0)
        return OK;
    if (SEQCMP(ptcp->tcp_seq, ptcb->tcb_lwseq) == 0 &&
```

^① Chapter 13 discusses output processing and the output state machine.



```
SEQCMP(ptcp->tcp_ack, ptcb->tcb_lwack) < 0)
    return OK;

/* else, we have a send window update */

/* compute the last sequences of the new and old windows */

owlast = ptcb->tcb_lwack + ptcb->tcb_swindow;
wlast = ptcp->tcp_ack + ptcp->tcp_window;

ptcb->tcb_swindow = ptcp->tcp_window;
ptcb->tcb_lwseq = ptcp->tcp_seq;
ptcb->tcb_lwack = ptcp->tcp_ack;
if (SEQCMP(wlast, owlast) <= 0)
    return OK;
/* else, window increased */
if (ptcb->tcb_ostate == TCPO_PERSIST) {
    tmclear(tcps_oport, MKEVENT(PERSIST, ptcb-&tcbtab[0]));
    ptcb->tcb_ostate = TCPO_XMIT;
}
tcpkick(ptcb);           /* do something with it */
return OK;
}
```

In the TCB, field tcb_swindow always contains the number of bytes that TCP can send beyond the currently acknowledged sequence. That is, it contains the value from the most recently received window advertisement. However, because segments can arrive out of order, TCP must be careful when updating tcb_swindow. It must verify that the incoming segment was generated after the segment that was last used to update the window. To do so, it keeps a record of the sequence (tcb_lwseq) and acknowledgement (tcb_lwack) fields from the segment whenever it updates the window.

When a segment arrives, tcpswindow compares the sequence and acknowledgement fields to the stored values. If the value in the sequence field is smaller than the stored sequence value, the segment has arrived out of order and the window advertisement must be ignored. Furthermore, if the sequence number in the segment matches the stored sequence value, but the acknowledgement in the segment is smaller than the stored acknowledgement, the acknowledgement has arrived out of order, so the window advertisement must be ignored. When tcpswindow determines that the segment contains a valid advertisement, it stores the new window size in field tcb_swindow and updates the stored sequence and acknowledgement values.



15.6 Maximum Segment Size Computation

We saw that when tcprwindow applies receiver-side silly window avoidance, it needs to know the size of the largest possible segment that can be expected to arrive. In addition, when TCP generates segments that carry data, it limits their size to the maximum segment site (MSS) allowed for the connection. TCP negotiates the MSS for both outgoing and incoming segments when it exchanges requests during the three-way handshake. Once it establishes an MSS in each direction, TCP never changes them.

15.6.1 The Sender's Maximum Segment Size

To understand how the example TCP software chooses a maximum segment size for output, look again at procedure tcpwinit in Chapter 12^①. Tcpwinit computes an initial value for the maximum segment size (MSS), and stores it in field tcb_smss of the TCB. To help avoid IP fragmentation, the host requirements document specifies that TCP must use an initial maximum segment size of 536 octets if the connection passes through a gateway. For connections that lie on a directly connected network, TCP chooses an initial value such that the network packets will be as full as possible (i.e., it computes an initial maximum data size by subtracting the size of TCP and IP headers from the MTU for the local network used to reach the remote machine). Tcpwinit determines whether the connection will pass through a gateway by finding whether the route to the destination has a metric greater than zero.

After choosing an initial MSS, TCP processes the maximum segment size option found in incoming SYN segments. Procedure tcpsmss handles the details of processing the MSS option.

```
/* tcpsmss.c - tcpsmss */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpsmss - set sender MSS from option in incoming segment
 */
int tcpsmss(ptcb, ptcp, popt)
struct tcb *ptcb;
struct tcp *ptcp;
```

^① Tcpwinit initializes the MSS when a remote client establishes a connection to a local server; a similar piece of code initializes the MSS when a local client forms a connection to a remote server.



```
char      *popt;
{
    unsigned mss, len;

    len = *++popt;
    ++popt;      /* skip length field */
    if ((ptcp->tcp_code & TCPF_SYN) == 0)
        return len;
    switch (len-2) { /* subtract kind & len */
    case sizeof(char):
        mss = *popt;
        break;
    case sizeof(short):
        mss = net2hs(*(unsigned short *)popt);
        break;
    case sizeof(long):
        mss = net2hl(*(unsigned long *)popt);
        break;
    default:
        mss = ptcb->tcb_smss;
        break;
    }
    mss -= TCPMHLEN; /* save just the data buffer size */
    if (ptcb->tcb_smss)
        ptcb->tcb_smss = min(mss, ptcb->tcb_smss);
    else
        ptcb->tcb_smss = mss;
    return len;
}
```

A maximum segment size can only be negotiated during the three-way handshake, so `tcpsmss` ignores the option unless the segment carrying it has the SYN bit set. It then selects one of four cases, using the number of octets in the option value to choose a case. Our implementation supports MSS option values of 8, 16, or 32 bits^①. `Tcpsmss` extracts the option value and converts it to local machine byte order. In other cases, `tcpsmss` substitutes the initial MSS from the TCB. Finally, after extracting a value for the MSS from the option, `tcpsmss` compares it to the initial MSS in the TCB, and uses the minimum of the two. Thus, `tcpsmss` never allows the MSS option on an incoming segment to increase the initial MSS value.

^① Many implementations only support 16 bit values.



15.6.2 Option Processing

Procedure tcpopts handles option processing, and calls tcpsmss to extract the MSS option.

```
/* tcpopts.c - tcpopts */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpopts - handle TCP options for an inbound segment
 */
int tcpopts(ptcb, pep)
struct tcb *ptcb;
struct ep *pep;
{
    struct ip *pip = (struct ip *)pep->ep_data;
    struct tcp *ptcp = (struct tcp *)pip->ip_data;
    char *popt, *popend;
    int len;

    if (TCP_HLEN(ptcp) == TCPMHLEN)
        return OK;
    popt = ptcp->tcp_data;
    popend = &pip->ip_data[TCP_HLEN(ptcp)];
    do {
        switch (*popt) {
        case TPO_NOOP:      popt++;
            /* fall through */
        case TPO_EOOL:      break;
        case TPO_MSS:
            popt += tcpsmss(ptcb, ptcp, popt);
            break;
        default:
            break;
        }
    } while (*popt != TPO_EOOL && popt<popend);

    /* delete the options */
}
```



```
len = pip->ip_len-IP_HLEN(pip)-TCP_HLEN(ptcp);
if (len)
    blkcopy(ptcp->tcp_data,&pip->ip_data[TCP_HLEN(ptcp)],len);
pip->ip_len = IP_HLEN(pip) + TCPMHLEN + len;
ptcp->tcp_offset = TCPHOFFSET;
return OK;
}
```

Because the current TCP standard specifies only one real option, MSS, the code is extremely simple. In addition to the MSS option, tcopts must also handle option codes that denote no-operation and end of options.

Once tcopts reaches the end-of-options code, it deletes the options field altogether by moving the data portion of the segment and adjusting the length field in the segment header. Removing the option field makes it possible for procedures throughout the TCP software to assume a fixed offset for the data.

15.6.3 Advertising An Input Maximum Segment Size

Procedure tcprmss creates the maximum segment size option in a SYN segment. It assumes the maximum segment size has already been computed and stored in the TCB.

```
/* tcprmss.c - tcprmss */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcprmss - set receive MSS option
 */
int tcprmss(ptcb, pip)
struct tcb *ptcb;
struct ip *pip;
{
    struct tcp *ptcp = (struct tcp *)pip->ip_data;
    int mss, hlen, olen, i;

    hlen = TCP_HLEN(ptcp);
    olen = 2 + sizeof(short);
    pip->ip_data[hlen] = TPO_MSS;           /* option kind */
    pip->ip_data[hlen+1] = olen;            /* option length */
```



```
mss = hs2net((short)ptcb->tcb_smss);
for (i=olen-1; i>1; i--) {
    pip->ip_data[hlen+i] = mss & LOWBYTE;
    mss >>= 8;
}
hlen += olen + 3; /* +3 for proper rounding below */
/* header length is high 4 bits of tcp_offset, in longs */
ptcp->tcp_offset = ((hlen<<2) & 0xf0) | ptcp->tcp_offset & 0xf;
}
```

Option creation is straightforward. The option consists of a single octet containing TPO_MSS, a single octet that gives the option length, and a binary integer in network byte order that contains the maximum segment size. Tcprmss adjusts the TCP header length in the segment to include the option octets,

15.7 Congestion Avoidance And Control

When congestion occurs, delays increase, causing TCP to retransmit segments. In the worst case, retransmissions increase congestion and produce an effect known as congestion collapse. To avoid adding to congestion, the standard now specifies that TCP should use strategies that reduce retransmission when packed delay or loss occurs. The first strategy is known as multiplicative decrease.

15.7.1 Multiplicative Decrease

The idea behind multiplicative decrease is simple: the sender-side of TCP maintains an internal variable known as the congestion window that it uses to restrict the amount of data being sent. When transmitting, TCP uses the minimum of the receiver's advertised window and the internal congestion window to determine how much data to send.

To compute the congestion window size, assume the number of retransmissions provides a measure of congestion in the internet. While no congestion or loss occurs, set the congestion window size to the receiver's advertised window size. That is, use the receiver's advertised window to determine how much data to send. When congestion begins (i.e., when a retransmission occurs), reduce the congestion window size by a multiplicative constant. In particular, reduce the congestion window by half each time retransmission occurs, but never reduce it to less than the size required for one segment.

Procedure `tcpexmt`, shown in section 15.4.2, implements multiplicative decrease. In the code, variable `tcb_cwnd` contains the congestion window size. `Tcpexmt` sets variable `tcb_ssthresh` to one half of either the advertised window (`tcb_swindow`) or the



current congestion window (tcb_cwnd), whichever is smaller. It then checks to make sure that the computed value does not go below 1 MSS.

Although the technique is called multiplicative, the congestion window threshold will decrease exponentially when measured in lost segments. The first loss drops it to one-half of the original window, the second to one-quarter, the third to one-eighth, and so on.

15.8 Slow-Start And Congestion Avoidance

15.8.1 Slow-start

We said that when an internet carrying TCP segments becomes congested, additional retransmissions can exacerbate the situation. To help recover from congestion, the standard now requires TCP to reduce its rate of transmission. In particular, TCP assumes that packet loss results from congestion, and immediately uses a technique known as slow-start during the recovery. To further improve performance and to avoid having new connections add to the congestion, TCP uses slow-start whenever it starts sending data on a newly established connection.

Slow-start is the reverse of multiplicative decrease — it provides multiplicative increase^①. The idea is again simple: start the congestion window at the size of a single segment (the MSS) and send it. If communication is successful and an acknowledgement arrives before the retransmission timer expires, add one segment to the congestion window size (i.e., double it to two segments). Continue adding one segment to the congestion window each time an acknowledgement arrives. Thus, if both segments arrive successfully in the second round of transmissions, the congestion window will increase to 4 segments, and it will continue to increase exponentially until it reaches the threshold that has been set by multiplicative decrease.

15.8.2 Slower Increase After Threshold

Once the congestion window reaches the threshold, TCP slows down. Instead of adding a new segment to the congestion window every time an acknowledgement arrives, TCP increases the congestion window size by one segment for each round trip time. To estimate a round trip time, the code uses the time to send and receive acknowledgements for the data in one window. Of course, TCP does not wait for an entire window of data to be sent and acknowledged before increasing the congestion window size. Instead, it adds a small increment to the congestion window size each time an acknowledgement arrives. The small increment is chosen to make the increase

^① Slow-start is an unfortunate name because it starts flow quickly in the absence of loss; it only remains slow if loss continues.



average approximately one segment over an entire window. To understand how the idea translates into code, think of TCP sending maximum size segments, and remember that we want to increase the congestion window by:

```
increase = segment / window
```

Because the system has experienced congestion, the current window is limited to the congestion window size, which means that the number of increments TCP makes is determined by the number of segments that fit in the congestion window.

```
segments per window = congestion window / max segment size
```

Let N denote the segments per window. To increment by one segment over the entire window, TCP increments by $1/N$ for each of the N acknowledgements. Thus, when an acknowledgement arrives, TCP increments by:

```
increment = (one segment / N)
= (MSS bytes / N)
= MSS / (congestion window/MSS)
```

or

```
increment = ( MSS * MSS ) / congestion window
```

15.8.3 Implementation Of Congestion Window Increase

The last few lines of procedure `tcprrtt` implement congestion window increase when acknowledgements arrive^①.

```
/* tcprrtt.c - tcprrtt */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcprrtt - do round trip time estimates & cong. window processing
 */
int tcprrtt(ptcb)
struct tcb *ptcb;
{
    int rrt, /* raw round trip */
        delta; /* deviation from smoothed */

    rrt = tmclear(tcps_oport, MKEVENT(RETRANSMIT, ptcb->tcbtab[0]));
}
```

^① The remaining code in `tcprrtt` participates in retransmission timer estimation, and is discussed in detail later.



```
if (rrt != SYSERR && ptcb->tcb_ostate != TCPO_REXMT) {  
    if (ptcb->tcb_srt == 0)  
        ptcb->tcb_srt = rrt<<3; /* prime the pump */  
    /*  
     * "srt" is scaled X 8 here, so this is really:  
     *   delta = rrt - srt  
     */  
    delta = rrt - (ptcb->tcb_srt>>3);  
    ptcb->tcb_srt += delta;      /* srt = srt + delta/8 */  
    if (delta < 0)  
        delta = -delta;  
    /*  
     * "rtde" is scaled X 4 here, so this is really:  
     *   rtde = rtde + (|delta| - rtde)/4  
     */  
    ptcb->tcb_rtde += delta - (ptcb->tcb_rtde>>2);  
    /*  
     * "srt" is scaled X 8, rtde scaled X 4, so this is:  
     *   rto = srt + 2 * rtde  
     */  
    ptcb->tcb_rexmt = ((ptcb->tcb_srt>>2)+ptcb->tcb_rtde)>>1;  
    if (ptcb->tcb_rexmt < TCP_MINRXT)  
        ptcb->tcb_rexmt = TCP_MINRXT;  
    }  
    if (ptcb->tcb_cwnd < ptcb->tcb_ssthresh)  
        ptcb->tcb_cwnd += ptcb->tcb_smss;  
    else  
        ptcb->tcb_cwnd += (ptcb->tcb_smss * ptcb->tcb_smss) /  
            ptcb->tcb_cwnd;  
    return OK;  
}
```

The final if statement of tcprtt handles congestion window increase. If variable tcb_cwnd is less than tcb_ssthresh, procedure tcprtt provides slow-start by incrementing the congestion window by the maximum segment size. If the congestion window has passed the threshold, tcprtt uses additive increase and increments the congestion window by MSS^2/tcb_cwnd .



15.9 Round Trip Estimation And Timeout

From the beginning, researchers recognized that TCP performance depends on its ability to estimate the mean of the round trip time on a connection. The best way to think of the problem is to imagine a sequence of round trip measurements that arrive over time. TCP uses the history of measurements to estimate the current round trip delay, and chooses a retransmission timeout derived from its estimate of round trip delay. Because the round trip delay varies over time, TCP weights recent measurements more heavily than older ones. However, because individual measurements of round trip delay can fluctuate wildly from the norm when congestion occurs, TCP cannot ignore the history of measurements completely.

Performance studies have shown that TCP can exhibit significantly higher throughput if it estimates the variance in round trip delay as well as the mean. Knowing the variance makes it possible to compute a timeout that accommodates expected fluctuations without retransmitting unnecessarily. The standard now specifies using the improved round trip estimation technique described here.

15.9.1 A Fast Mean Update Algorithm

It would be foolish for TCP to keep a history of round trip measurements for purposes of computing the mean and variance in round trip delay because good incremental algorithms exist. Thus, TCP keeps a "running average" which it updates each time it obtains a new measurement. For example, it updates the average by computing:

```
error = measurement - average
```

and

```
average = average + δ*error
```

where δ is a fraction less than 1. In fact, TCP can keep a "running mean deviation" and use the error term above to update the deviation:

```
deviation = deviation + (|error| - deviation)
```

In practice, implementations achieve efficiency by scaling the computation by a power of two, and using integer arithmetic instead of floating point. For example, choosing:

$$\delta = 1/2^n$$

allows the code to perform division by shifting. The value $n = 3$ is convenient. If average stores a scaled form of the average, the code becomes:

```
error = measurement - (average >> 3);
average = average + error;
if (error < 0)
    error = -error;
```



```
error = error - (deviation >> 3);
deviation = deviation + error;
```

To further improve performance, TCP uses a slightly larger value for δ , making the final form:

```
error = measurement - (average >> 3);
average = average + error;
if (error < 0)
    error = -error;
error = error - (deviation >> 2);
deviation = deviation + error;
retransmission_timer = ((average>>2)+ deviation) >> 1;
```

Procedure `tcprrt` implements round trip estimation. It is called whenever an acknowledgement arrives. Thus, a call to `tcprrt` signals the end of retransmissions, and the output state reverts to TRANSMIT.

`Tcprrt` first calls `tmclear`. The call accomplishes two things: it deletes the pending retransmission event for the connection, and it computes the time that has elapsed since the event was scheduled.

Recall that acknowledgement ambiguity makes round trip measurement impossible for retransmitted segments, and that Karn's algorithm specifies ignoring acknowledgements for retransmitted segments. `Tcprrt` implements Karn's algorithm by testing to make sure the round trip measurement is valid and that the acknowledgement does not correspond to a retransmitted segment (i.e., TCP is not in The RETRANSMIT state).

If the elapsed time, `rtt` is valid, `tcprrt` assumes it provides a raw measure of round trip delay, and uses it to compute a new value for the smoothed round trip time, `tcb_srt`. It also produces a new estimate for the mean deviation (`tcb_rtde`). Finally, it computes a retransmission timeout from the new round trip estimate and stores it in `tcb_rexmt`. Note that our implementation uses constant `TCP_MINRXT` as a fixed minimum retransmission delay to insure that the computed value does not approach zero.

15.9.2 Handling Incoming Acknowledgements

Procedure `tcpacked` handles acknowledgements in incoming segments. It computes and returns to its caller the number of octets in the sequence space beyond those octets acknowledged by previous segments.

```
/* tcpacked.c - tcpacked */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```



```
/*-----  
 * tcpacked - handle in-bound ACKs and do round trip estimates  
 *-----  
 */  
  
int tcpacked(ptcb, pep)  
struct tcb *ptcb;  
struct ep *pep;  
{  
    struct ip *pip = (struct ip *)pep->ep_data;  
    struct tcp *ptcp = (struct tcp *)pip->ip_data;  
    int acked, tcbnum, cacked;  
    STATWORD ps;  
  
    if (!(ptcp->tcp_code & TCPF_ACK))  
        return SYSERR;  
    acked = ptcp->tcp_ack - ptcb->tcb_suna;  
    cacked = 0;  
    if (acked <= 0)  
        return 0; /* duplicate ACK */  
    if (SEQCMP(ptcp->tcp_ack, ptcb->tcb_snext) > 0)  
        if (ptcb->tcb_state == TCPS_SYNRCVD)  
            return tcpreset(pep);  
        else  
            return tcpackit(ptcb, pep);  
    tcprtt(ptcb);  
    ptcb->tcb_suna = ptcp->tcp_ack;  
    if (acked && ptcb->tcb_code & TCPF_SYN) {  
        acked--;  
        cacked++;  
        ptcb->tcb_code &= ~TCPF_SYN;  
        ptcb->tcb_flags &= ~TCBF_FIRSTSEND;  
    }  
    if ((ptcb->tcb_code & TCPF_FIN) &&  
        SEQCMP(ptcp->tcp_ack, ptcb->tcb_snext) == 0) {  
        acked--;  
        cacked++;  
        ptcb->tcb_code &= ~TCPF_FIN;  
        ptcb->tcb_flags &= ~TCBF_SNDFIN;  
    }  
    if ((ptcb->tcb_code & TCPF_SUPOK) &&
```



```
SEQCMP(ptcp->tcp_ack, ptcb->tcb_supseq) >= 0) {  
    ptcb->tcb_sbstart = (ptcb->tcb_sbstart+acked) % ptcb->tcb_sbsize;  
    ptcb->tcb_sbcount -= acked;  
    if (acked) {  
        disable(ps);  
        if (scount(ptcb->tcb_ssema) <= 0)  
            signal(ptcb->tcb_ssema);  
        restore(ps);  
    }  
    tcpstate(ptcb, acked+cacked);  
    return acked;  
}  
}
```

If called with a segment that does not contain an acknowledgement bit, `tcpacked` returns an error value. If an acknowledgement is present, `tcpacked` computes the number of new octets acknowledged (`acked`) by subtracting the start of unacknowledged data stored in the TCB from the acknowledgement number in the segment. If the acknowledgement specifies a sequence number less than the currently recorded start of unacknowledged data, the segment must be a duplicate or have arrived out of order, so `tcpacked` returns zero to indicate that no additional octets were acknowledged.

`Tcpacked` includes a special check for acknowledgements that specify a sequence number beyond the sequence number of data that has been sent. For most states, the standard specifies that TCP must acknowledge such segments. Thus, `tcpacked` calls `tcpackit` to generate an acknowledgement. For the SYN-RECEIVED state, however, an acknowledgement beyond the current sequence number means an incorrect 3-way handshake and must be answered by a RESET.

Once `tcpacked` has checked to see that the acknowledgement lies in the expected range, it calls `tcprrt` to update the smoothed round trip estimate and compute a new retransmission timeout. It also updates field `tcb_sunna`, which contains the starting sequence number of unacknowledged data.

`Tcpacked` handles two special cases: FIN and SYN processing. Conceptually, both lie in the sequence space. `Tcpacked` records the presence of a SYN by clearing bit `TCBF_FIRSTSEND` in the TCB. It also decrements by 1 the count of acknowledged data returned to the caller, because the count specifies only data and should not include the SYN. Similarly, `tcpacked` clears bit `TCPF_FIN` in the TCB code flags if the segment acknowledges a FIN for the connection.

The final section of `tcpacked` updates variables in the TCB to reflect changes caused by the arrival of the acknowledgement. Basically, it manipulates counters and buffer pointers to discard outgoing data in the send buffer that has been acknowledged. First, it moves the buffer pointer (`tcb_sbstart`) forward acked positions, wrapping around



to the start of the buffer if it passes the end. Second, it subtracts the number of octets acked from the count of data in the buffer (tcb_sbcount). Third, if acknowledged data has been removed from the buffer, and one or more application programs are blocked waiting for space in the buffer, tcpacked signals the send buffer semaphore, allowing the next program to write into the buffer.

15.9.3 Generating Acknowledgments For Data Outside The Window

We said that TCP was required to generate an acknowledgement in response to incorrect incoming acknowledgements. In particular, TCP sends an acknowledgement whenever the remote side acknowledges data that lies beyond the current output window. Procedure tcpackit generates such acknowledgements.

```
/* tcpackit.c - tcpackit */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 *  tcpackit - generate an ACK for a received TCP packet
 *-----*
 */
int tcpackit(ptcb, pepin)
struct tcb *ptcb;
struct ep *pepin;
{
    struct ep *pepout;
    struct ip *pipin = (struct ip *)pepin->ep_data, *pipout;
    struct tcp     *ptcpin = (struct tcp *)pipin->ip_data, *ptcpout;

    if (ptcpin->tcp_code & TCPF_RST)
        return OK;
    if (pipin->ip_len <= IP_HLEN(pipin) + TCP_HLEN(ptcpin) &&
        !(ptcpin->tcp_code & (TCPF_SYN|TCPF_FIN)))
        return OK;      /* duplicate ACK */
    pepout = (struct ep *)getbuf(Net.netpool);
    if (pepout == SYSERR)
        return SYSERR;
    pipout = (struct ip *)pepout->ep_data;
    blkcopy(pipout->ip_src, pipin->ip_dst, IP_ALEN);
    blkcopy(pipout->ip_dst, pipin->ip_src, IP_ALEN);
    ptcfout = (struct tcp *)pipout->ip_data;
```



```
ptcpout->tcp_sport = ptcpin->tcp_dport;
ptcpout->tcp_dport = ptcpin->tcp_sport;
ptcpout->tcp_seq = ptcb->tcb_snxt;
ptcpout->tcp_ack = ptcb->tcb_rnxt;
ptcpout->tcp_code = TCPF_ACK;
ptcpout->tcp_offset = TCPHOFFSET;
ptcpout->tcp_window = tcprwindow(ptcb);
ptcpout->tcp_urgptr = 0;
ptcpout->tcp_cksum = 0;
tcp2net(ptcpout);
ptcpout->tcp_cksum = tcpcsum(pipout);
TcpOutSegs++;
return ipsend(pipout->ip_dst, pepout, TCPMHLEN, IPT_TCP,
IPP_NORMAL, IP_TTL);
}
```

To prevent infinite loops, tcpakit does not respond to a RESET segment. For all others, it allocates a network buffer, fills in the TCP and IP headers, sets the ACK bit, and calls ipsend to forward the datagram on toward its destination.

15.9.4 Changing Output State After Receiving An Acknowledgement

After an acknowledgement arrives, TCP must examine the TCB to determine whether the output state should be changed. If all outstanding data has been acknowledged, the output reverts to the idle state. If some outstanding data has been acknowledged, the output changes from the retransmit state to the transmit state. Tcpakit calls procedure tcpostate to make the necessary tests.

```
/* tcpostate.c - tcpostate */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpostate - do TCP output state processing after an ACK
 */
int tcpostate(ptcb, acked)
struct tcb *ptcb;
int acked;
{
```



```
int tcbnum = ptcb - &tcbtab[0];

if (acked <= 0)
    return OK; /* no state change */
if (ptcb->tcb_ostate == TCPO_REXMT) {
    ptcb->tcb_rexmtcount = 0;
    ptcb->tcb_ostate = TCPO_XMIT;
}
if (ptcb->tcb_sbcount == 0) {
    ptcb->tcb_ostate = TCPO_IDLE;
    return OK;
}
tcpkick(ptcb);
return OK;
}
```

Tcpstate receives two arguments, a pointer to a TCB and the amount of new data acknowledged by the ACK that arrived. If no new data has been acknowledged, the state should not change. Tcpstate checks argument acked, and returns immediately if it is zero.

Tcpstate then examines the current output state given by field tcb_ostate. If the connection was in the retransmit state, it moves to the transmit state. Furthermore, if all outstanding data has been acknowledged, the connection moves to the idle state. Finally, if outstanding data remains, tcpstate calls tcpkick to initiate a transmission event.

15.10 Miscellaneous Notes And Techniques

Procedure tcprtt also aids sender-side silly window avoidance. It does so by checking for a pending RETRANSMIT event before scheduling another one. The idea is to avoid needless transmissions, because they trigger a sequence of acknowledgements. If the peer does not implement silly window avoidance, the acknowledgments may report small increments in window size, which, in turn, can cause the sender to transmit small segments. Thus, avoiding unnecessary acknowledgements helps conserve network bandwidth and prevent silly window behavior.

15.11 Summary

Adaptive retransmission lies at the heart of TCP, and makes it operate over a wide variety of underlying networks. To make TCP robust and efficient, adaptive



retransmission algorithms must be tuned in several areas, including: retransmission timer backoff, window-based flow control, maximum segment size computation, congestion avoidance and control, and round trip estimation.

Experience and analysis have produced techniques and heuristics that are now either suggested or required by the TCP protocol standard. While each technique requires only a few lines of code, taken together they make a significant difference in raising throughput and lowering delay.

15.12 FOR FURTHER STUDY

Postel [RFC 793] contains the original standard for TCP that includes a description of adaptive retransmission; many of the techniques found in this chapter have been added by later RFCs. Braden [RFC 1121] incorporates significant changes in the standard. Clark [RFCs 813 and 816] describe window management and fault recovery. Postel [RFC 879] comments on maximum segment size. Nagle [RFC 865] gives the technique for silly window avoidance. Karn and Partridge [1987] reviews TCP performance improvements, including estimation of round trip times and Karn's algorithm. Jacobson [1988] gives the congestion control algorithms that are now a required part of the standard. Mills [RFC 889] discusses measurement of Internet round trip delays. Borman [April 1989] summarizes experiments with high-speed TCP on Cray computers.

15.13 EXERCISES

1. This chapter mentioned using both sender-side and receiver-side silly window avoidance techniques. Will the receiver-side technique perform well even if the sender does not use silly window avoidance? Explain.
2. To find out how much slow-start limits throughput on an Ethernet connection (MTU=1500 octets), assume a round trip delay of 3 milliseconds. Calculate the throughput of the first 32 packets sent (a) using slow-start, an (b) without slow-start.
3. Examine the code call to `tcpstate` in `tcpacked` and suggest an optimization.
4. If a sender has a 16K byte buffer and a 1K byte maximum segment size, how many lost acknowledgments does it take before the congestion window reaches 1 MSS?
5. Karn's algorithm specifies ignoring round trip estimates when segments must be retransmitted. What happens if TCP always associates ACKs with the original transmission? With the most recent retransmission?



6. Read Jacobson [1988], a paper on congestion avoidance. What does it suggest will happen if the multiplicative decrease uses a value of 8 instead of 5?
7. Consider the effect of buffer size on throughput. Suppose two different implementations of TCP use 16K byte and 64K byte output buffers, and both implementations transmit across a single, congested path to receivers with large receiver-side buffers. How does the loss of a single segment affect throughput on each of two connections? The loss of 10 successive segments?
8. What should happen when TCP receives an ICMP source quench message that corresponds to an active connection? Why? (Hint: consider the relationship between source quench and congestion.)
9. Read about MTU discovery techniques. Explain how TCP can use them.
10. Argue that requiring an MSS of 536 for nonlocal connections places an unnecessary limit on TCP.
11. Will our example implementation work correctly on a local network that has an MTU just slightly less than the input (or output) buffer size? Explain.
12. Our implementation uses an aggressive acknowledgement policy (i.e., it does not delay sending acknowledgements to wait for window changes or data flowing in the opposite direction). How does aggressive acknowledgement improve performance? How can delayed acknowledgements affect slow-start and congestion control?
13. TCP does not provide a keepalive function in the protocol. However, some applications need to know when a connection fails even if they are not sending data. Extend our implementation to provide a keepalive function.
14. Some applications prefer to have TCP shutdown a connection that has been idle more than N minutes, where N is an argument specified by the application. Implement an automatic shutdown mechanism for idle connections.
15. Look at each state in the TCP finite state machine (Figure 11.2), and consider what happens to one side of a connection if the computer on the other side crashes and does not reboot. Determine which states need a long-term timeout to delete the TCB.



16 TCP: Urgent Data Processing And The Push Function

16.1 Introduction

Previous chapters considered conventional TCP input and output as well as heuristics to improve flow control and adaptive retransmission. This chapter considers two remaining aspects of TCP: its ability to send out-of-band notification, and its ability to bypass normal buffering. Out-of-band notification provides increased functionality and makes it possible to build programs that provide for abnormal events such as a program interruption or program abort function. Buffering is important because it makes TCP more efficient and lowers network overhead. However, buffering increases data transmission delays. Programs that need immediate delivery can bypass buffering and force TCP to deliver data without delay. The next sections consider how TCP provides out-of-band notification and how an application can bypass buffering.

16.2 Out-Of-Band Signaling

TCP uses a stream paradigm for normal data transfer. An application at one end of a connection creates a data stream, which it passes to TCP on its local machine. TCP sends data from the stream across the internet to TCP on the machine at the other end of the connection, which delivers it to the application on that machine. The stream paradigm works well for many applications, but does not suffice for all communication because it forces the receiver to process all data in sequence. Sometimes, an application needs to communicate an out-of-band notification that bypasses the normal data stream. For example, remote login protocols use out-of-band notification to signal the remote site in cases when a program misbehaves and must be aborted. The signal to abort must be processed even if the program has stopped consuming data in the normal stream. Thus, it cannot be sent as part of the normal data stream.



16.3 Urgent Data

TCP uses the term urgent data to refer to out-of-hand notification. When sending an out-of-band notification, the sender sets the urgent data bit (TCPF_URG) in the code field of the segment header, places the urgent data in the segment and assigns the urgent data pointer field an offset in the data area of the segment at which the urgent data ends. Because the protocol standard provides only one pointer in the header, the segment does not contain an explicit pointer to the beginning of urgent data.

When a segment carrying urgent data arrives, the receiver must notify the receiving application immediately: it cannot delay to wait for the application to process normal data that may be waiting. To do so, TCP places the application in urgent mode, which informs the application that urgent data exists. After it receives notification that urgent data exists, the application reads from the connection until it reaches the end of the urgent data. Finally, TCP informs the application that the end of urgent data has occurred.

16.4 Interpreting The Standard

The specification of urgent data is among the least understood and least documented parts of TCP. The original standard failed to provide complete answers for several questions. First, how can a receiver know where urgent data begins? Second, how can TCP inform an application that urgent data has arrived? Third, how does TCP inform an application when all urgent data has been processed? Fourth, what happens if multiple segments carrying urgent data arrive out of order? Fifth, does the urgent pointer point to the last octet of urgent data or to one location beyond the end of it^①? More to the point, the standard assumes a message-passing interface, and fails to describe TCP semantics in the open-read-write-close paradigm most implementations use. Although subsequent revisions have only provided partial solutions, a consensus has emerged about the interpretation of urgent data. We will examine the consensus view after considering one of the alternative views.

16.4.1 The Out-Of-Band Data Interpretation

Implementors have taken two views of urgent data processing. The first, which has fallen out of favor, views urgent data as out-of-band data. In essence, any implementation that follows the out-of-band data view provides two independent streams of data: the normal stream and the urgent stream. When urgent data arrives, TCP notifies the application immediately. TCP then returns urgent data whenever the

^① While the current standard is quite explicit about this (the pointer specifies the last location of urgent data), most extant implementations including those derived from 4BSD UNIX, have chosen to interpret the pointer as pointing one location beyond urgent data.



application reads. When the application consumes all urgent data, TCP notifies the application and returns normal data.

To understand the subtleties needed for out-of-band data processing, consider the conceptual diagram in Figure 16.1.

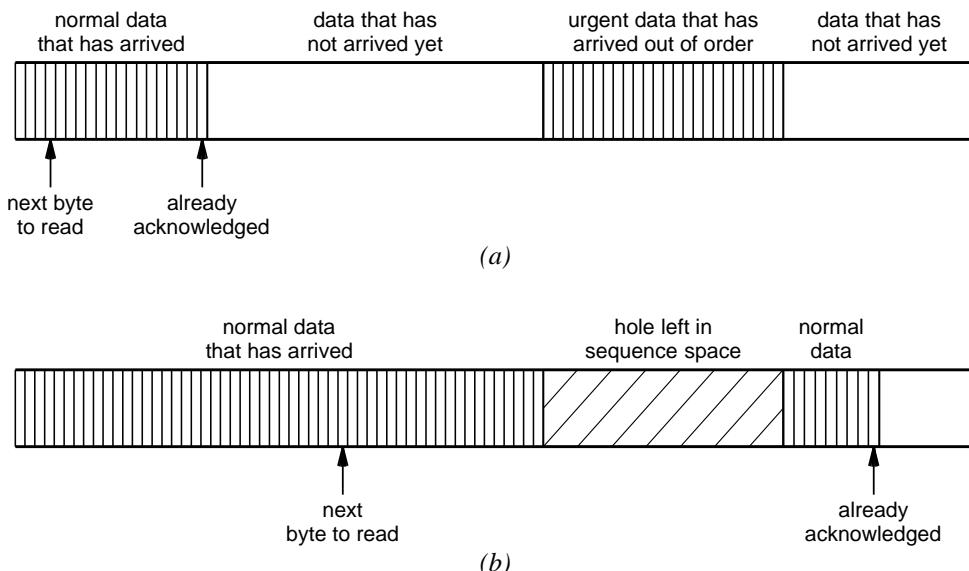


Figure 16.1 (a) The TCP sequence space when urgent data arrives before all normal data, and (b) the same sequence space after urgent data has been processed and additional normal data has arrived. The space occupied by urgent data forms a hole in the space.

Urgent data occupies part of the sequence space used for normal data transfer. In most cases, urgent data arrives in order with other data, so the receiver has already received and acknowledged the sequence up to the start of the urgent message. However, as Figure 16.1a shows, because the underlying IP layer does not guarantee to deliver segments in order, a segment carrying urgent data can arrive earlier than some of the segments carrying normal data for lower values in the sequence space. In fact, because urgent messages are usually short, and because some gateways give short datagrams priority, out-of-order delivery is quite common.

Figure 16.1b shows a consequence of the out-of-band-data interpretation. Urgent data, which occupies part of the sequence space, can be delivered out-of-order. When the urgent data arrives, TCP immediately passes it to the application program that is reading from the connection. After processing the urgent data, the application returns to normal data processing and continues to acquire octets from the place it was reading when the urgent data arrived. As a consequence, a "hole" remains in the sequence space at those locations where urgent data resided. When the application reaches the hole, TCP must skip over it as if those locations did not exist,



Implementations that use the out-of-band data interpretation need auxiliary data structures. For example, TCP needs to maintain a list of holes left in the address space when a program reads urgent data before normal data. In addition, such implementations must contend with a fundamental problem: urgent data can reside at locations in the sequence space beyond the current window^①. For these reasons, the out-of-band-data interpretation has become unpopular.

16.4.2 The Data Marie Interpretation

Our example code uses an alternative interpretation of urgent data that has become widely accepted. The alternative is known as the data mark interpretation. Implementations that use the data mark interpretation do not deliver data out-of-band. Instead, they merely treat the urgent data pointer in a segment as marking a location in the data stream. When a segment arrives carrying urgent data, the receiving TCP records the location in the data stream, and notifies the application that the stream contains urgent data. The application must read data from the stream until it reaches the urgent data mark. TCP then informs the application that all urgent data has been read.

The data-mark interpretation of urgent data is much simpler to implement than the out-of-band data interpretation. To provide a data-mark interpretation, TCP only needs to remember a location in the sequence space for which an urgent pointer has been received. It does not need to divide the data into two streams, nor does it need to record the position of holes that remain after urgent data has been consumed,

In essence the data-mark interpretation makes the application responsible for handling urgent data and for separating it from normal data. Although TCP notifies an application that urgent data has arrived somewhere ahead in the data stream, TCP does not deliver the urgent data to the application quickly, nor does it identify which data in the stream arrived in segments with the urgent data bit set.

The data-mark interpretation works best for applications that can read and discard normal data whenever urgent data arrives. For example, a TELNET server that receives an abort character as urgent data can read and discard keystrokes up to the abort because an abort will terminate the program that was reading data. However, an application that needs to process normal data in the stream after it receives and handles urgent data, must be designed to store normal data it encounters while searching for urgent data. After the application reaches the end of the urgent data, it must go back and handle the normal data that it stored.

^① It can be particularly important for a receiver to process urgent data while advertising a zero window because urgent data is used to abort an application that has stopped reading.



16.5 Configuration For Berkeley Urgent Pointer Interpretation

To allow our code to follow either the standard interpretation or the Berkeley UNIX interpretation of the urgent pointer (i.e., whether the urgent pointer points to the last octet of urgent data or to one octet beyond it), the code contains a configuration constant (BSDURG). When defined, symbolic constant BSDURG makes TCP interpret the pointer as the position ore beyond the last octet of urgent data.

BSDURG must be defined when compiling the code, and the definition applies to the entire system. Thus, the interpretation of the urgent pointer is compiled into the software, and cannot be set for each individual connection. We chose to make BSDURG part of the global configuration because TCP does not provide a way to negotiate the choice or to detect which interpretation to use.

16.6 Informing An Application

Our implementation uses a passive method to inform an application that urgent data has arrived. It waits for the application to issue a read request. Once urgent data arrives, the next call to read will place the process in urgent mode and return the special code TCPE_URGENTMODE. Subsequent calls to read return octets of data from the stream until all urgent data has been consumed. After the application consumes all the urgent data, the next call to read places the process in normal mode and returns the special code TCPE_NORMALMODE to inform the application that it has reached the end of urgent data. If no process ever calls read, urgent data will not be consumed. To summarize:

Our implementation handles urgent data passively. It waits for an application to read from the connection before informing the application that urgent data has arrived. The TCP software never takes action to handle the urgent data, nor does it create a process to interpret urgent messages.

A single call to read can request an arbitrarily large amount of data. Usually, large requests do not pose a problem. In the case of urgent data, however, an application can request data that exceeds an urgent data boundary. For example, consider the following sequence of events:

1. The receive buffer for a given connection contains ten octets of normal data that has not been read; the sequence space numbers are 1 through 10.
2. A segment arrives that contains an octet of urgent data at sequence 11.
3. A segment arrives that contains ten additional octets of normal data with sequence space 12 through 21.
4. The application calls read and TCP returns code TCPE_URGENTMODE to



inform the application that urgent data has arrived.

5. The application calls read and specifies a buffer large enough to hold all twenty one octets.

The standard specifies that TCP should only return the urgent data to the application before switching back to normal mode. In the example, TCP should only return the first 10 octets of data. To insure that an application does not read past the end of urgent data in a single call to read, TCP places an artificial limit on the data read:

Although an application may request a large amount of data, TCP does not allow a single call to read to exceed the end of urgent data.

16.6.1 Multiple Concurrent Application Programs

Although a concurrent processing system allows multiple processes to read from a single connection, the TCP standard does not specify the semantics of how they should process urgent data. We decided to implement the following scheme: for each process the system maintains an independent record of whether the process is in urgent mode. Whenever a process reads from a TCP connection, the system checks the status of the process and the status of the connection. If a process operating in normal mode reads from a connection that has only normal data, TCP transfers the data requested. Similarly, if a process operating in urgent mode reads from a connection for which urgent data has arrived, TCP returns data. However, if the mode of the process and status of the connection do not agree. TCP changes the process' status and returns one of the special return codes. A process operating in normal mode is placed in urgent mode and receives the return code TCPE_URGENTMODE if it reads from a connection for which urgent data has arrived. Similarly, a process operating in urgent mode is placed in normal mode and receives the return code TCPE_NORMALMODE if it reads from a connection that has no urgent data pending. However, all processes share a single copy of the data stream — octets read by one process will not be available to another.

In essence, our scheme for concurrent processing assumes that programmers take responsibility for coordinating processes that share a data stream. It also means that if multiple processes read from a single stream without coordinating, one process can receive the urgent data notification and another process can call read and consume the data. Thus, any application program that allows concurrent use of a TCP connection should be prepared to expect zero octets of urgent data after it receives a message telling it to enter urgent mode.

16.7 Reading Data From TCP

Now that we have seen all the pieces, it should be easy to understand how an



application obtains input data from TCP. Procedure tcpgetdata handles the task.

```
/* tcpgetdata.c - tcpgetdata */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpgetdata - copy data from a TCP receive buffer to a user buffer
 */
int tcpgetdata(ptcb, pch, len)
struct tcb *ptcb;
char *pch;
int len;
{
    tcpseq seq;
    unsigned cc;

    seq = ptcb->tcb_rnext - ptcb->tcb_rbcnt; /* start sequence */
    if (ptcb->tcb_flags & TCBF_RUPOK) {
        int nbc, ubc; /* normal & urgent byte counts */

        nbc = ptcb->tcb_rnext - ptcb->tcb_rupseq - 1;
        if (nbc >= 0) { /* urgent boundary in buffer */
            ubc = ptcb->tcb_rbcnt - nbc;
            if (len >= ubc) {
                len = ubc;
                ptcb->tcb_flags &= ~TCBF_RUPOK;
            }
        }
    }
    for (cc=0; ptcb->tcb_rbcnt && cc < len;) {
        *pch++ = ptcb->tcb_rcvbuf[ptcb->tcb_rbstart];
        --ptcb->tcb_rbcnt;
        if (++ptcb->tcb_rbstart >= ptcb->tcb_rbsize)
            ptcb->tcb_rbstart = 0;
        ++cc;
    }
    if (ptcb->tcb_rbcnt == 0)
        ptcb->tcb_flags &= ~TCBF_PUSH;
}
```



```
/*
 * open the receive window, if it's closed and we've made
 * enough space to fit a segment.
 */
if (SEQCMP(ptcb->tcb_cwin, ptcb->tcb_rnext) <= 0 &&
    tcprwindow(ptcb)) {
    ptcb->tcb_flags |= TCBF_NEEDOUT;
    tcpkick(ptcb);
}
return cc;
}
```

Tcpgetdata begins by calculating the sequence number of the first octet of data it will extract from the buffer. To do so, it subtracts the count of octets in the buffer from the sequence number of the highest octet received. Tcpgetdata then examines bit TCBF_RUPOK to determine whether a segment has arrived carrying urgent data. If so, tcpgetdata must guarantee that a single read operation does not cross the boundary between urgent data and normal data.

To limit data in a single read operation when urgent data is present, tcpgetdata first calculates the number of normal data octets in the buffer beyond the end of urgent data, and places the result in variable nbc. If the urgent data boundary lies in the buffer, tcpgetdata calculates the number of octets of urgent data, ubc, by subtracting nbc from the total count of octets in the buffer. Finally, tcpgetdata compares ubc to the length of data requested, and limits the length to ubc.

Once it has found the starting sequence number and computed a length of data to transfer, tcpgetdata enters its main loop. The loop iterates while there are characters in the buffer and the number of characters obtained is less than the number the caller requested. On each iteration, tcpgetdata extracts one octet of data from the input buffer and copies it to the caller's buffer.

Tcpgetdata also contains the code that handles window manipulation described in Chapter 15. Before it returns, tcpgetdata checks to see if tcb_rnext has moved past the currently advertised window and if tcprwindow returns a nonzero window advertisement. If so, it sets bit TCBF_NEEDOUT in the TCB, and calls tcpkick to generate a gratuitous acknowledgement. Acknowledgements created by tcpgetdata do not correspond to arriving data at all — TCP uses them merely to report to the sender that the window size has increased without waiting for a probe. Of course, if tcprwindow computes a zero-size window, tcpgetdata will not send an unnecessary acknowledgement.



16.8 Sending Urgent Data

The interpretation of urgent data affects how TCP sends urgent data as well as how it handles incoming urgent data. In particular, under the data-mark interpretation, the receiving TCP does not distinguish among multiple occurrences of urgent data. It merely tells the receiving application the highest point in the sequence space at which urgent data ends. As a consequence, a sender that uses the data-mark interpretation for urgent data only needs to store a single value that marks the location in the sequence space at which urgent data ends. The sending TCP does not need to separate outgoing data into urgent and normal streams, nor does it need to distinguish multiple occurrences of urgent data that the application writes. That is, if a sending application writes urgent data on a connection, writes normal data on the connection, and then writes additional urgent data on the connection, TCP only needs to report the end of the second set of urgent data to the receiving application. Thus, the sending TCP can update its notion of the urgent data location immediately, even if existing urgent data remains unacknowledged — the receiver will update its record of the urgent data pointer as soon as the new value arrives in a segment.

In addition to the consequences discussed above, two design decisions help simplify the code. First, our interface requires an application program to write urgent data in a single call. That is, the program must assemble an entire urgent message and pass it to TCP through a single procedure call. Second, an application must write urgent data in sequence. In particular, an application cannot skip ahead, set the urgent data mark, and then fill in the data up to that point. Thus, urgent data is placed in the TCP output buffer along with normal data.

These two design decisions help eliminate special cases and make it possible to use most of the code already designed for handling normal data output. Because TCP receives an entire urgent message in a single call, it does not need a special mechanism to collect pieces of the message before transmission. Because urgent data falls in the usual place in the sequence space and output buffer, it can be stored exactly like normal output data. Furthermore, because urgent data always occurs in the sequence space contiguous to normal data, the sender can handle retransmission as it does for normal data. However, one special case does occur when transmitting urgent data: according to the standard, the sender must force transmission even if the receiver has closed the window.

Procedure `tcpsend` transmits urgent data^①. When sending a segment, `tcpsend` examines bit `TCBF_SUPOK` in the TCB to determine whether the sender's urgent pointer field is valid. If so, field `tcb_supseq` contains the position in the sequence space of the highest octet of urgent data written to the connection. `TcpSend` must convert the sequence space position into a relative displacement. To calculate the relative

^① See page 256 for a listing of `tcpsend.c`.



displacement, `tcpSend` subtracts `tcp_seq`, the sequence value for the first octet in the segment. `TcpSend` then stores the result in the segment's urgent data field, and sets bit `TCPF_URG` in the code held.

16.9 TCP Push Function

Usually, TCP buffers both input and output. When processing output, it collects as much data as possible into each segment to improve throughput and lower overhead. When processing input, it collects incoming data from individual segments into the input buffer from which application programs extract it. Buffering improves overall efficiency. Inside a host or gateway, it lowers context switching and procedure call overhead. Outside of the computer system, buffering lowers network overhead by passing more data in each packet.

In general, buffering improves throughput by trading lower throughput for increased delay. Sometimes high delays caused by buffering create problems for communicating applications. To allow applications to bypass buffering, TCP supplies a push function. A sending application executes a push request to request that TCP send all existing data without delay. When TCP sends the data, it sets the push bit in the segment code field, so the receiver knows about the request as well. When data arrives in a segment that has the push bit set, TCP makes the data available to the receiving application without delay.

16.10 Interpreting Push With Out-Of-Order Delivery

In principle, honoring the push bit in an incoming segment should be simple: when a segment arrives with the push bit set, TCP should make the data available to the application immediately. However, because segments may arrive out of order, the notion of a push must be defined carefully. There are two extremes. First, a segment carrying the push bit may arrive before segments that carry data which appears earlier in the sequence space. Because TCP must deliver data to the application in sequence, it cannot deliver the data that arrived with the push bit set. The standard specifies that the push request refers to data in the buffer and not merely to a point in the sequence. Thus, TCP must remember that a push has been requested for specific data and switch to immediate delivery after intervening data arrives. Second, a segment with the push bit set may arrive later than segments carrying data with higher sequence numbers. That is, the segment carrying a push bit may fill in a gap in the sequence space. The protocol standard says that when a segment arrives with the push bit set, TCP must deliver all available data to the application program. Thus, the arrival of a segment with the push bit set may cause TCP to immediately deliver data beyond the data in the segment.

The idea underlying the definition of the push function is fundamental: TCP does



not observe record boundaries. A programmer cannot depend on TCP to deliver data exactly to the point of a push operation because the amount delivered depends on the buffer contents and the timing of arrivals. To summarize:

Push does not mark record boundaries. When a sender invokes the push operation, TCP will transmit and deliver all data in its buffers. Thus, if a segment S with the push bit set arrives late, TCP on the receiving side may deliver additional data to the application beyond the data carried in S.

However, even though push does not provide explicit record boundaries, the protocol does specify that a receiver must maintain some state information concerning the push. The rules are fairly simple: the receiver should immediately deliver all available data, or it should remember that a push has arrived, so it can begin immediate delivery as soon as it has received everything in the sequence up to and including the data in the segment that carries the push. TCP must remember how much data to deliver immediately so it can stop delivering in push mode when it reaches the appropriate point in the sequence space.

16.11 Implementation Of Push On Input

To record the arrival of a segment with the push bit set, our system uses two variables in the TCB. Field tcb_pushseq records the sequence number of the octet just past the end of the segment that arrived with the push bit set. Field tch_code contains a bit that indicates whether a push has arrived. The bit in the code field can be thought of as a mode bit. When TCBF_PUSH is not set, TCP uses normal delivery. When TCBF_PUSH has been set, the input has received a push and will deliver data immediately until the buffer has been cleared.

We have already examined the procedures that implement the push operation. Procedure `tcpdodat`^① processes the push bit when it extracts data from an incoming segment. Two cases arise. Either the arriving segment extends the currently acknowledged sequence (i.e., there are no gaps in the sequence space before the segment that contains the push), or the arriving segment has come out of order. In both cases `tcpdodat` sets the bit TCBF_PUSH in the TCB code field to indicate that TCP should deliver data in push mode. If the new segment extends the available sequence space, the data should be passed to waiting application programs immediately, so `tcpdodat` calls `tcpwakeup` to awaken them. If the sequence space contains gaps, `tcpdodat` cannot pass data to application programs immediately. Instead, it delays push processing until later. To do so, it computes the sequence number of the first octet beyond the data in the

^① See page 23.2 for a listing of `tcpdodat.c`.



segment (by adding the length of data in the segment to the segment sequence number), and records the result in TCB field tcb_pushseq. It then turns off the push bit in the TCB to await segments that fill the gap.

Procedure tfcoalesce^① handles delayed push processing. Tfcoalesce reconstructs the incoming data stream by inserting each arriving segment in a list ordered by sequence number. As it inserts a segment, tfcoalesce checks to see if the sequence number of received data has reached the stored value of the push sequence. If so, all data up through the segment that contains a push must be present, so tfcoalesce turns on the push bit and resets the push sequence field.

Once enabled, push mode continues until the receiving application has emptied the buffer, which occurs when the count of octets in the buffer reaches zero. The code to turn off push mode can be found in procedure tcpgetdata, covered in section 16.7 of this chapter.

16.12 Summary

TCP supports an urgent message facility that allows communicating application programs to send out-of-band notification. Two alternative interpretations of urgent data exist, but the data-mark interpretation has become widely accepted. The data-mark interpretation treats an urgent data pointer as marking a location in the normal data stream.

When an application reads input from a connection on which urgent data has arrived, TCP places the application in urgent mode; the application remains in urgent mode until it consumes all data up to the maximum point in the data stream that contained urgent data. After an application consumes the last octet of urgent data, TCP places the application in normal mode. The data-mark interpretation makes each application program responsible for buffering normal data that the application encounters while searching for urgent data.

When transmitting urgent data, TCP stores urgent octets in the standard output buffer and uses the same acknowledgement mechanism it uses for normal data. However, TCP must send urgent data immediately, even if the receiver has closed its window.

The TCP push function allows a sender to specify that outgoing data should be transmitted and delivered without delay. Out-of-order delivery complicates push processing. If a segment arrives with the push bit set after segments that carry data for higher sequence values, the standard specifies TCP should push all available data. If a segment arrives with the push bit set before segments carrying data for lower sequence values, TCP records the sequence number of the segment that has the push bit set, and enables push when intervening data has arrived.

^① See page 235 for a listing of tfcoalesce.c.



16.13 FOR FURTHER STUDY

Postel [RFC 793] specifies urgent data processing and the push operation. Braden [RFC 1122] refines the specification

16.14 EXERCISES

1. Suppose a sender incorrectly transmits both urgent data and normal data for the same locations in the sequence space. How will the sample code behave? What will the application receive?
2. Suppose two segments carrying urgent data arrive out of order, such that the first segment to arrive specifies sequence 1000, and the second segment to arrive specifies sequence 900. What will an application program receive if it reads from the connection after the first segment arrives, but before the second arrives? What will it receive if it waits until both have arrived before reading?
3. What happens if the sender sets the push sequence and the sends enough additional segments to always keep the receive buffer nonempty?
4. Suppose a receive buffer contains some data when a segment arrives with the push bit set, but there is a gap in the sequence space between the data already received and the new segment. Does the protocol standard forbid, recommend, or require TCP to deliver the existing data in push mode?
5. In the previous question, what happens in our implementation if `tcpdodat` sets the push bit in the TCB on when it assigns `tcb_pushseq`?
6. Suppose a segment carrying urgent data is lost. How does our implementation respond? What does protocol standard specify should happen?



17 Socket-Level Interface

17.1 Introduction

Each operating system defines the details of the interface between application programs and the protocol software. This chapter explores an interface from the Xinu operating system. Although the procedures and exact order of arguments are specific to the Xinu system, this interface contains the same basic structure found in other systems. Studying the interface procedures will help clarify the underlying protocol software and show how it interacts with application programs.

17.2 Interfacing Through A Device

Unlike UNIX systems, which incorporate services and devices into the file system, Xinu incorporates services and files into devices. It uses a device paradigm for all input and output operations, including communication between an application program and protocol software. To do so, the system provides a device abstraction, and defines a set of devices, most of which correspond to peripheral I/O hardware devices. For example, the system uses the device abstraction to provide a CONSOLE that application programs use to communicate with the console terminal. In addition to abstract devices that correspond directly to conventional hardware devices, Xinu provides many device definitions that permit applications to access system services. For example, Xinu provides devices, used to access individual files on local disks, remote files, and protocol software,

Whether it uses an abstract device that corresponds to physical hardware or to a service, an application program follows the open-read-write-close paradigm to use it. The program calls open with three arguments:

```
d = open(device, name, other);
```

The first argument is an integer device identifier that specifies the device to be used, and the second argument specifies the name of an object associated with that device. The meaning of the third argument depends on the device being opened. For many devices, it



specifies whether the object should be opened for reading, writing, or both. Open returns a device descriptor to be used for accessing the specified object.

Once an object has been opened and a device descriptor has been created for it, an application program uses functions read and write to transfer data from or to the object. Both take three arguments: a device descriptor for the object, a pointer to a buffer, and a transfer size (in bytes). Most devices interpret the transfer size as a maximum buffer length. A call to read specifies a maximum buffer length and returns the count of bytes reads.

```
len = read(device, buffer, buflen);
```

A call to write returns a status code.

```
status = write(device, buffer, baflen);
```

When an application finishes using an object, it calls close to terminate use. Because Xinu uses a global device descriptor space and allows processes to exchange device descriptors through shared memory, it cannot know how many processes are currently using a given device. Thus, unlike many systems, Xinu does not close open devices automatically when an application terminates^①. Instead, the application is responsible for calling close explicitly.

17.2.1 Single Byte I/O

Because some programs find it easier to transfer a single byte (character) at a time, the device system supports two additional functions. Function getc^② reads a single byte from a device and returns the character as the function value. Getc is most often used with terminal devices.

```
ch = getc(CONSOLOB);
```

Procedure putc takes a device descriptor and a character as an argument and writes the character to the specified device.

```
putc(CONSOLE, '\n'); /* move Console to next line */
```

17.2.2 Extensions For Non-Transfer Functions

In addition to the operations that open and close a device, and the operations that transfer data (read, write, getc, and putc), Xinu supports two functions used to control devices. Programs use the first function, seek, to position physical devices (especially disks) and abstract devices (especially files),

```
seek(device, position);
```

Argument device must be an integer device descriptor, while argument position is a long

^① Xinu does support the UNIX notions of standard input, standard output, and standard error, and it calls close on each of these three descriptors when a process terminates.

^② In Xinu getc and putc are system calls and not library routines.



integer that gives an offset at which the device should be positioned.

A second, more generic function, control handles all other nontransfer operations^①. Control takes a variable number of arguments.

```
control(device, FUNC, arg1, arg2, ...);
```

It always requires two arguments that specify the descriptor of a device to be controlled and a control function to be used on that device. Some control functions require additional arguments.

Control functions include any operation that does not specify data transfer, or operations that involve special handling (e.g., writing TCP urgent data). For example, control can be used to specify whether a terminal device echoes characters (applications usually turn off echo before prompting for a password or other secret information that should not be displayed). An application can also use control to change a device's mode of operation.

A programmer must consult the documentation to determine the set of control functions available for a given device and the exact meaning of each. If a specific control function requires additional arguments, the documentation specifies their types and meanings.

17.3 TCP Connections As Devices

The Xinu system uses two types of devices for TCP connections. It provides a TCP master device (used to create connections), and TCP slave devices (used to communicate once a connection has been established). Both clients and servers use the master device to create a connection.

When a client program wishes to make a TCP connection, it calls open on the TCP master device, specifying the remote destination with which it wishes to communicate. The call allocates a TCP slave device for the connection, initializes the internal data structures associated with it, and returns the slave device descriptor to the caller. The caller then uses the slave device descriptor with read or write to pass data across the connection. When the client finishes transferring data, it calls close on the slave device to shut down the connection and make the slave device available for reuse.

17.4 An Example TCP Client Program

It will be helpful to examine the code for an example client. When called,

^① Many of the Xinu TCP control functions parallel BSD UNIX system call. For example, while BSD UNIX supports an accept system call, Xinu supports a TCPC_ACCEPT control function that has the same effect.



procedure finger implements a finger command. The Finger function allows a user on one machine to find out which users are logged into another machine. To do so, the client opens a TCP connection to the remote server, sends one line of text, and then prints all data that comes back from the server. The line of text sent either gives the login name of a user to finger or consists of an empty line, which requests information on all users logged into the system.

```
/* finger.c - finger */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

#define FINGERPORT    79 /* TCP port finger server uses */

/*
 * finger - client procedure to print information about remote users
 */
finger (rhost, user, outdev)
char *rhost;      /* domain name of remote host to contact */
char *user;       /* name of specific user or null pointer */
int outdev;       /* device on which to print output */
{
    IPAddr   addr;      /*holds IP address of remote host */
    int dd;      /* device descriptor for connection */
    int cc;      /* count of characters read */
    char buff[2048];  /*buffer to hold finger information */

    /* convert domain name to IP address and place in dotted */
    /* decimal string "xxx.xxx.xxx.xxx:port"for call to open */

    name2ip(addr, rhost);
    sprintf (buf, "%u.%u.%u.%u:%d", BYTE(addr, 0), BYTE(addr,1),
             BYTE(addr, 2), BYTE(addr,3), FINGERPORT);
    /* open connection. write on line (so it works with any server */
    /* and then repeatedly read and print data that arrives over */
    /* the connection. */

    dd = open(TCP, buf, ANYLPORT);
    if (user)
```



```
        write(dd, user, strlen(user));
        write(dd, "\r\n", 2);
        while ((cc=read(dd, buf, sizeof(buf)))>0)
            write(outdev, buf, cc);
        close(dd);
        return(OK);
    }
```

In the code, the call to open takes three arguments: the TCP master device descriptor (constant TCP), a string that specifies the remote machine's IP address and port number (buf), and an integer that specifies the local protocol port number to use. Because the client can use an arbitrary local port number, it uses constant ANYLPORT.

Once finger opens a connection, it writes a single line of text, and then repeatedly reads and prints information that the server returns. The call to read will block until the server replies and a full buffer of data is available. Of course, the call will also return if all data has arrived and the server has closed the connection, even if the buffer is not full.

17.5 An Example TCP Server Program

Servers are more complex than clients because a server must be able to queue an incoming connection request while servicing an existing connection. To do so, a server calls open on the master device, specifying that it wants to create a TCP device in passive mode. The server uses two control calls to manipulate the passive device. First, the server calls control using function code TCPC_LISTENQ to set the length of the incoming request queue. Then the server enters a loop in which it calls control using function code TCPC_ACCEPT to accept the next incoming connection. The system allocates a slave device for each new connection, and returns the slave device descriptor.

Consider the example finger server shown below. A finger service provides information about users logged into the computer. The server begins by opening the TCP device to obtain a passive descriptor that it uses to accept incoming connections. Each time it accepts a connection, the server reads one line of input from the connection, and responds by sending information about users logged into the local machine. After it finishes sending, the server closes the connection. To keep the example code simple, our server merely returns fixed information found in the strings declared at the beginning of the program.

```
/* fingerd.c - fingerd */

#include <conf.h>
#include <kernel.h>
```



```
#include <network.h>

#define FINGERPORT    79          /* TCP port finger server use      */
#define BUFFERSIZ   120          /* size of input buffer      */
#define QUEUESIZE    5           /* size of conn. request queue  */

str1 = "Login      Name \n"; /* fake information to return      */
str2 = "dls David L. Stevens\n";
str3 = "comer    Douglas E. Comer\n";

/*-----
 * fingerd - server to provide information about users logged in
 *-----
 */

PROCESS fingerd()
{
    int dd;                  /* descrirptor for server      */
    int dd2;                 /* descrirptor for a connection */
    char request[BUFFERSIZ]; /* space to read request      */

    /* Open TCP in passive mode (no connection) for the server      */
    /* and set maximum queue size for incoming connections      */

    fd = open(TCP, ANYFPORT, FINGERPORT);
    control(fd, TCPC_LISTENQ, QUEUESIZE);

    /* Continually wait for next connection, read one line from it      */
    /* and respond to the request                                         */

    while (TRUE) {
        dd2 = control(dd, TCPC_ACCEPT);

        /* Fake version: read and then ignore what client sends */

        if (read(dd2, request, sizeof(request)) < 2) {
            close(dd2);
            continue;
        }
        write(dd2, str1, strlen(str1));
        write(dd2, str2, strlen(str2));
        write(dd2, str3, strlen(str3));
    }
}
```



```
        close(dd2);  
    }  
}
```

In the code, the control call using function code TCPC_ACCEPT returns the device descriptor of a slave device for a given connection (dd2). The server then uses descriptor dd2 to read and write information. Meanwhile, if new connection requests arrive, TCP will associate them with the original device and enqueue them. When the server finishes using a connection, it closes the slave device descriptor, making it available for use with new connections.

17.6 Implementation Of The TCP Master Device

To implement a device, a programmer must supply procedures that correspond to all of the high-level operations: open, close, read, write, putc, getc, seek, and control. For the TCP master device, only two of these operations are meaningful — the remainder merely return an error code if called. The meaningful operations consist of open, used to form a connection, and control, used to set default parameters that apply to all connections.

17.6.1 TCP Master Device Open Function

Tcpmopen implements the open operation for the master device.

```
/* tcpmopen.c - tcpmopen */  
  
#include <conf.h>  
#include <kernel.h>  
#include <network.h>  
#include <proc.h>  
  
/*-----  
 * tcpmopen - open a fresh TCP pseudo device and return descriptor  
 *-----  
 */  
int tcpmopen(pdev, fport, lport)  
struct devsw *pdev;  
char *fport;  
int lport;  
{  
    struct tcb *ptcb;
```



```
int      error;

ptcb = (struct tcb *)tcballoc();
if (ptcb == (struct tcb *)SYSERR)
    return SYSERR;
ptcb->tcb_error = 0;
proctab[currpid].ptcpumode = FALSE;
if (fport == ANYFPORT)
    return tcpserver(ptcb, lport);

if (tcpbind(ptcb, fport, lport) != OK || 
    tcpsync(ptcb) != OK) {
    ptcb->tcb_state = TCPS_FREE;
    sdelete(ptcb->tcb_mutex);
    return SYSERR;
}
if (error = tcpcon(ptcb))
    return error;
return ptcb->tcb_dvnum;
}
```

To open a new connection, `tcpmopen` calls `tcballoc`^① to allocate an unused TCB, and initializes the urgent mode Boolean in the process table entry of the calling process. It then examines the foreign port argument, `fport`, to see whether the caller requested a passive or active open. As the finger example shows, a server specifies the constant `ANYFPORT`, while a client specifies a particular foreign destination. If the foreign port argument indicates the caller is a server, `tcpmopen` passes control to procedure `tcpserver`. Otherwise, it calls `tcpbind` to record the foreign address, `tcpsync`^② to initialize fields of the TCB, and `tcpcon` to initiate an active connection.

17.6.2 Forming A Passive TCP Connection

`Tcpmopen` calls procedure `tcpserver` to handle the details of passive open. `Tcpserver` fills in a previously allocated TCB so it is ready to receive and queue connection requests, and returns the slave device descriptor corresponding to the TCB so a server can use it.

```
/* tcpserver.c - tcpserver */
```

^① See page 202 for a listing of `tcballoc.c`.

^② See page 238 for a listing of `tcpsync.c`.



```
#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *  tcpserver - do a TCP passive open
 */
int tcpserver(ptcb, lport)
struct    tcb  *ptcb;
int        lport;
{
    if (lport == ANYLPORT) {
        ptcb->tcb_state = TCPS_FREE;
        sdelete(ptcb->tcb_mutex);
        return SYSERR;
    }
    ptcb->tcb_type = TCPT_SERVER;
    ptcb->tcb_lport = lport;
    ptcb->tcb_state = TCPS_LISTEN;
    ptcb->tcb_lqsize = tcps_lqsize;
    ptcb->tcb_listenq = pcreate(ptcb->tcb_lqsize);
    ptcb->tcb_smss = 0;
    signal(ptcb->tcb_mutex);
    return ptcb->tcb_dvnum;
}
```

Tcpserver begins by examining the local port specification. While a passive connection can have an arbitrary, unspecified foreign port, it must have a specific local port (i.e., the well-known port at which the server operates). If the caller requests an arbitrary local port, tcpserver deallocates the mutual exclusion semaphore and returns SYSERR to its caller. Otherwise, tcpserver assigns fields in the allocated TCB to make it ready to accept connections, and returns the device descriptor used to access the newly allocated TCB.

17.6.3 Forming An Active TCP Connection

We said that tcpmopen calls three procedures when it needs to form an active connection: tcpbind, tcpsync, and tcpcon. Procedure tcpbind stores the foreign and local endpoint addresses in a TCB.

```
/* tcpbind.c - tcpbind */
```



```
#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 *  tcpbind - bind a TCP pseudo device to its addresses and port
 *-----
 */
int tcpbind(ptcb, fport, lport)
struct    tcb  *ptcb;
char      *fport;
int       lport;
{
    struct    route     *prt, *rtget();
    struct    tcb  *ptcb2;
    int        slot;

    if (dnparse(fport, ptcb->tcb_rip, &ptcb->tcb_rport) == SYSERR)
        return SYSERR;
    prt = rtget(ptcb->tcb_rip, RTF_LOCAL);
    if (prt == 0)
        return SYSERR;
    if (prt->rt_ifnum == NI_LOCAL)
        blkcopy(ptcb->tcb_lip, ptcb->tcb_rip, IP_ALEN);
    else
        blkcopy(ptcb->tcb_lip, nif[prt->rt_ifnum].ni_ip, IP_ALEN);
    ptcb->tcb_pni = &nif[prt->rt_ifnum];
    rtfree(prt);
    if (lport == ANYLPORT) {
        ptcb->tcb_lport = tcpnxtp(); /* pick one */
        return OK;
    }
    ptcb->tcb_lport = lport;
    for (slot=0, ptcb2=&tcbtab[0]; slot<Ntcp; ++slot, ++ptcb2) {
        if (ptcb == ptcb2 ||
            ptcb2->tcb_state == TCPS_FREE ||
            ptcb2->tcb_rport != ptcb2->tcb_rport ||
            ptcb2->tcb_lport != ptcb2->tcb_lport ||
            !blkequ(ptcb->tcb_rip, ptcb2->tcb_rip, IP_ALEN) ||
            !blkequ(ptcb->tcb_lip, ptcb2->tcb_lip, IP_ALEN))
```



```
        continue;

        return SYSERR;
    }

    return OK;
}
```

Tcpbind begins by calling procedure dnparse to parse the remote endpoint specification and break it into a remote protocol port number (field tcb_rport) and a remote IP address (field tcb_rip). Tcpbind checks to make sure a route exists to the specified remote machine. It also checks to see if the route leads to the pseudo interface (i.e., for the case where two local processes are using TCP to communicate), and uses the IP address from that interface if it does.

After handling the remote machine endpoint, tcpbind handles the local port. If the caller specified an arbitrary local port, tcpbind calls function tcpnntp to allocate an unused local port number. Otherwise, it must check the specified port to insure that no other connection is using the port. To do so, it fills in the specified endpoint in the TCB, and iterates through all TCBs, comparing the connection endpoint to the endpoint specified for the new one. TCP allows multiple connections to use the same local port number as long as the remote endpoints differ. Therefore, tcpbind must check the remote endpoint's IP address and protocol port in the comparison. If none are equal, tcpbind returns OK. If it finds that another connection already has the same endpoints allocated, tcpbind returns the error code SYSERR.

17.6.4 Allocating An Unused Local Port

Procedure tcpnntp allocates an unused local port and returns it to the caller

```
/* tcpnntp.c - tcpnntp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#define IPPORT_RESERVED          1024 /* from BSD */

/*-----
 *  tcpnntp  -  return the next available TCP local "port" number
 *-----
 */
short tcpnntp()
{
    static short     lastport=1;    /* #'s 1-1023 */
```



```
int      i, start;

wait(tcps_tmutex);

for (start=lastport++; start != lastport; ++lastport) {

    if (lastport == IPPORT_RESERVED)
        lastport = 1;

    for (i=0; i<Ntcp; ++i)
        if (tcbs[i].tcb_state != TCPS_FREE &&
            tcbs[i].tcb_lport == lastport)
            break;

    if (i == Ntcp)
        break;

}

if (lastport == start)
    panic("out of TCP ports");

signal(tcps_tmutex);

return lastport;
}
```

Tcpnntp uses static variable lastport to retain the integer index of the most recently assigned port across calls. Thus, when tcpnntp begins, lastport has the same value as it had during the previous call.

Tcpnntp uses a simple algorithm. It iterates through all possible local port numbers until it finds one not in use. On a given call, variable start records the starting value of variable lastport, and the iteration continues until tcpnntp has tried all possible values once.

Although the TCP standard does not restrict TCP ports to small values, our example code follows a convention used by BSD UNIX systems. Such systems reserve ports 1 through 1024 for privileged programs. Thus, allocating a port in that range guarantees that the client can communicate effectively, even if the server requires it to use a privileged port.

17.6.5 Completing An Active Connection

Once tcpbind has stored the connection endpoints in a TCB and verified that no other connection has them assigned, tcpmopen calls tcpsync^① to initialize most fields in the TCB, and then it calls procedure tcpcon to form a connection.

```
/* tcpcon.c - tcpcon */
```

^① See page 238 for a listing of tcpsync.c.



```
#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 *  tcpcon - initiate a connection
 *-----
 */

int tcpcon(ptcb)
struct tcb *ptcb;
{
    struct netif *pni = ptcb->tcb_pni;
    struct route *prt, *rtget();
    Bool local;
    int error, mss;

    prt = (struct route *)rtget(ptcb->tcb_rip, RTF_REMOTE);
    local = prt && prt->rt_metric == 0;
    rtfree(prt);
    if (local)
        mss = ptcb->tcb_pni->ni_mtu-IPMHLEN-TCPMHLEN;
    else
        mss = 536; /* RFC 1122 */
    ptcb->tcb_smss = mss; /* default */
    ptcb->tcb_rmss = ptcb->tcb_smss;
    ptcb->tcb_swindow = ptcb->tcb_smss; /* conservative */
    ptcb->tcb_cwnd = ptcb->tcb_smss; /* 1 segment */
    ptcb->tcb_ssthresh = 65535; /* IP Max window size */
    ptcb->tcb_rnnext = 0;
    ptcb->tcb_finseq = 0;
    ptcb->tcb_pushseq = 0;
    ptcb->tcb_flags = TCBF_NEEDOUT|TCBF_FIRSTSEND;
    ptcb->tcb_ostate = TCPO_IDLE;
    ptcb->tcb_state = TCPS_SYNSENT;
    tcpkick(ptcb);
    ptcb->tcb_listening = SYSERR;
    TcpActiveOpens++;
    signal(ptcb->tcb_mutex);
    wait(ptcb->tcb_ocsem);
    if (error = ptcb->tcb_error)
        tcbdealloc(ptcb);
```



```
    return error;           /* usually 0 */
}
```

Tcpcon initializes the maximum segment size, sequence space counters, and buffer pointers. It calls tcpkick to start the connection, and returns to its caller.

17.6.6 Control For The TCP Master Device

Procedure tcpcmcntl implements the control operation for the TCP master device.

```
/* tcpcmcntl.c - tcpcmcntl */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpcmcntl - control function for the master TCP pseudo-device
 */
int tcpcmcntl(pdev, func, arg)
struct devsw *pdev;
int func;
int arg;
{
    int rv;

    if (pdev != &devtab[TCP])
        return SYSERR;

    switch (func) {
    case TCPC_LISTENQ:
        tcps_lqsize = arg;
        rv = OK;
        break;
    default:
        rv = SYSERR;
    }
    return rv;
}
```

The control operation for the master TCP device allows the caller to set parameters



or control processing for all newly created slave devices. The current implementation of `tcpctl` provides only one possible control function — it allows the caller to set the default size of the listen queue for passive opens. After the default size has been set, all passive opens will begin with the new queue size.

17.7 Implementation Of A TCP Slave Device

Once the master device open operation has created a slave device and allocated a new TCB for a connection, the application uses the slave for input and output. Usually, the application invokes read and write operations on the slave device. It can also use `getc` or `putc` to transfer a single byte at a time, or control to control the individual device.

17.7.1 Input From A TCP Slave Device

Procedure `tcpread` implements the read operation for a TCP slave device.

```
/* tcpread.c - tcpread */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*
 * tcpread - read one buffer from a TCP pseudo-device
 */
tcpread(pdev, pch, len)
struct devsw *pdev;
char *pch;
int len;
{
    struct tcb *ptcb = (struct tcb *)pdev->dvioblk;
    int state = ptcb->tcb_state;
    int cc;

    if (state != TCPS_ESTABLISHED && state != TCPS_CLOSEWAIT)
        return SYSERR;
    retry:
    wait(ptcb->tcb_rsema);
    wait(ptcb->tcb_mutex);
```



```
if (ptcb->tcb_state == TCPS_FREE)
    return SYSERR; /* gone */
if (ptcb->tcb_error) {
    tcpwakeup(READERS, ptcb); /* propagate it */
    signal(ptcb->tcb_mutex);
    return ptcb->tcb_error;
}
if (ptcb->tcb_flags & TCBF_RUPOK) {
    if (!proctab[currid].ptcpumode) {
        proctab[currid].ptcpumode = TRUE;
        cc = TCPE_URGENTMODE;
    } else
        cc = tcpgetdata(ptcb, pch, len);
} else {
    if (proctab[currid].ptcpumode) {
        proctab[currid].ptcpumode = FALSE;
        cc = TCPE_NORMALMODE;
    } else if (len > ptcb->tcb_rbcnt &&
               ptcb->tcb_flags & TCBF_BUFFER &&
               (ptcb->tcb_flags & (TCBF_PUSH|TCBF_RDONE)) == 0) {
        signal(ptcb->tcb_mutex);
        goto retry;
    } else
        cc = tcpgetdata(ptcb, pch, len);
}
tcpwakeup(READERS, ptcb);
signal(ptcb->tcb_mutex);
return cc;
}
```

If the TCB is not in the ESTABLISHED or CLOSE-WAIT states, input is not permitted, so tcpread returns SYSERR. Before extracting data from the input buffer, tcpread waits on the read semaphore. When input arrives, tcpwakeup will signal the read semaphore, allowing the application to proceed. Tcpread then waits on the mutual exclusion semaphore to guarantee exclusive access to the TCB.

Once it has exclusive access, tcpread checks to see if the connection has disappeared (i.e., the TCB has moved to state TCPS_FREE), and returns SYSERR if it has. Next, tcpread checks to see if an error has occurred, and calls tcpwakeup to unblock the next process waiting to read. It then returns the error code to its caller.

Otherwise, tcpread is ready to extract data from the buffer. Tcpread compares the



urgent mode of the calling process to the urgent status of the connection. It examines bit TCBF_RUPOK to determine whether the connection contains urgent data, and field ptcpcumode in the process table to determine whether the calling process is currently operating in urgent mode. If the two values disagree, tcpread changes the process urgent mode status and returns a code to inform the caller.

If urgent data has arrived on the connection and the calling process is already in urgent mode, tcpread calls tcpgetdata to extract urgent data. If no urgent data is waiting, tcpread checks to see if sufficient data remains in the buffer to satisfy the request. Tcpread will normally continue to block until sufficient data has been received (i.e., until the buffer contains at least len bytes). However, in three special cases tcpread does not block. First, if data has arrived with the push bit set, tcpread delivers the data immediately. Second, if the sender has finished transmission and closed the connection, tcpread must deliver the final data or it will block forever (no additional data will arrive). Third, if the application program specifies unbuffered delivery (i.e., clears the TCBF_BUFFER bit in the TCB flags field), tcpread delivers the data that has arrived without waiting. Thus, if sufficient data is available to satisfy the request, or if one of the three special cases occurs, tcpread delivers data without waiting for more to arrive. It calls tcpgetdata to extract the data and copy it into the application program's buffer.

Once it has finished extracting data, tcpread calls tcpwakeup to allow the next waiting reader to determine whether additional data remains. It then signals the mutual exclusion semaphore and returns. Tcpread either returns an error code (which is less than zero), or the count of characters extracted as its function value. Thus, the application that called read knows exactly how many bytes of data were received.

17.7.2 Single Byte Input From A TCP Slave Device

Function tcpgetc implements the getc operation for a TCP slave device. It simply calls tcpread to read a single character and returns the result to its caller.

```
/* tcpgetc.c - tcpgetc */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*
 *-----*
 *  tcpgetc - read one character from a TCP pseudo-device
 *-----*
 */
int
tcpgetc(pdev)
```



```
struct devsw *pdev;
{
    char ch;
    int cc;

    cc = tcpread(pdev, &ch, 1);
    if (cc < 0)
        return SYSERR;
    else if (cc == 0)
        return EOF;
    /* else, valid data */
    return ch;
}
```

17.7.3 Output Through A TCP Slave Device

Procedure `tcpwrite` implements the write operation, and procedure `tcpputc` implement single character (byte) output. They both call a single underlying procedure, `tcpwr`.

```
/* tcpwrite.c - tcpwrite */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * tcpwrite - write one buffer from a TCP pseudo-device
 */
int tcpwrite(pdev, pch, len)
struct devsw *pdev;
char *pch;
int len;
{
    return tcpwr(pdev, pch, len, TWF_NORMAL);
}

/* tcpputc.c - tcpputc */

#include <conf.h>
#include <kernel.h>
```



```
#include <proc.h>
#include <network.h>

/*
 * tcpputc - write one character to a TCP pseudo-device
 */
int tcpputc(pdev, ch)
struct devsw *pdev;
char ch;
{
    return tcpwr(pdev, &ch, 1, TWF_NORMAL);
}

/* tcpwr.c - tcpwr */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*
 * tcpwr - write urgent and normal data to TCP buffers
 */
int tcpwr(pdev, pch, len, isurg)
struct devsw *pdev;
char *pch;
int len;
Bool isurg;
{
    struct tcb *ptcb = (struct tcb *)pdev->dvioblk;
    int state = ptcb->tcb_state;
    unsigned sboff;
    int tocopy;

    if (state != TCPS_ESTABLISHED && state != TCPS_CLOSEWAIT)
        return SYSERR;
    tocopy = tcpgetspace(ptcb, len); /* acquires tcb_mutex */
    if (tocopy <= 0)
        return tocopy;
```



```
sboff = (ptcb->tcb_sbstart+ptcb->tcb_sbcount) % ptcb->tcb_sbsize;
if (isurg) {
    ptcb->tcb_supseq = ptcb->tcb_snext + len - 1;
    ptcb->tcb_flags |= TCBF_SUPOK;
}
while (tocopy--) {
    ptcb->tcb_sndbuf[sboff] = *pch++;
    ++ptcb->tcb_sbcount;
    if (++sboff >= ptcb->tcb_sbsize)
        sboff = 0;
}
ptcb->tcb_flags |= TCBF_NEEDOUT;
tcpwakeup(WRITERS, ptcb);
signal(ptcb->tcb_mutex);

if (isurg || ptcb->tcb_snext == ptcb->tcb_sunap)
    tcpkick(ptcb);
return len;
}
```

If the connection is not in the ESTABLISHED state, output is prohibited, so tcpwr returns SYSERR. Otherwise, it calls tcpgetspace to allocate space in the output buffer. If tcpgetspace returns an error, tcpwr returns the error to its caller. Otherwise, tcpwr computes the position in the output buffer for the data it is about to write (by adding the count of characters already in the send buffer to the starting position of those characters, and wrapping the pointer around the end of the buffer). Tcpwr then copies the data into the buffer starting at that position,

Finally, after creating new output data to be sent, tcpwr sets bit TCBF_NEEDOUT to indicate that new data is awaiting output, and calls tcpwakeup to awaken other writers. If the caller requested urgent delivery or no output is in progress, tcpwr calls tcpkick to start output immediately (sender silly window avoidance).

17.7.4 Closing A TCP Connection

Once an application finishes using a TCP connection, it calls tcpclose on the slave device to shutdown the connection. Tcpclose also deallocates the slave device.

```
/* tcpclose.c - tcpclose */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```



```
/*-----  
 *  tcpclose - close a TCP connection  
 *-----  
 */  
  
int tcpclose(pdev)  
struct devsw *pdev;  
{  
    struct tcb *ptcb = (struct tcb *)pdev->dvioblk;  
    int error;  
  
    wait(ptcb->tcb_mutex);  
    switch (ptcb->tcb_state) {  
        case TCPS_LISTEN:  
        case TCPS_ESTABLISHED:  
        case TCPS_CLOSEWAIT:  
            break;  
        case TCPS_FREE:  
            return SYSERR;  
        default:  
            signal(ptcb->tcb_mutex);  
            return SYSERR;  
    }  
    if (ptcb->tcb_error || ptcb->tcb_state == TCPS_LISTEN)  
        return tcbdealloc(ptcb);  
    /* to get here, we must be in ESTABLISHED or CLOSE_WAIT */  
  
    TcpCurrEstab--;  
    ptcb->tcb_flags |= TCBF SNDFIN;  
    ptcb->tcb_slast = ptcb->tcb_sunap + ptcb->tcb_sbcount;  
    if (ptcb->tcb_state == TCPS_ESTABLISHED)  
        ptcb->tcb_state = TCPS_FINWAIT1;  
    else /* CLOSE_WAIT */  
        ptcb->tcb_state = TCPS_LASTACK;  
    ptcb->tcb_flags |= TCBF NEEDOUT;  
    tcpkick(ptcb);  
    signal(ptcb->tcb_mutex);  
    wait(ptcb->tcb_ocsem); /* wait for FIN to be ACKed */  
    error = ptcb->tcb_error;  
    if (ptcb->tcb_state == TCPS_LASTACK)  
        tcbdealloc(ptcb);
```



```
        return error;  
    }
```

One can only close a connection that is in the ESTABLISHED, CLOSE_WAIT, or LISTEN states. For other states, tcpclose returns SYSERR.

Closing a connection from the LISTEN state simply means deallocating the TCB (no connection is in progress). Similarly, if an application closes a connection after an error has occurred, tcpclose deallocates the TCB.

For the ESTABLISHED state, closing a connection means moving to state FIN-WAIT-1; for the CLOSE-WAIT state, it means moving to state LAST-ACK. In either case, tcpclose sets bit TCBF_SNDFIN to show that a FIN is needed, sets bit TCBF_NEEDOUT to cause the output process to send the FIN, and calls tcpkick to start the output process (which will send the FIN).

Finally, tcpclose blocks on the open-close semaphore associated with the TCB to await the acknowledgement of the FIN. When the ACK arrives, input procedure tcpfin1 or input procedure tcplastack will signal the open-close semaphore. When tcpclose resumes execution, it deallocates the TCB and returns to its caller. If the connection terminates abnormally, before the FIN has been acknowledged, tcpclose returns an appropriate error code.

17.7.5 Control Operations For A TCP Slave Device

TCP allows applications to control parameters for individual connections by providing a control function for slave devices. Procedure tcpcntl implements the control operation.

```
/* tcpcntl.c - tcpcntl */  
  
#include <conf.h>  
#include <kernel.h>  
#include <network.h>  
  
/*-----  
 * tcpcntl - control function for TCP pseudo-devices  
 *-----  
 */  
int tcpcntl(pdev, func, arg, arg2)  
struct devsw *pdev;  
int func;  
char *arg, *arg2;  
{  
    struct tcb *ptcb = (struct tcb *)pdev->dvioblk;  
    int rv;
```



```
if (ptcb == NULL || ptcb->tcb_state == TCPS_FREE)
    return SYSERR;

wait(ptcb->tcb_mutex);
if (ptcb->tcb_state == TCPS_FREE) /* verify no state change */
    return SYSERR;

switch (func) {
case TCPC_ACCEPT: if (ptcb->tcb_type != TCPT_SERVER) {
                    rv = SYSERR;
                    break;
                }
                signal(ptcb->tcb_mutex);
                return preceive(ptcb->tcb_listenq);
case TCPC_LISTENQ: rv = tcplq(ptcb, arg);
                    break;
case TCPC_STATUS:  rv = tcpstat(ptcb, arg);
                    break;
case TCPC_SOPT:
case TCPC_COPT:      rv = tcpuopt(ptcb, func, arg);
                    break;
case TCPC_SENDURG: /* 
                     * tcpwr acquires and releases tcb_mutex
                     * itself.
                     */
                     signal(ptcb->tcb_mutex);
                     return tcpwr(pdev, arg, arg2, TWF_URGENT);
default:
    rv = SYSERR;
}
signal(ptcb->tcb_mutex);
return rv;
}
```

When `tcpctl` begins, it verifies that the TCB is valid and acquires exclusive access to the TCB. `Tcpctl` then examines the `func` argument to see which control operation the caller requested.



17.7.6 Accepting Connections From A Passive Device

The example in section 17.5 illustrates how servers use the TCPC_ACCEPT function to accept an individual connection from a TCB in the LISTEN state. The implementation of TCPC_ACCEPT is straightforward, Tcpctl verifies that the TCB has been opened for use by a server, signals the mutual exclusion semaphore, and calls preceive to acquire the slave descriptor for the next incoming connection request. Tcpctl must signal the mutual exclusion semaphore to permit TCP software to process incoming SYN requests. However, it cannot proceed until a new connection has been established. Preceive blocks until the connection succeeds and the slave device can be used.

17.7.7 Changing The Size Of A Listen Queue

Function TCPC_LISTENQ allows the caller to change the size of the queue of incoming connections. Recall that an application can use the same function on the master device to change the default size that all servers receive when they issue a passive open. The difference here is that tcpctl only changes the size of the queue for a single slave device. Tcpctl calls procedure tcplq to make the change.

```
/* tcplq.c - tcplq */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mark.h>
#include <ports.h>

/*
 * tcplq - set the listen queue size for a TCP pseudo device
 */
int tcplq(ptcb, lqsize)
struct tcb *ptcb;
int lqsize;
{
    if (ptcb->tcb_state == TCPS_FREE)
        return SYSERR;
    ptcb->tcb_lqsize = lqsize;
    if (ptcb->tcb_type == TCPT_SERVER) {
        pdelete(ptcb->tcb_listenq, PTNODISP);
        ptcb->tcb_listenq = pcreate(ptcb->tcb_lqsize);
    }
}
```



```
    return OK;  
}
```

To change the size of a connection queue, tcplq replaces the queue size stored in the TCB with the new size, deletes the existing queue, and creates a new one.

17.7.8 Acquiring Statistics From A Slave Device

Often, network management software needs to extract and report about the status of individual connections. The control function TCPC_STATUS provides a mechanism for doing so. It calls procedure tcpstat to gather and report statistics about a connection to the caller.

```
/* tcpstat.c - tcpstat */  
  
#include <conf.h>  
#include <kernel.h>  
#include <network.h>  
  
/*-----  
 * tcpstat - return status information for a TCP pseudo device  
 *-----  
 */  
int tcpstat(ptcb, tcps)  
struct tcb *ptcb;  
struct tcpstat *tcps;  
{  
    tcps->ts_type = ptcb->tcb_type;  
    switch (ptcb->tcb_type) {  
        case TCPT_SERVER:  
            /* should increase to entire TCP MIB */  
            tcps->ts_connects = TcpActiveOpens;  
            tcps->ts_aborts = TcpEstabResets;  
            tcps->ts_retrans = TcpRetransSegs;  
            break;  
        case TCPT_CONNECTION:  
            blkcopy(tcps->ts_laddr, ptcb->tcb_lip, IP_ALEN);  
            tcps->ts_lport = ptcb->tcb_lport;  
            blkcopy(tcps->ts_faddr, ptcb->tcb_rip, IP_ALEN);  
            tcps->ts_fport = ptcb->tcb_rport;  
            tcps->ts_rwin = ptcb->tcb_rbsize - ptcb->tcb_rbcnt;  
            tcps->ts_swin = ptcb->tcb_swindow;
```



```
    tcps->ts_state = ptcb->tcb_state;
    tcps->ts_unacked = ptcb->tcb_suna;
    tcps->ts_prec = 0;
    break;

case TCPT_MASTER:
    break;
}

return OK;
}
```

Tcpstat assumes the caller has passed the address of a tcpstat structure into which it must place various statistics. File tcpstat.h contains the declaration of the structure as well as the definitions of various shorthand identifiers used to access individual fields.

```
/* tcpstat.h */

/* The union returned by the TCP STATUS control call */
struct tcpstat {
    int ts_type; /* which kind of TCP status? */
union {
    struct {
        long tsu_connects; /* # connections */
        long tsu_aborts; /* # aborts */
        long tsu_retrans; /* # retransmissions */
    } T_unt;
    struct {
        IPAddr tsu_laddr; /* local IP */
        short tsu_lport; /* local TCP port */
        IPAddr tsu_faddr; /* foreign IP */
        short tsu_fport; /* foreign TCP port */
        short tsu_rwin; /* receive window */
        short tsu_swin; /* peer's window */
        short tsu_state; /* TCP state */
        long tsu_unacked; /* bytes unacked */
        int tsu_prec; /* IP precedence */
    } T_unc;
    struct {
        long tsu_requests; /* # connect requests */
        long tsu_qmax; /* max queue length */
    } T_uns;
} T_un;
};
```



```
#define ts_connects    T_un.T_unt.tsu_connects
#define ts_aborts      T_un.T_unt.tsu_aborts
#define ts_retrans     T_un.T_unt.tsu_retrans

#define ts_laddr       T_un.T_unc.tsu_laddr
#define ts_lport       T_un.T_unc.tsu_lport
#define ts_faddr       T_un.T_unc.tsu_faddr
#define ts_fport       T_un.T_unc.tsu_fport
#define ts_rwin        T_un.T_unc.tsu_rwin
#define ts_swin        T_un.T_unc.tsu_swin
#define ts_state       T_un.T_unc.tsu_state
#define ts_unacked    T_un.T_unc.tsu_unacked
#define ts_prec        T_un.T_unc.tsu_prec

#define ts_requests   T_un.T_uns.tsu_requests
#define ts_qmax        T_un.T_uns.tsu_qmax
```

17.7.9 Setting Or Clearing TCP Options

The control operation also allows an application to clear or set option bits in the tcb_flags field. While most of these bits are intended for internal use, at least two of them can be pertinent to an application. First, bit TCBF_BUFFER determines whether read behaves synchronously or asynchronously. In particular, if the caller requests TCP to read n bytes of data, a synchronous call will block until n bytes of data arrive, while an asynchronous read will return as soon as any data arrives, even if it contains fewer than n bytes. (Of course, read always returns without waiting for n bytes if the sender shuts down the connection or specifies push.) Second, an application can set bit TCBF_DELACK to cause TCP to delay sending acknowledgements. Although delayed acknowledgements are not recommended for general use^①, some connections use them to reduce traffic (because acknowledgements will be piggybacked in outgoing data segments).

To clear or set options bits, the application program calls control with TCPC_COPT or TCPC_SOPT as the function argument and a bit mask as the third argument. When tcpcntl finds either of the clear or set requests, it passes the request to procedure tcpuopt.

```
/* tcpuopt.c - tcpuopt, ISUOPT */

#include <conf.h>
```

^① Unfortunately, the host requirements document, RFC 1122 does recommend delayed acknowledgements, even though many researchers agree that their use will confuse TCP round-trip estimation and can lead to poor performance except in a few unusual cases.



```
#include <kernel.h>
#include <network.h>

#define ISUOPT(flags)  (!!(flags & ~(TCBF_DELACK|TCBF_BUFFER)))

/*-----
 *  tcpuopt - set/clear TCP user option flags
 *-----
 */
int tcpuopt(ptcb, func, flags)
struct    tcb  *ptcb;
int       func;
int       flags;
{
    if (!ISUOPT(flags))
        return SYSERR;
    if (func == TCPC_SOPT)
        ptcb->tcb_flags |= flags;
    else
        ptcb->tcb_flags &= ~flags;
    return OK;
}
```

Tcpuopt uses the macro ISUOPT to check whether the user has specified any bits other than TCBF_DELACK or TCBF_BUFFER. If so, it rejects the request. Tcpuopt then examines the function code to determine whether it should set or clear the specified bits. The bits will remain set as long as the TCB remains allocated.

17.8 Initialization Of A Slave Device

At system startup, the operating system initializes each device, including TCP slave devices used for connections. Procedure tcpinit handles initialization of a slave device,

```
/* tcpinit.c - tcpinit */
```

```
#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mark.h>

static MARKER tcpmark;
```



```
/*
 *-----*
 *  tcpinit - initialize TCP slave pseudo device marking it free
 *-----*
 */

int tcpinit(pdev)
struct devsw *pdev;
{
    struct tcb *tcb;

    if (unmarked(tcpmark)) {
        mark(tcpmark);
        tcps_tmutex = screate(1);
        tcps_lqsize = 5; /* default listen Q size */
    }
    pdev->dvioblk = (char *) (tcb = &tcbtab[pdev->dvmminor]);
    tcb->tcb_dvnum = pdev->dvnum;
    tcb->tcb_state = TCPS_FREE;
    return OK;
}

#ifndef Ntcp
struct tcb tcbtab[Ntcp]; /* tcp device control blocks */
#endif
```

Initialization consists of assigning the constant TCPS_FREE to the state field in the corresponding TCB. Once the TCB has been initialized, it becomes available for allocation by procedure tcballoc.

17.9 Summary

TCP does not specify the exact details of the interface between protocol software and application programs. Instead, it allows the operating system to choose an interface. Many systems use the socket interface taken from BSD UNIX.

Our example interface uses the device abstraction along with the open-read-write-close paradigm. An application calls open on a TCP master device to allocate a device descriptor it can use for an individual connection. When a client opens the TCP master device, it receives a connected descriptor used for data transfer; when a server open the TCP master device, it receives a stub descriptor used only to accept incoming connections. The server then repeatedly issues a control call on the stub



descriptor to acquire descriptors for individual connections.

Once a connection has been established, applications use read, write, getc, or putc to transfer data. Finally, the application calls close to terminate use and make the descriptor available for new connections.

We examined the implementation of procedures that provide I/O operations on both the TCP master device as well as TCP slave devices.

17.10 FOR FURTHER STUDY

Comer [1988] presents an overview of the BSD UNIX socket interface, while Leffler, McKusick, Karels, and Quarterman [1989] discusses its implementation. Stevens [1990] examines how applications use various UNIX protocol interfaces. Comer [1984] presents details of the Xinu device structure, and shows how operations like read and write map into underlying functions like tcpread and tcpwrite.

17.11 EXERCISES

1. The remote machine specification used by the TCP master device may seem awkward. State several reasons why it might have been chosen.
2. As an alternative to the design presented here, consider a design in which the master TCP device passes all control operations on to all currently active slave devices. For example, one can imagine an ABORT control function which, when applied to a TCP slave device aborts the connection for that slave and, when applied to the TCP master device, aborts all connections in progress on all slave devices. What are the advantages and disadvantages of such a scheme?
3. Examine the code for mutual exclusion at the beginning of tcpcntl carefully. Can the call to wait ever result in an error? Will it affect the outcome in any way? Explain.
4. What happens if one or more connection requests have arrived before a server uses control to change the size of the listen queue for its slave device?
5. List additional control functions that an application program might find useful.
6. Our code uses control to implement all nontransfer operations. Compare this approach to one that uses separate system calls for each special operation (e.g., accept, listen, etc.). What are the advantages and disadvantages of each?
7. Large buffers generally imply higher throughput, but sometimes large buffers do not. Suppose multiple application programs continually attempt to read from a single TCP connection with the TCBF_BUFFER bit set. Argue that



whichever of them uses the largest buffer will receive the least service.

8. In the question above, under what circumstances is it possible that if many applications attempt to read from a single connection one of them will not receive any data at all, while the others continue to receive data?



18 *RIP: Active Route Propagation And Passive Acquisition*

18.1 Introduction

Earlier chapters showed the structure of an IP routing table and the procedures IP uses when forwarding datagrams toward their destinations. Hosts or gateways, interconnected with simple internet topologies, initialize their IP routing tables at system startup by inserting a few entries that never change. In most environments, however, gateways propagate routing information dynamically to provide automated computation of minimal paths and automatic recovery from temporary network or gateway failures. Hosts and gateways that receive the propagated information update their routing table entries accordingly.

This chapter examines the Routing Information Protocol (RIP), one of the most popular protocols used to propagate routing information among gateways and hosts. Although RIP seems simple on the surface, we will see that there are many subtle rules that govern which routes to advertise and when to advertise them. The rules help prevent routing loops, and make route propagation both faster and more reliable.

Route propagation is among the most complex tasks in an internet. Small deviations from the standard or the omission of a few heuristics can lead to severe problems, such as nonoptimal routes or instabilities. Furthermore, while most errors in protocol software affect only the machine that runs the incorrect software, poorly written route propagation software is especially dangerous, because it can affect all machines on its internet. Thus, like all routing protocols, a correct implementation of RIP requires careful attention to detail.



18.2 Active And Passive Mode Participants

TCP/IP internets follow the premise that gateways know correct routes because they exchange routing information with other gateways. By contrast, hosts only learn routes from gateways: their routing information, may not be complete or authoritative. Hence, hosts are forbidden from informing other machines about routes. In summary:

Gateways engage in active propagation of routing information, while hosts acquire routing information passively and never propagate it.

The RIP protocol honors this rule by providing two basic modes of operation. Hosts use RIP in passive mode, to passively listen for RIP messages sent by gateways, extract routing information from them, and update their own routing tables. Passive RIP does not propagate information from the local routing table. Gateways use RIP in active mode. Active participants listen for RIP messages from other gateways, install new routes in their routing tables, and send messages that contain the updated routing table entries. Thus, active participants engage in two activities (transmission and reception), while passive participants engage in only one (reception). The next sections focus on active participants.

18.3 Basic RIP Algorithm And Cost Metric

RIP uses a vector-distance algorithm to propagate routes and local network broadcast to deliver messages. Each gateway periodically broadcasts routes from its current IP routing table on all network interfaces. Like other vector-distance protocols, the RIP message contains pairs that consist of a destination network and the distance to that network.

When a RIP update message arrives, the receiving machine examines each entry and compares the entry to its current route for the same destination, D. The receiver uses a triangle inequality to test whether the advertised route to D is superior to the existing route. That is, when examining an entry received from gateway G, the receiver asks whether the cost of going to G, plus the cost of going from G to D, is less than the current cost of going to D. Expressed in mathematical terms, the receiver R asks if

$$\text{cost}(R, G) + \text{cost}(G, D) < \text{cost}(R, D)$$

where $\text{cost}(i, j)$ denotes the cost of the least expensive path from i to j. The receiver only updates its routing table entry for a destination if the cost of sending traffic through gateway G is less than the current cost. When changing a route, the receiver assigns it a cost equal to

$$\text{cost}(R, G) + \text{cost}(G, D)$$

Because the cost of reaching a neighboring gateway is 1, the new cost becomes



$$\text{cost}(R, D) = \text{cost}(G, D) + 1/$$

Although the above description seems simple, one final detail complicates it. Suppose R's current route to destination D goes through gateway G. When a new update arrives from G, R must change its cost for the route independent of whether G reports a decrease or an increase in cost. Thus, the final version of the algorithm becomes:

When a RIP update arrives with metric M for destination D from gateway G, compare it to the current route. If no route exists, create one with next hop equal to G and cost equal to M+1. If the current route specifies G as the next hop, set the cost in the route to M+1. Otherwise, if the cost of the current route is greater than M+1, set the cost to M+1, and set the next hop to G.

18.4 Instabilities And Solutions

18.4.1 Count To Infinity

Most vector-distance algorithms share a common problem because they allow temporary routing loops. A routing loop occurs for destination D when two or more gateways become locked in a circular sequence, such that each gateway thinks the optimal path to destination D uses the next gateway in the sequence. The simplest routing loops involve two gateways that each think the other is the best next hop along a route to a given destination,

Gateways using RIP cannot easily detect ranting loops. When a routing loop does occur for destination D, the RIP protocol causes the gateways involved to slowly increment their metrics one at a rime. They will continue until the metric reaches infinity, a value so large that the routing software interprets it to mean, "no route exists for this destination." To help limit the damage routing loops cause, RIP defines infinity to be a small number. We can summarize:

To limit the time a routing loop can persist, RIP defines infinity to be 16. When a routing metric reaches that value, RIP interprets it to mean "no route exists."

18.4.2 Gateway Crashes And Route Timeout

RIP requires all participating gateways and hosts to apply a timeout to all routes. A route must expire when its timeout occurs. To understand timeout, consider what happens when a gateway G, that has been actively participating in RIP, crashes. Neighboring gateways have received update messages from G, and have installed routes



that use C as the next hop. When G crashes, neighbors have no way of knowing that the routes using it as a next hop have become invalid. In essence, the cost for the route has become infinity, but the neighbors have no way of learning about the change because the gateway responsible for broadcasting the routing updates has crashed. Thus, gateways that receive information from RIP take responsibility for insuring it remains correct.

When installing or changing a route, associate a timer with it. If no information arrives to revalidate the route before the timer expires, declare the route to be invalid.

18.4.3 Split Horizon

One of the most common causes of routing loops arises if gateways advertise all routing information on all network interfaces. To understand the problem, consider three gateways, A, B, and C, attached to the same Ethernet. Suppose gateway A has a cost 1 path to destination D, and has advertised it by broadcasting a RIP update packet. Both B and C have received the update and have installed routes for destination D with cost 2. If they advertise their routes, no problem occurs because their routes are more expensive than the route A advertises.

Now suppose that gateway A crashes. If B or C continue to advertise their cost 2 route to D long enough, machines on the network will eventually time out the route that A advertised, and will adopt a cost 2 route. In fact, as soon as the route A advertised expires, either B or C will adopt the route the other one advertises, creating a temporary routing loop.

To avoid routing loops, RIP uses a technique known as split horizon. The rule is simple.

When sending a RIP update over a particular network interface, never include routing information acquired over that interface.

One way to look at this rule is from the viewpoint of the routing that occurs within a gateway. If a gateway G learned a route to destination D through the interface for network N, then G's route must specify a next hop that lies on network N. That is, G will route all datagrams headed to D to a gateway on N. Now suppose that G includes its route to destination D when broadcasting a RIP update on network N. If a gateway or host on network N has no current route to D (perhaps because an error has occurred), it will install the advertised route and send all datagrams destined for D to G. If a datagram does arrive at G destined for D, G will forward the datagram to the next hop, which lies on network N. Thus, G will forward datagrams that arrive over network N back out over the same network on which they arrived. Split horizon solves the problem by avoiding



advertisements that could cause it.

Whenever a gateway changes the metric for a route, it must send an update message to all its neighbor immediately without waiting for the usual periodic update cycle.

Triggered updates help RIP break routing loops that involve more than two gateways because it causes RIP to propagate infinite cost routes without waiting for periodic broadcasts. To understand how triggered updates help, consider a set of n gateways ($n > 2$) which have entered a routing loop. The first gateway advertises its route to the second, which advertises its route to the third, and so on, until the final gateway advertises its route back to the first. In such a situation, the metrics will increase by n after updates pass around the cycle once. Thus, counting to infinity can take an extremely long time, even when infinity is defined to be small. Split horizon alone does not break loops that involve multiple gateways, because for each pair of gateways, the route advertisements only propagate in one direction and never directly back.

Triggered updates improve robustness by propagating routes quickly. In particular, when a gateway G loses its connection to a given destination D, it sends a triggered update to propagate a route for D with cost infinity. Any neighboring gateway that depends on G to reach destination D will receive the update and change its cost for D to infinity. The change in neighbors of G triggers another round of updates sent by those neighbors, and so on. The triggered updates result in a cascade of updates. In fact, if the triggered updates occur quickly enough^①, they completely prevent routing loops.

18.4.4 Poison Reverse

In general, vector-distance protocols like RIP allow routing loops to persist because they do not propagate information about route loss quickly. A heuristic known as poison reverse (or split horizon with poison reverse) helps solve the problem. It modifies the split horizon technique. Instead of avoiding propagation of routes out over the network from which they arrived, poison reverse uses the updates to carry negative information.

When sending a RIP update over a particular network interface, include all routes, but set the metric to infinity for those routes acquired over that interface.

Poison reverse will break routing loops quickly. If two machines each have a route for destination D that points to the other machine, arranging to have them send an update with the cost set to infinity will break the loop as soon as one machine sends its update.

Of course, using split horizon with poison reverse has a disadvantage: it increases the size of update messages (and therefore uses more network bandwidth). For most gateways, however, the increased update message size does not cause problems.

^① Quickly enough means that the cascade must complete before any normal updates occur.



18.4.5 Route Timeout With Poison Reverse

We said that RIP requires gateways to place a timeout on each route and to invalidate the route when the timeout occurs. The most obvious implementation merely removes a route from the routing table when its timer expires. However, when RIP uses poison reverse, it cannot discard routes after they become invalid. Instead, it must keep a record that the route existed and now has cost infinity.

RIP only needs to retain expired routes until outgoing messages propagate the information to neighboring gateways. In principle, RIP only needs to retain an expired route through one update cycle. Because the underlying UDP and IP protocols can drop datagrams, RIP keeps a record of expired routes through four update cycles. After four cycles RIP assumes neighboring gateways have received at least one update that reports the route at cost infinity, so it deletes the route.

18.4.6 Triggered Updates

One additional technique helps make RIP more robust in the presence of large routing loops. The technique, known as triggered updates, employs rapid updates to speed the process of convergence after a change.

18.4.7 Randomization To Prevent Broadcast Storms

The protocol standard specifies that RIP must randomize the transmission of triggered updates. That is:

Whenever a gateway changes the metric for a route, it must send an update messages to all its neighbors after a short random delay, but without waiting for the usual periodic update.

To understand how a random delay helps, remember that RIP uses hardware broadcast to deliver update messages, and imagine multiple gateways that share an Ethernet. Think of poison reverse. Whenever one of the gateways sends an update for some destination D, all other gateways on the Ethernet install the change, which triggers updates (including a poison reverse update for the Ethernet over which the information arrived). Thus, all gateways will attempt to broadcast their triggered update simultaneously. A broadcast storm results. In fact, if the site has chosen to purchase all of its gateways from the same vendor, they will all use the same hardware and run the same software, making them generate a triggered response at exactly the same time. To eliminate simultaneous transmission, RIP specifies that a gateway must wait for a small, random delay before sending triggered updates.



18.5 Message Types

RIP provides two basic message types. It allows a client to send a request message that asks about specific routes, and it provides a response message used either to answer a request or to advertise routes periodically. The protocol standard defines additional message types but they are obsolete.

In general, few clients poll for updates. Instead, most implementations rely on gateways to generate a periodic update message. Technically, the periodic broadcast message is called a gratuitous response because it uses the response message type even though no request message caused it to occur.

18.6 Protocol Characterization

RIP can be characterized by the following:

- Operates at application level
- Uses UDP for all transport
- Limits message size
- Is connectionless
- Achieves reliability with k-out-of-n algorithm
- Specifies fixed timeouts
- Uses a well-known protocol port number
- Uses uniform message Format

The next paragraphs discuss each of these.

Level and transport. RIP operates at the application level and uses UDP to carry requests and responses. The chief advantage of using UDP is that it allows clients to send requests to remote gateways (i.e., gateways that do not connect to the same network as the client). The chief disadvantage of using UDP is that RIP must provide its own reliability.

Small message size. RIP limits individual update messages to 512 octets of data. If a gateway needs to propagate more information than will fit in a single message, it sends multiple messages, but does not number or otherwise identify them as a set. Thus, if packet loss occurs, it is possible to receive partial updates without knowing about the loss.

Connectionless communication. RIP maintains no notion of connections, nor does it provide acknowledgements.

K-out-of-N Reliability. RIP achieves reliability using a k-out-of-n approach. Each receiver watches for periodic broadcasts and declares routing information to be invalid unless it receives at least 1 out of every 6 updates. A receiver never polls or verifies that



a route has expired — it merely waits passively for new updates.

Fixed timeouts. RIP uses fixed values for all timeouts. It broadcasts update messages once every 30 seconds, waits 180 seconds after receiving an update before declaring that a route has become invalid, and retains the route with cost infinity an additional 120 seconds (for poison reverse). These timeouts are independent of the underlying physical networks using RIP or the round-trip time, making it impractical to use RIP on networks that exhibit extremely high delay or high packet loss.

Well-Known Protocol Port. RIP uses protocol port 520 for most communication. Clients always send requests to 520, and servers always answer from 520. Furthermore, all periodic update broadcasts originate from 520; clients can broadcast a request from port 520 to ask all gateways on the local network to generate a response. Only clients ever use a port other than 520, and they only use such a port when sending a request to a specific gateway; a server sends its response from port 520 to the port from which the request originated.

Uniform message format. RIP uses a single message format for all communication. In requests, the message contains destination addresses for which the client wants routing information, and the metric is not used. In responses, the destination addresses and metrics contain information from the sender's routing table.

18.7 Implementation Of RIP

18.7.1 The Two Styles Of Implementation

Broadly speaking, a designer must choose between two implementation styles. In the first style, RIP keeps its own routing database separate from the routing table that IP uses to forward datagrams. When new information arrives, RIP updates its routing database and then installs changes in the IP routing table. In the second style, RIP uses the same routing table as IP. Because all routes reside in the IP table regardless of their origin, routing table entries may need extra fields that RIP uses.

The advantage of using a separate table lies in separation of functionality. RIP can operate completely independent of IP and only needs to coordinate when it updates the IP routing table. Keeping RIP data separate from the IP routing database also allows the designer to modify RIP data structures, or replace the programs that implement RIP without changing IP.

The chief disadvantage of keeping RIP and IP information separate arises because the separation makes it difficult for RIP to learn about existing routes or install new routes. For example, if the gateway obtains a route through other protocols or if the manager installs a route manually, RIP will not know about the change. Thus, routing updates sent by RIP may not accurately reflect the current routing table. Similarly, RIP may continue to advertise routes after they have been deleted unless the manager also



notifies RIP about every change.

Although we have chosen to use the second technique, storing RIP information in the IP routing table has a drawback related to timer management. Because the unified table stores all routing information, it must store the information RIP needs for poison reverse. In particular, it must contain entries for routes that have expired but which are kept for purposes of generating poison reverse updates, even though such routes must not be used when forwarding datagrams. Thus, IP must not use entries in the table that have cost infinity.

18.7.2 Declarations

File rip.h contains declarations of the rip message format and constants used throughout the code.

```
/* rip.h */

#define RIPHSIZE 4      /* size of the header in octets          */
#define RIP_VERSION 1    /* RIP version number                   */
#define AF_INET     2    /* Address Family for IP               */

/* RIP commands codes: */

#define RIP_REQUEST 1    /* message requests routes            */
#define RIP_RESPONSE 2   /* message contains route advertisement */

/* one RIP route entry */

struct riprt {
    short rr_family;    /* 4BSD Address Family                */
    short rr_mbz;       /* must be zero                      */
    char rr_addr[12];   /* the part we use for IP: (0-3)    */
    int rr_metric;      /* distance (hop count) metric      */
};

#define MAXRIPROUTES 25  /* MAX routes per packet             */
#define RIP_INFINITY 16   /* dest is unreachable              */

#define RIPINT         300 /* interval to send (1/10 secs)      */
#define RIPDELTA 50      /* stray RIPINT +/- this (1/10 secs) */
#define RIPOUTMIN 50    /* min interval between updates (1/10 s)*

#define RIPRTTL        180 /* route ttl (secs)                 */
```



```
#define RIPZTIME 120 /* zombie route lifetime (secs) */  
  
/* RIP packet structure */  
struct rip {  
    char rip_cmd; /* RIP command */  
    char rip_vers; /* RIP_VERSION, above */  
    short rip_mbz; /* must be zero */  
    struct riprt rip_rts[MAXRIPROUTES];  
};  
  
/* RIP process definitions */  
  
extern int rip(), ripout(); /* processes to implement RIP */  
#define RIPISTK 49152 /* RIP process stack size */  
#define RIPOSTK 4096 /* RIP process stack size */  
#define RIPPRI 50 /* RIP process priority */  
#define RIPNAM "rip" /* RIP main process name */  
#define RIPONAM "ripout" /* RIP output process name */  
#define RIPARGC 0 /* Num. args for main RIP proc. */  
#define RIPOARGC 0 /* Num. args for RIP out. proc. */  
  
#define MAXNRIP 5 /* max # of packets per update */  
  
/* Per-interface RIP send data */  
struct rq {  
    Bool rq_active; /* TRUE => this data is active */  
    IPAddr rq_ip; /* destination IP address */  
    unsigned short rq_port; /* destination port number */  
/* what we've built already: */  
    struct ep *rq_pep[MAXNRIP]; /* packets to send */  
    int rq_len[MAXNRIP]; /* length of each */  
/* in-progress packet information */  
    int rq_cur; /* current working packet */  
    int rq_nrts; /* # routes this packet */  
    struct rip *rq_prip; /* current rip data */  
};  
  
extern int riplock; /* ripin/ripout synchronization */  
extern int rippid; /* PID for ripout() process */  
extern Bool dorip; /* TRUE => do active RIP */
```



Structure rip defines the RIP message format, while structure riprt defines a single route within the message. To maintain compatibility with BSD UNIX systems, the constant that defines the message type has been named AF_INET.

18.7.3 Conceptual Organization For Output

Because heuristics like the split horizon and poison reverse require the contents of an update message to vary depending on the interface, RIP cannot generate a single update message and send it on all interfaces. Instead, RIP simultaneously generates a separate copy of the update message for each network interface. When it adds a route to the update message, it applies rules like poison reverse to decide what it should add to each copy.

RIP uses structure rq to hold the contents of an individual copy of the update message^①. Because not all networks use RIP, the structure contains Boolean field rq_active that allows the manager to decide whether RIP should send updates on that interface. The structure seems complicated because RIP may need multiple datagrams to carry the update. It contains an array of pointers to packets (field rq_pcp), a corresponding array of packet lengths (field rq_len), an integer that tells how many packets RIP has added to the message so far (field rq_cur), how many routes RIP has added to the current packet (field rq_nrts), and a pointer to the current position in the packet where the next route belongs (field rq_prip).

18.8 The Principle RIP Process

Procedure rip performs two chores: it initializes the RIP data structures and it handles all incoming RIP packets.

```
/* ripin.c - rip */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <proc.h>

/*
 *  rip - do the RIP route exchange protocol
 */
PROCESS rip()
```

^① The output routines organize the individuals rq structures into array rqinfo, which has one entry per network interface.



```
{  
    struct    xgram      ripbuf;  
    struct    rip *prip;  
    int       fd, len;  
  
    fd = open(UDP, ANYFPORT, UP_RIP);  
    if (fd == SYSERR)  
        panic("rip: cannot open rip port");  
    riplock = screate(1);  
    if (gateway)  
        resume(create(ripout, RIPSTK, RIPPRI, RIPONAM, RIPOARGC));  
  
    while (TRUE) {  
        len = read(fd, &ripbuf, sizeof(ripbuf));  
        if (len == SYSERR)  
            continue;  
        prip = (struct rip *)ripbuf.xg_data;  
        if (ripcheck(prip, len) != OK)  
            continue;  
        switch (prip->rip_cmd) {  
        case RIP_RESPONSE:  
            if (ripbuf.xg_fport == UP_RIP)  
                riprecv(prip, len, ripbuf.xg_fip);  
            break;  
        case RIP_REQUEST:  
            if (gateway || ripbuf.xg_fport != UP_RIP)  
                riprepl(prip, len, ripbuf.xg_fip,  
                        ripbuf.xg_fport);  
            break;  
        default:  
            break;  
        }  
    }  
}  
  
Bool dorip = FALSE;  
int rippid = BADPID;  
int riplock;
```

To initialize RIP, procedure rip opens UDP port UP_RIP, and creates the lock semaphore. It then checks to see if the code executes on a gateway or host, and starts the



RIP output process if it finds that it is executing on a gateway.

After initialization, rip enters an infinite loop where it reads the next arriving message into structure ripbuf and processes it. According to the standard, RIP must verify that all fields marked must be zero contain zeroes. To perform the verification, rip calls function ripcheck, and discards incorrect messages.

After verifying the message, rip uses the command type to decide whether the message is a request (RIP_REQUEST) or a response (RIP_RESPONSE), and calls either riprep1 or riprecv to handle it. Following the protocol standard, it ignores response messages that do not originate at the well-known port (520). The standard specifies that gateways must answer all RIP requests, regardless of their originating port, but hosts that run passive RIP must ignore all requests that originate from the well-known port.

18.8.1 Must Be Zero Field Must Be Zero

Procedure ripcheck verifies that fields in the message that the protocol specifies to be zero are indeed zero.

```
/* ripcheck.c - ripcheck */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ripcheck - check a RIP packet for proper format
 */
int ripcheck(prip, len)
struct    rip *prip;
int        len;
{
    int i, j, nrts;

    switch (prip->rip_vers) {
    case 0:           /* never OK          */
        return SYSERR;
    case 1:           /* more checks below */
        break;
    default:          /* >1 always ok      */
        return OK;
    }
/* check all "must be zero" fields */
```



```
if (prip->rip_mbz)
    return SYSERR;

nrts = (len - RIPHSIZE)/sizeof(struct riprt);
for (i=0; i<nrts; ++i) {
    struct    riprt    *prr = &prip->rip_rts[i];

    if (prr->rr_mbz)
        return SYSERR;
    for (j=IP_ALEN; j<sizeof(prr->rr_addr); ++j)
        if (prr->rr_addr[j])
            return SYSERR;
}
return OK; /* this one's ok in my book... */
}
```

According to the standard, ripcheck refuses to accept messages with version number 0. It accepts messages that have version number greater than zero, but only checks zero fields in messages that have version number equal to 1. When it does check zero fields, ripcheck examines the appropriate fields in the message header. It then iterates through all route entries in the message, checking to be sure that address octets not used by IP contain zeroes.

18.8.2 Processing An Incoming Response

Procedure riprecv handles incoming response messages. The arguments consist of a pointer to the RIP message, the message length, and the sender's IP address.

```
/* riprecv.c - riprecv */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * riprecv - process a received RIP advertisement
 */
int riprecv(prip, len, gw)
struct    rip    *prip;
int      len;
IPAddr      gw;
```



```
{  
    struct route *prt, *rtget();  
    IPAddr mask;  
    int nrts, rn, ifnum;  
  
    nrts = (len - RIPHSIZE)/sizeof(struct riprt);  
    prt = rtget(gw, RTF_REMOTE); /* find the interface number */  
    if (prt == NULL)  
        return SYSERR;  
    ifnum = prt->rt_ifnum;  
    rtfree(prt);  
    wait(riplock); /* prevent updates until we're done */  
    for (rn=0; rn<nrts; ++rn) {  
        struct riprt *rp = &prip->rip_rts[rn];  
  
        rp->rr_family = net2hs(rp->rr_family);  
        rp->rr_metric = net2hl(rp->rr_metric);  
        if (!ripok(rp))  
            continue;  
        netmask(mask, rp->rr_addr);  
        rtadd(rp->rr_addr, mask, gw, rp->rr_metric,  
              ifnum, RIPRTTLE);  
    }  
    signal(riplock);  
    return OK;  
}
```

After computing the number of entries in the message, riprecv calls rtget to find the local machine's route to the sending gateway. If the machine has no route to the gateway, it cannot use the update, so it returns SYSERR. From the return route, riprecv extracts the index of the network interface used to reach the gateway, and places it in variable ifnum, which it uses when installing the route.

Riprecv iterates through individual entries in the update messages. For each entry, it calls ripok to check for malformed or illegal addresses, calls netmask to compute the network mask, and calls rtadd to add or update the route in the local IP routing table. Note that rtadd^① applies the RIP update rules: it creates a new route to the destination if no route currently exists, it replaces the metric if a route exists through the sending gateway, and it ignores the route if it already knows a less expensive one. Finally, rtadd updates the routing table.

^① See page 98 for a listing of rtadd.c.



18.8.3 Locking During Update

As we will see, the RIP output process periodically scans the routing table when it forms and sends an update. To insure that the output software does not send triggered updates until all entries in an incoming packet have been processed, riprecv waits on semaphore riplock before making any changes. It signals the semaphore once changes have been completed.

18.8.4 Verifying An Address

We said that riprecv calls procedure ripok to verify the format of an address. Ripok verifies that the address entry specifies an IP address type, the advertised metric is not more than infinity, the advertised address is not class D or E, the address does not have a zero octet in the network portion and a nonzero host portion, and the address does not specify the local loopback network (127). Other addresses specify valid destinations, and may be used for routing.

```
/* ripok.c - ripok */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 * ripok - determine if a received RIP route is ok to install
 *-----*
 */
int ripok(rp)
struct    riprt      *rp;
{
    if (rp->rr_family != AF_INET)
        return FALSE;
    if (rp->rr_metric > RIP_INFINITY)
        return FALSE;
    if (IP_CLASSD(rp->rr_addr) || IP_CLASSE(rp->rr_addr))
        return FALSE;
    if (rp->rr_addr[0] == 0 &&
        !blkequ(rp->rr_addr, ip_anyaddr, IP_ALEN))
        return FALSE;      /* net 0, host non-0          */
    if (rp->rr_addr[0] == 127)
        return FALSE;      /* loopback net              */
    return TRUE;
}
```



18.9 Responding To An Incoming Request

When a request message arrives, rip calls procedure riprepl to generate a response.

```
/* riprepl.c - riprepl */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 * riprepl - process a received RIP request
 *-----
 */

int riprepl(pripin, len, gw, port)
struct    rip  *pripin;
int       len;
IPAddr      gw;
unsigned short   port;
{
    struct    ep   *pep;
    struct    rip  *prip;
    struct    route   *prt, *rtget();
    int       rn, nrts;

    nrts = (len - RIPHSIZE)/sizeof(struct riprt);
    if (nrts == 1 && pripin->rip_rts[0].rr_family == 0 &&
        net2hl(pripin->rip_rts[0].rr_metric) == RIP_INFINITY)
        return ripsend(gw, port); /* send the full table */
    pep = (struct ep *)getbuf(Net.netpool);
    /* get to the RIP data... */
    prip = (struct rip *)((struct udp *)
        ((struct ip *)pep->ep_data)->ip_data)->u_data;
    blkcopy(prip, pripin, len);
    for (rn = 0; rn < nrts; ++rn) {
        struct riprt  *rp = &prip->rip_rts[rn];

        if (net2hs(rp->rr_family) != AF_INET)
            continue;
        prt = rtget(rp->rr_addr, RTF_LOCAL);
        if (prt) {
            rp->rr_metric = hl2net(prt->rt_metric);
```



```
    rtfree(prt);
} else
    rp->rr_metric = hl2net(RIP_INFINITY);
}
prip->rip_cmd = RIP_RESPONSE;
prip->rip_vers = RIP_VERSION;
prip->rip_mbz = 0;
udpsend(gw, port, UP_RIP, pep, len, 1);
return OK;
}
```

RIP allows one special case in requests. The standard specifies that if the request contains exactly one entry that has address family identifier 0 and metric infinity, the recipient should generate a full update message as a response. Otherwise, the recipient should respond by supplying its local routes for each address specified in the request.

Riprep1 calls procedure ripsend, which is also used by the output process, to generate a full update in response to the special case request. For normal requests, riprep1 allocates a buffer to hold the response. Once it has copied the incoming message into the new buffer, riprep1 iterates through each entry. It uses rtget to look up the local route for the entry, and copies the metric from that route into the response, or assigns RIP_INFINITY, if no route exists. Before sending the response, riprep1 assigns values to the command field in the message header (to make it a response) and the version field (to indicate the appropriate version number). It then calls udpsend to transmit the datagram.

18.10 Generating Update Messages

Procedure ripsend creates and sends an update.

```
/* ripsend.c - ripsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ripsend - send a RIP update
 */
int ripsend(IPAddr gw, /* remote gateway (FFFFFF == all) */
```



```
unsigned short      port;          /* remote port           */
{
    struct    rq    *prq, rqinfo[NIF]; /* here, so reentrant   */
    struct    route   *prt;
    int        i, pn;

    if (ripifset(rqinfo, gw, port) != OK)
        return SYSERR;

    wait(Route.ri_mutex);
    for (i=0; i<RT_TSIZE; ++i)
        for (prt=rtable[i]; prt; prt=prt->rt_next)
            ripadd(rqinfo, prt);
    if (Route.ri_default)
        ripadd(rqinfo, Route.ri_default);
    signal(Route.ri_mutex);

    for (i=0; i<Net.nif; ++i)
        if (rqinfo[i].rq_active) {
            prq = &rqinfo[i];
            for (pn=0; pn<=prq->rq_cur; ++pn)
                udpsend(prq->rq_ip, prq->rq_port, UP_RIP,
                         prq->rq_pep[pn], prq->rq_len[pn]);
        }
}
```

Ripsend begins by calling ripifset to initialize array rqinfo, the array that holds an update message for each of the network interfaces. It then waits on semaphore Route.ri_mutex to obtain exclusive use of the routing table, and iterates through all possible routes. For each route, ripsend calls procedure ripadd. Ripadd implements heuristics like split horizon; it iterates through the set of network interfaces and determines whether to add the route to the copy of the update message associated with each. Finally, after adding all the routes, ripsend iterates through the network interfaces and calls udpsend to send the appropriate copy of the update message. It does not send an update to the local host because field rq_active is FALSE in the array element that corresponds to the local host.

18.11 Initializing Copies Of An Update Message

Procedure ripifset initializes array rqinfo. The second argument either contains the



IP address of a client that sent a specific request, or the all 1s address, which requests a broadcast to all networks.

```
/* ripifset.c - ripifset */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ripifset - set the per-interface data for a RIP update
 */
int ripifset(rqinfo, gw, port)
struct    rq    rqinfo[];
IPAddr      gw;          /* remote gateway (FFFFFF == all)   */
unsigned short  port;        /* remote port                      */
{
    struct    route    *prt, rtget();
    int      ifn;

    if (!blkque(gw, ip_maskall, IP_ALEN)) {
        for (ifn=0; ifn<Net.nif; ++ifn)
            rqinfo[ifn].rq_active = FALSE;
        prt = rtget(gw, RTF_LOCAL);
        if (prt == 0)
            return SYSERR;
        ifn = prt->rt_ifnum;
        rtfree(prt);
        if (ifn == NI_LOCAL)
            return SYSERR;
        blkcopy(rqinfo[ifn].rq_ip, gw, IP_ALEN);
        rqinfo[ifn].rq_port = port;
        rqinfo[ifn].rq_active = TRUE;
        rqinfo[ifn].rq_cur = -1;
        rqinfo[ifn].rq_nrts = MAXRIPROUTES;
        return OK;
    }
    /* else, all interfaces */
    for (ifn=0; ifn<Net.nif; ++ifn) {
        blkcopy(rqinfo[ifn].rq_ip, nif[ifn].ni_brc, IP_ALEN);
        rqinfo[ifn].rq_port = port;
    }
}
```



```
    rqinfo[ifn].rq_active = TRUE;
    rqinfo[ifn].rq_cur = -1;
    rqinfo[ifn].rq_nrts = MAXRIPROUTES;
}
rqinfo[NI_LOCAL].rq_active = FALSE; /* never do this one */
return OK;
}
```

If the request specifies a particular address, ripifset disables RIP processing on all interfaces except the one that leads to the specified address. Otherwise, it enables RIP processing on all interfaces except the pseudo-network interface for the local host. When enabling an interface, ripifset initializes the index of routes in the current datagram to -1, indicating that there are no datagrams present.

18.11.1 Adding Routes to Copies Of An Update Message

Procedure ripadd adds a route to each copy of an update message.

```
/* ripadd.c - ripadd */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ripadd - add the given route to the RIP packets yet to send
 */
int ripadd(rqinfo, prt)
struct    rq    rqinfo[];
struct    route    *prt;
{
    IPAddr    net;
    int i, metric, pn, rn;

    for (i=0; i<Net.nif; ++i) {
        struct    rq    *prq = &rqinfo[i];
        struct    riprt    *rp;

        if (!rqinfo[i].rq_active || nif[i].ni_state != NIS_UP)
            continue;
        metric = ripmetric(prt, i);
        if (metric == SYSERR)
```



```
        continue;

    if (prq->rq_nrts >= MAXRIPROUTES &&
        ripstart(prq) != OK)
        continue;

    pn = prq->rq_cur;
    rn = prq->rq_nrts++;
    rp = &prq->rq_prip->rip_rts[rn];
    rp->rr_family = hs2net(AF_INET);
    rp->rr_mbz = 0;
    netnum(net, prt->rt_net);
    bzero(rp->rr_addr, sizeof(rp->rr_addr));
    if (blkque(nif[i].ni_net, net, IP_ALEN) ||
        blkque(prt->rt_mask, ip_maskall, IP_ALEN)) {
        blkcopy(rp->rr_addr, prt->rt_net, IP_ALEN);
    } else /* send the net part only (esp. for subnets) */
        blkcopy(rp->rr_addr, net, IP_ALEN);
    rp->rr_metric = hl2net(metric);
    prq->rq_len[pn] += sizeof(struct riprt);
}

return OK;
}
```

Given a route as an argument, ripadd iterates through the set of network interfaces and examines whether the route should be added to each. If the RIP entry that corresponds to the interface is active and the interface is "up," ripadd proceeds to add the route. It calls ripmetric to compute a metric. If allocates space in the message being constructed, and calls ripstart to allocate another datagram if the current one is full. It fills in the route family and must be zero field for the route.

Ripadd compares the destination address of the route to the IP address of the interface over which it will be sent to determine the exact form of address to use. Normally, RIP masks off the subnet and host portions of a destination address and only propagates the network portion. However, it propagates subnet information within a subnetted network. Thus, if the interface over which the route should be sent lies on a subnet of the destination address, it propagates the subnet portion of the address along with the network portion. Once ripadd computes the correct IP address to advertise and the metric to use, it fills in the next entry in the message and continues iterating through the routes.

18.11.2 Computing A Metric To Advertise

Procedure ripmetric computes the metric that will be advertised along with a given



route.

```
/* ripmetric.c - ripmetric */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ripmetric - compute the RIP metric for a route we advertise
 */
int ripmetric(prt, ifnum)
struct route *prt;
int ifnum;
{
    /* only advertise the net route for our interfaces */

    if (prt->rt_ifnum == NI_LOCAL &&
        blkequ(prt->rt_mask, ip_maskall, IP_ALEN))
        return SYSERR;

    if (prt->rt_ifnum == ifnum)
        return RIP_INFINITY; /* poison reverse */
    /* else, add one to our cost */
    return prt->rt_metric + 1;
}
```

Ripmetric implements poison reverse by changing the cost of a route to infinity if the route directs datagrams out over the interface on which the route is being advertised. In other cases it translates from the internal metric, which uses cost 0 for direct connections, to the standard RIP metric, which uses cost 1 to direct connections.

18.11.3 Allocating A Datagram For A RIP Message

As we have seen, ripadd collects routing advertisements into an update message, making a copy for each network interface. When it finds that it has filled a RIP message completely, rip add calls ripstart to allocate a new datagram.

```
/* ripstart.c - ripstart */

#include <conf.h>
#include <kernel.h>
```



```
#include <network.h>

/*
 * ripstart - initialize an interface's RIP packet data
 */
int ripstart(prq)
struct rq *prq;
{
    struct ep *pep;
    struct ip *pip;
    struct udp *pudp;
    struct rip *prip;
    int pn;

    pn = ++prq->rq_cur;
    if (pn >= MAXN RIP)
        return SYSERR;
    prq->rq_nrts = 0;
    prq->rq_pep[pn] = pep = (struct ep *)getbuf(Net.netpool);
    if (pep == SYSERR)
        return SYSERR;
    pip = (struct ip *)pep->ep_data;
    pudp = (struct udp *)pip->ip_data;
    prip = (struct rip *)pudp->u_data;

    prq->rq_prip = prip;
    prq->rq_len[pn] = RIPHSIZE;
    prip->rip_cmd = RIP_RESPONSE;
    prip->rip_vers = RIP_VERSION;
    prip->rip_mbz = 0;
    return OK;
}
```

Ripstart takes a single argument that contains a pointer to an entry in the rqinfo array. It increments the index of the current datagram in that entry, allocates a buffer for another one, fills fields in the RIP header in the newly allocated buffer, and sets the must be zero field in the new datagram to zero.



18.12 Generating Periodic RIP Output

Gateways send RIP responses periodically. To do so, they create a process to execute procedure ripout. The output process enters an infinite loop that delays for 30 seconds, and then calls ripsend to generate an update on all interfaces. The code is written so that it allows triggered updates using message passing. If a message arrives during the delay, ripout aborts the delay and immediately sends the update.

```
/* ripout.c - ripout */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ripout - do the RIP route exchange protocol, output side
 */
PROCESS ripout(argc)
{
    int rnd;

    rippid = getpid();
    dorip = TRUE;
    /* advertise our routes */

    rnd = 0; /* "randomizer" */
    while (TRUE) {
        sleep10(RIPOUTMIN); /* minimum time between each */
        if (++rnd > RIPDELTA)
            rnd = -RIPDELTA;
        recvtim(RIPINT-RIPOUTMIN+rnd);
        wait(riplock);
        ripsend(ip_maskall, UP_RIP);
        signal(riplock);
    }
}
```

Although the code seems trivial, it handles three small details. First, to prevent triggered updates from occurring until it has formed and sent one complete update, ripout waits on semaphore riplock before calling ripsend. Second, ripout imposes a minimum delay of RIPOUTMIN tenths of seconds between updates (even if they are



triggered). To insure the minimum delay, it calls sleep10 directly for RIPOUTMIN tenths of seconds before calling recvtime for the remaining time.

When computing the remaining time to delay for the call to recvtime, ripout starts with the standard RIP update period, subtracts the minimum delay that has already occurred, and finally adds a small "random" integer. The small random delay helps RIP avoid broadcast storms caused by triggered updates.

The code simulates random delay by keeping a global integer that it increments by 1 for each call. When the integer becomes larger than RIPDELTA, ripout sets it to negative RIPDELTA and continues.

18.13 Limitations Of RIP

The RIP design limits the environments in which it can be used. First, because RIP uses 16 for infinity, it cannot be used in an internet that has a diameter greater than 15. Second, because RIP uses fixed values for the update period and timeout, it cannot be used in networks that have high loss. Third, because RIP uses fixed metrics when propagating routes, it cannot be used in internets that use dynamic measures to select routes (i.e., current delay or current load).

18.14 Summary

The Routing Information Protocol, RIP, uses a vector-distance algorithm to propagate routes. To maintain stability, RIP uses heuristics like split horizon, route timeout, and poison reverse.

We examined an implementation in which separate input and output processes handled RIP. The input process accepts a direct query, a response to a direct query, or a periodic response broadcast to all machines on the local network. The output process generates periodic broadcast of updates or triggered updates.

To trigger an update, any program can send a message directly to the output process. To prevent triggered updates from occurring as routes change, the RIP software uses semaphore riplock. The output process waits on the semaphore before compiling an update message.

18.15 FOR FURTHER STUDY

Hedrick [RFC 1058] defines the protocol and specifies the heuristics discussed in this chapter. It also gives algorithms for handling incoming requests and responses.



18.16 EXERCISES

1. Procedure ripsend allocates array rqinfo as an automatic, stack variable. What would happen if it allocated the array as a static, global variable instead?
2. Read the code carefully. How does it handle specific requests?
3. Consider the interaction between RIP and IP. Describe a design that separates the policy RIP uses to choose routes from the policy that IP uses to forward datagrams.
4. Can our RIP software propagate multiple default routes? Explain.
5. Suppose a network manager needed to restrict route propagation. How would you modify the code to allow the manager to specify routes that should not be propagated?



19 OSPF: Route Propagation With An SPF Algorithm

19.1 Introduction

The previous chapter describes an interior gateway protocol that uses a vector-distance algorithm to propagate routing information among gateways and hosts. This chapter examines, an alternative interior gateway protocol known as the Open Shortest Path First (OSPF) protocol. OSPF uses a link-state algorithm to propagate routing information.

When using a vector-distance protocol, each gateway propagates a list of the networks it can reach along with the distance to each network. Gateways that receive the list use it to compute new routes, and then propagate their list of reachable networks. When using a link-state protocol, each gateway propagates the status of its individual connections to networks. The protocol delivers each link state message to all other participating gateways. When link state information arrives at a given gateway, the gateway adds the information to its current topology database. If an incoming link state message changes the gateway's database, the gateway must recompute routes. To compute the next-hop along the shortest path to each destination, the gateway runs a local computation.

19.2 OSPF Configuration And Options

Because OSPF attempts to encompass a wide variety of internet architectures and routing configurations, the protocol includes many alternatives. For example, consider an internet in which a serial line connects two gateways. A conventional implementation of IP treats each connection as a network, and requires that the line be assigned an IP address. However, some implementations of IP conserve protocol addresses by allowing two gateways to communicate across an anonymous connection, a serial line that does not have an IP address. Unlike most routing protocols, OSPF can propagate routing



information about anonymous connections as well as conventional connections.

The generality in OSPF has a cost — the protocol is both large and complex, and the specification can be difficult to understand. More important, to be totally general, OSPF software must be designed so that it can be configured for a specific internet. Indeed, the protocol contains many possible variants and special cases. For example, although OSPF has been designed as an interior gateway protocol for use within an autonomous system, it allows a system manager to partition the autonomous system into subsets called areas. Each gateway must be placed, in one of the areas, and OSPF specifies how a gateway at the border of one area communicates routing information with a gateway at the border of another.

To help understand OSPF without becoming entangled in generality and special cases, this chapter examines features of the protocol individually. The next section describes OSPF's graph-theoretic model of network connectivity, and defines important concepts and terminology used throughout the chapter. Later sections show how an OSPF gateway discovers neighboring gateways and maintains information about them, how the set of OSPF gateways attached to a network elects a single, distinguished gateway to handle link state updates for the network, how OSPF disseminates link state information, and how OSPF computes routes.

19.3 OSPF's Graph-Theoretic Model

Like most link-state algorithms, OSPF uses a graph-theoretic model of network topology to compute shortest paths. Each gateway periodically broadcasts information about the status of its connections to networks; OSPF floods each status message to all participating gateways. A gateway uses arriving link state information to assemble a graph. Whenever a gateway receives information that changes its copy of the topology graph (e.g., because a connection failed), it runs a conventional graph algorithm to compute shortest paths in the graph, and uses the results to build a new next-hop routing table.

OSPF uses a directed graph to model an internet. Because the OSPF graph models internet connections, it is sometimes called a topology graph. Each node in an OSPF topology graph either corresponds to a gateway or a network. If a physical connection exists between two objects in an internet, the OSPF graph contains a pair of directed edges (one in each direction) between the two nodes that represent the objects. For example, Figure 19.1 shows a connection between a gateway and a network, and the corresponding edges in an OSPF graph.

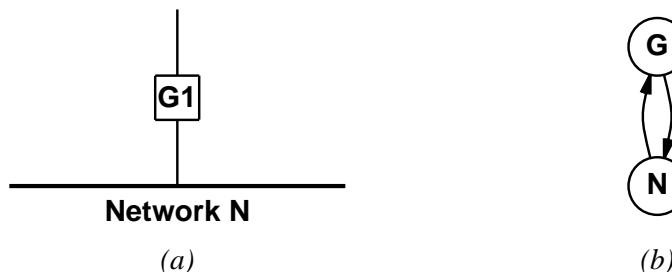


Figure 19.1 (a) An example connection between a gateway and a network, and (b) the corresponding pair of edges in an OSPF graph.

OSPF uses the term multiaccess network to refer to a network that connects multiple gateways (e.g., an Ethernet with two gateways attached). The OSPF graph for a multiaccess network consists of a node for the network, a node for each gateway, and a pair of edges for each connection between a gateway and a network. For example, Figure 19.2 shows a multiaccess network with multiple gateways attached and the corresponding OSPF graph.

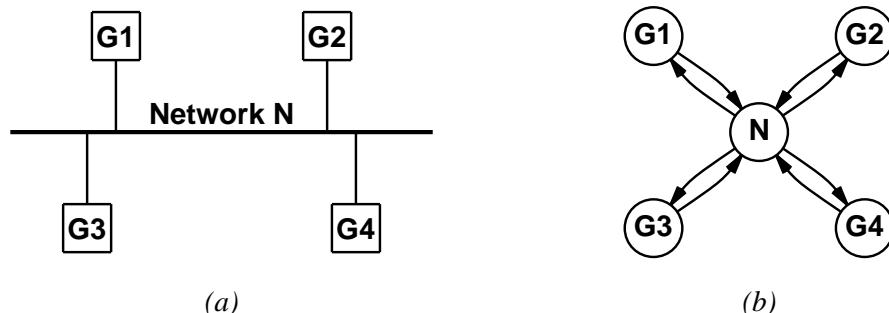


Figure 19.2 (a) a multiaccess network with four gateways attached, and (b) the corresponding OSPF graph for the four connections.

OSPF also permits the graph to model an anonymous point-to-point connection between a pair of gateways. To model an anonymous serial connection, OSPF uses a pair of edges connecting the nodes that represent the two gateways. Thus, unlike the graph for a multiaccess network, the graph for an anonymous serial connection does not contain a node for the serial line.

To enable gateways to compute shortest paths, each edge in an OSPF graph is assigned a weight that corresponds to the cost of using the path. A network administrator who configures OSPF assigns weights by giving a positive number for each network interface. The cost assigned to a particular interface can be chosen for administrative or technical reasons. For example, a cost can reflect the monetary cost of using the



interface, the network bandwidth available through the interface, or an administrative policy designed to encourage or discourage use of the interface. Figure 19.3 shows an example internet with costs assigned to each interface.

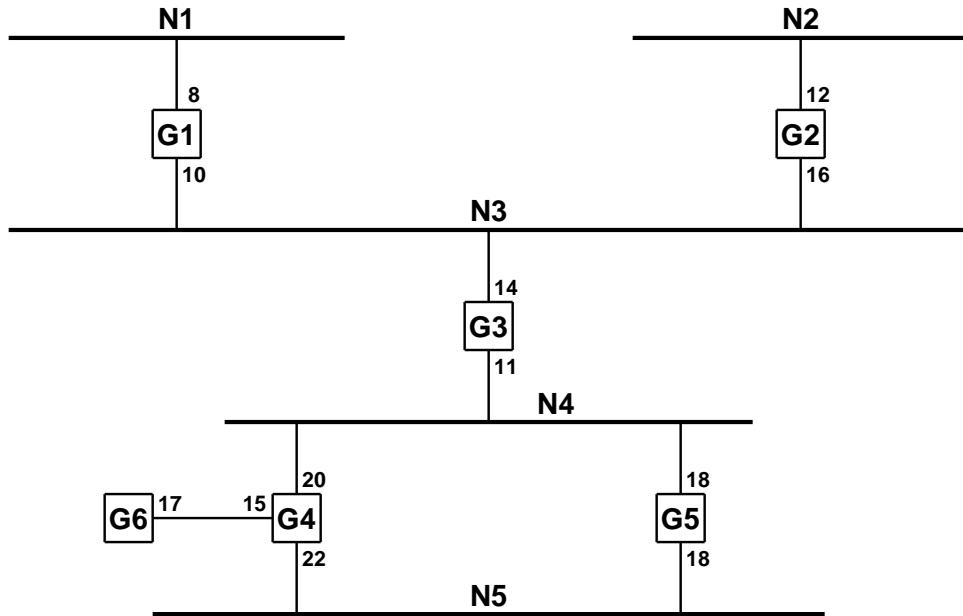


Figure 19.3 An example internet of networks and gateways with costs assigned to each interface.

Like most routing algorithms, OSPF builds routing tables that forward datagrams along a least-cost path. Thus, although both gateways G4 and G5 in Figure 19.3 provide connectivity between networks N4 and N5, OSPF will route datagrams through gateway G5 because it has been assigned lower costs.

Figure 19.4 shows the OSPF graph model of the internet in Figure 19.3. The graph contains a node for each gateway and a node for each multiaccess network. The graph does not contain a node for the anonymous connection between gateways G4 and G6. Instead, a pair of edges directly connects the node for G4 to the node for G6.

As the figure shows, each edge in an OSPF graph that leads from a node representing a gateway to a node representing a network has a weight assigned equal to the cost of using the interface. However, each edge from a node that represents a network to a node that represents a gateway has zero weight. The reason for an asymmetric assignment of weights is simple: associating a separate weight with an edge from a gateway node to a network node makes it possible to assign a cost to each connection.

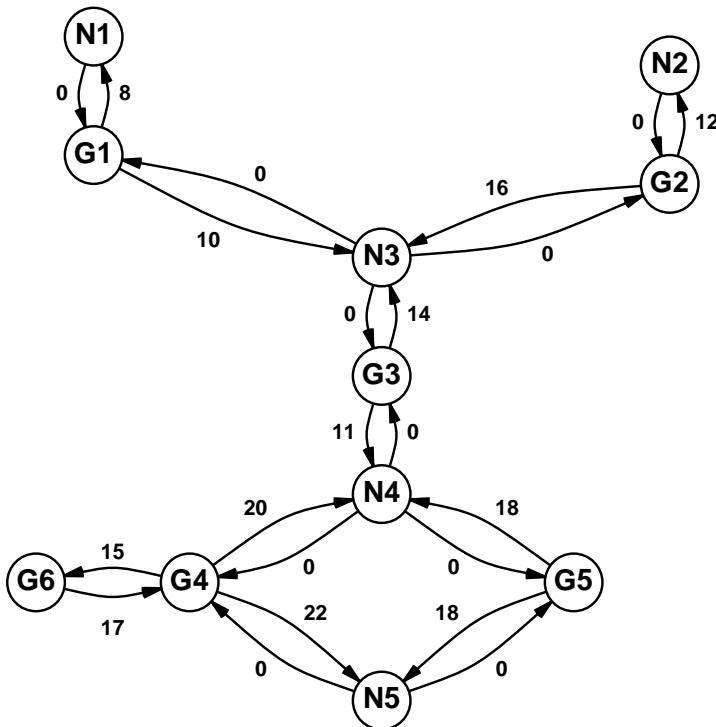


Figure 19.4 The OSPF graph for the internet in Figure 19.3. The number on an edge in the graph corresponds to the weight assigned to the network connection that it represents.

Having zero weight on an edge that leads from a network node to a gateway node ensures that OSPF only counts the cost once along a path through the network. Thus, a manager can exert administrative control by making the cost of accessing a given network from one gateway greater than the cost of accessing the same network from a different gateway. Furthermore, zero weight makes the costs easier to understand because they permit a manager to sum the costs along a path from the source to a destination. To summarize:

Because OSPF uses a directed graph model of an internet to compute shortest paths, and because network managers can assign costs independent of the network hardware, edges from nodes that represent networks to nodes that represent gateways have zero weight.

19.4 OSPF Declarations

To understand how OSPF uses the graph model when propagating information and



computing shortest paths, it is necessary to understand many details including the formal of OSPF packets and the internal data structures. This section presents three files of declarations used throughout the code.

19.4.1 OSPF Packet Format Declarations

File ospf_pkt.h contains declarations and constants that define the OSPF packet format and header constants.

```
/* ospf_pkt.h */

/* OSPF packet format */

struct ospf {
    unsigned char ospf_version; /* Version Number */
    unsigned char ospf_type; /* Packet Type */
    unsigned short   ospf_len; /* Packet Length */
    unsigned long    ospf_rid; /* Router Identifier */
    unsigned long    ospf_aid; /* Area Identifier */
    unsigned short   ospf_cksum; /* Check Sum */
    unsigned short   ospf_authtype; /* Authentication Type */
    unsigned char    ospf_auth[AUTHLEN]; /* Authentication Field */
    unsigned char    ospf_data[1];
};

#define MINHDRLEN 24 /* OSPF base header length */

/* OSPF Packet Types */

#define T_HELLO      1 /* Hello packet */
#define T_DATADESC   2 /* Database Description */
#define T_LSREQ      3 /* Link State Request */
#define T_LSUPDATE   4 /* Link State Update */
#define T_LSACK      5 /* Link State Acknowledgement */

/* OSPF Authentication Types */

#define AU_NONE      0 /* No Authentication */
#define AU_PASSWD1   /* Simple Password */

/* OSPF Hello Packet */

struct ospf_hello {
```



```
unsigned long oh_netmask; /* Network Mask */  
unsigned short oh_hintv; /* Hello Interval (seconds) */  
unsigned char oh_opts; /* Options */  
unsigned char oh_prio; /* Sender's Router Priority */  
unsigned long oh_rdintv; /* Seconds Before Declare Dead */  
unsigned long oh_drid; /* Designated Router ID */  
unsigned long oh_brid; /* Backup Designated Router ID */  
unsigned long oh_neighbor[1]; /* Living Neighbors */  
};  
  
#define MINHELLOLEN (MINHDRLEN + 20)  
  
/* OSPF Database Description Packet */  
  
struct ospf_dd {  
    unsigned short dd_mbz; /* Must Be Zero */  
    unsigned char dd_opts; /* Options */  
    unsigned char dd_control; /* Control Bits (DDC_* below) */  
    unsigned long dd_seq; /* Sequence Number */  
    struct ospf_lss dd_lss[1]; /* Link State Advertisements */  
};  
  
#define MINDDLEN (MINHDRLEN + 8)  
  
#define DDC_INIT 0x04 /* Initial Sequence */  
#define DDC_MORE 0x02 /* More to follow */  
#define DDC_MSTR 0x01 /* This Router is Master */
```

19.4.2 OSPF Interface Declarations

A gateway keeps information such as the assigned cost for each of its interfaces in structure `ospf_if`. File `ospf_if.h` contains the declarations.

```
/* ospf_if.h */  
  
/* OSPF Interface Information */  
  
struct ospf_if {  
    unsigned int if_type; /* one of IFT_* below */  
    unsigned char if_state; /* one of IFS_* below */  
    unsigned char if_event; /* one of IFE_* below */  
    unsigned int if_hintv; /* Hello Packet Interval */  
    unsigned int if_rdintv; /* Router Dead Interval */
```



```
unsigned int    if_rintv;      /* Retransmit Interval          */
timer_t        if_twait;      /* Wait Timer for WAITING state */
unsigned int    if_xdelay;     /* Estimated Transmission Delay */
IPAddr         if_dipa; /* IP Address of desig. router */
IPAddr         if_bipa; /* IP address of backup router */
unsigned int    if_metric;    /* Cost to use this Interface   */
unsigned char   if_auth[AUTHLEN]; /* Authentication Key           */
int            if_nbmutex;   /* Mut. Excl. Semaphor for nbtab*/
struct ospf_ar *if_area; /* Area Structure Back-pointer */
struct ospf_if *if_next; /* Next Interface for Area      */
struct ospf_nb if_nbtab[MAXNBR+1]; /* Neighbor list this net */

};

#define if_rid       if_nbtab[0].nb_rid
#define if_prio      if_nbtab[0].nb_prio
#define if_drid      if_nbtab[0].nb_drid
#define if_brid      if_nbtab[0].nb_brid
#define if_opts      if_nbtab[0].nb_opts

/* Interface Types */

#define IFT_BROADCAST 0          /* Interface Supports Broadcast */
#define IFT_MULTI 1             /* Interface Supports Multicast */
#define IFT_PT2PT 2             /* Interface is Point-to-point */
#define IFT_VIRTUAL 3            /* Interface is a Virtual Link */

/* Interface States */

#define IFS_DOWN 0              /* Interface non-functional */
#define IFS_LOOPBACK 1           /* Interface in loopback mode */
#define IFS_WAITING 2            /* Electing Backup Desig. Rtr. */
#define IFS_PT2PT 3              /* Interface is Point-to-Point */
#define IFS_DROTHER 4             /* not Designated Router */
#define IFS_BACKUP 5              /* Is Backup Designated Router */
#define IFS_DR 6                 /* Is Designated Router */

/* Scheduled Interface Events */

#define IFE_BSEEN 0x01           /* Have Seen Backup DR Claim */
#define IFE_NCHNG 0x02           /* A Neighbor State Changed */
```



19.4.3 Global Constant And Data Structure Declarations

File ospf.h defines remaining data structures and constants used throughout OSPF and includes the other .h files.

```
/* ospf.h */

/* OSPF process info */

extern PROCESS      ospf();

#define OSPFSTK        4096      /* stack size for OSPF input      */
#define OSPFPRI        90        /* OSPF priority      */
#define OSPFNAM        "ospfinp" /* name of OSPF input process   */
#define OSPFARGC        0         /* count of args to ospfinp */

extern PROCESS      ospfhello();

#define OSPFHSTK 4096      /* stack size for OSPF hello      */
#define OSPFHPRI 90        /* OSPF priority      */
#define OSPFHNAME "ospfhello" /* name of OSPF hello process   */
#define OSPFHARGC 0         /* count of args to ospfhello */

#define OSPFQLEN 20        /* OSPF input port queue length */
#define MAXNBR    32        /* Max # active neighbors */

/* Manifest OSPF Configuration Parameters */

#define OSPF_VERSION 2        /* protocol version number */
#define HELLOINTV10    /* Hello Interval (seconds) */
#define DEADINTV 4*HELLOINTV /* Router Dead Interval (secs) */
#define RXMTINTV 5        /* Retransmit Interval (secs) */

#define AUTHLEN     8        /* 64-bit Password */

typedef int timer_t;      /* OSPF count-down timers (secs)*/

/* Area Information */

struct ospf_ar {
    unsigned long ar_id; /* Area Identifier */
    unsigned char ar_authtype; /* Authentication Type */
    unsigned char ar_auth[AUTHLEN]; /* Password, if ARA_PASSWD */
    int ar_dbmutex; /* Mutex for Top. Database */
    int ar_hmod; /* Hash Table Modulus */
}
```



```
    struct ospf_db      **ar_htable; /* LSA Hash Table      */
    struct ospf_db      *ar_dblhead; /* Head of Top. Database   */
    struct ospf_if      *ar_if; /* List of Interfaces This Area */
};

#define ARA_NONE 0           /* No Authentication for Area   */
#define ARA_PASSWD 1          /* Simple-password Auth. */

/* Neighbor Information */

struct ospf_nb {
    unsigned char nb_state; /* neighbor conversation state */
    unsigned long nb_seq;   /* DD packet sequence number */
    unsigned long nb_rid;   /* neighbor's router ID */
    unsigned char nb_prio;  /* neighbor's router priority */
    IPAddr       nb_ipa;   /* neighbor's IP address */
    unsigned char nb_opts;  /* options */
    unsigned long nb_drid;  /* neighbor's designated router */
    unsigned long nb_brid;  /* backup designated router */
    Bool        nb_master; /* nonzero if we are master */
    timer_t      nb_lastheard; /* seconds since last HELLO */
    timer_t      nb_trexmt;  /* Retransmit Timer (secs) */
    timer_t      nb_tlastdd; /* Slave Last DD pkt. Timer */
    timer_t      nb_tlsr;    /* LSR Retransmit Timer */
    int         nb_lsal;   /* Link Status Adv. List */
    int         nb_dsl;    /* Database Summary List */
    int         nb_lsrl;   /* Link Status Req. List */
};

/* Neighbor States */

#define NBS_DOWN 0
#define NBS_ATTEMPT 1
#define NBS_INIT 2
#define NBS_2WAY 3
#define NBS_EXSTART 4
#define NBS_EXCHNG 5
#define NBS_LOADING 6
#define NBS_FULL 7

/* Neighbor Options */
```



```
#define NBO_T          0x01      /* Router Does Supports IP TOS */
#define NBO_E          0x02      /* Can Use External Routes */

#define NBMAXLIST 10           /* Max Retrans. Queue Lengths */

#include "ospf_ls.h"
#include "ospf_pkt.h"
#include "ospf_if.h"
#include "ospf_db.h"

extern struct ospf_ar  ospf_ar[];
extern struct ospf_if  ospf_if[];
extern int      ospf_iport;
extern int      ospf_lspspool;
extern IPAddr   AllSPFRouters, AllDRouters;
```

19.5 Adjacency And Link State Propagation

A set of gateways running OSPF uses flooding to disseminate link state messages (i.e., they propagate each link state message to all participating gateways). To ensure that all gateways in the set receive a given message, the gateways form logical connections among themselves and pass copies of link state messages along the connections. Usually, each logical connection between two gateways corresponds to a single physical connection (e.g., a serial line that interconnects the two gateways). However, OSPF can be configured so two gateways form a connection that spans intermediate gateways and networks.

In essence flooding spreads link state messages quickly by duplicating a message as needed — a gateway that receives a message on one connection sends a copy of the message on each of its other connections. However, the OSPF flooding scheme does not blindly forward copies of each link state message that arrives. In particular, if connections among gateways form a cycle, OSPF ensures that the gateways will not send a copy of a given message around the cycle forever.

To prevent gateways from forwarding extra, unneeded copies of a message, OSPF examines each message that arrives. When a message arrives at a gateway carrying new information, the gateway floods a copy across each of its logical connections. When a duplicate message arrives, the gateway discards the message without sending copies. Thus, OSPF does not rely on a network manager to configure loop-free connections; the protocol automatically eliminates duplicates.

Each logical connection between a pair of gateways is called an adjacency. The



easiest way to think about OSPF adjacencies is to imagine an undirected graph in which each node corresponds to a gateway, and each edge corresponds to an OSPF adjacency^①. When OSPF gateways propagate routing information, they do so along paths specified by the adjacency graph. Thus, for correct operation of the protocol, all gateways must be able to communicate by sending across adjacencies. That is, for an arbitrary pair of gateways, G1 and G2, it must be possible for G1 to reach G2 using a path composed entirely of adjacent gateways.

In most cases, network administrators configure OSPF to form adjacencies between pairs of neighboring gateways. For example, two neighboring gateways connected by a serial line usually form an adjacency, and send routing information across the connection. However, the adjacency relationship is dynamic — when a network or gateway fails, adjacencies may change to ensure that all participating gateways continue to receive routing information.

It may seem that every pair of neighboring gateways should form an adjacency. However, if all pairs of gateways form adjacencies, copies of link state messages can be forwarded unnecessarily, resulting in wasted bandwidth. The problem of excessive adjacencies becomes especially severe on a multiaccess network like an Ethernet because a physical path exists between all pairs of gateways that attach to the net. If N gateways attach to a given network, they can form $(N^2-N)/2$ possible adjacencies. Whenever a gateway transmits a message, it sends a copy of the message to N-1 adjacent neighbors. In the worst case, each of the neighbors floods the message to all other adjacent neighbors immediately. Thus, the message will pass across most adjacencies twice before the gateways detect the duplicate and stop forwarding copies. As a result, approximately N^2 copies of the message will be transferred, even though N copies suffice.

19.6 Discovering Neighboring Gateways With Hello

To discover neighboring gateways and maintain appropriate adjacency relationships, OSPF uses a Hello protocol^②. A pair of gateways that use the Hello protocol periodically exchange packets that ensure that both sides remain alive and agree to an adjacency relationship.

Hello is especially important on a multiaccess network like an Ethernet for two reasons. First, although a gateway can crash or reboot at any time, the network hardware does not detect or report such events. Thus, to maintain status information about each other, two gateways attached to a multiaccess network must exchange packets. Second all gateways attached to a multiaccess network can communicate directly. Thus, OSPF

^① The term adjacency is borrowed from graph theory.

^② OSPF's Hello protocol differs from an older protocol of the same name.

needs to prevent such gateways from forming excessive adjacencies.

The OSPF standard uses a finite state machine to specify how a gateway using Hello interacts with a neighboring gateway. Figure 19.5 shows the states and the major transitions among them.

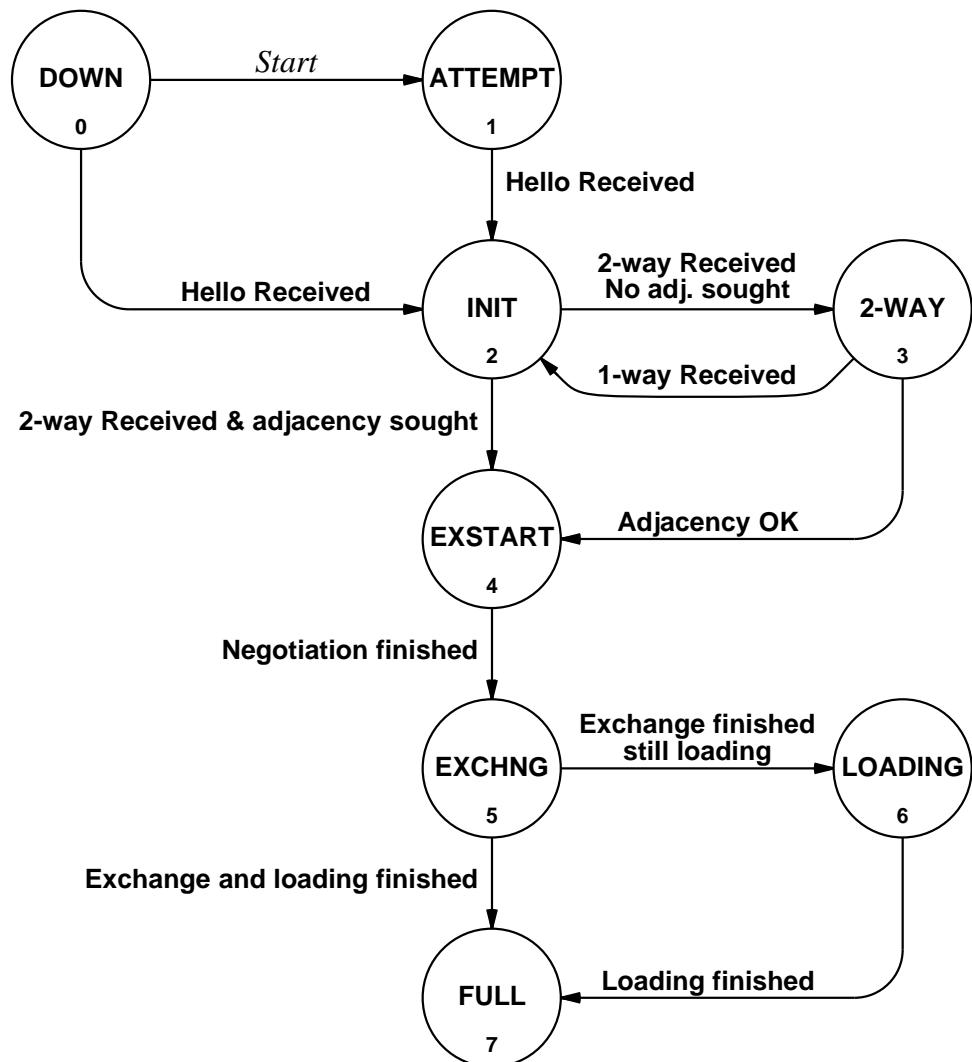


Figure 19.5 Eight possible states of a neighboring gateway as determined by OSPF's Hello protocol. The example software uses the integer values shown to impose a linear ordering among the states, as required by the standard.



In general, all neighbors start in the DOWN state, which means that no communication has been attempted. After receiving a Hello packet from a neighbor, a gateway moves the neighbor from the DOWN state to the INIT state. Following INIT, the neighbor moves to either 2-WAY, which means that communication has been established but the neighbor is not adjacent, or to EXSTART, which means that communication has been established and the two gateways are negotiating adjacency.

After negotiation finishes, the gateways begin exchanging information from their topology database to ensure that they have exactly the same underlying graph for the internet. One of the two adjacent gateways becomes a master, and polls the other for database information. The nonmaster returns database description packets that tell about the most recently received information for each link in the topology graph. Exchanging information is especially important when establishing an adjacency because the information in one gateway can become out of date during a network disconnection. Each piece of topology information includes a sequence number, so a gateway can tell whether information in a neighbors database description is more current than information in the gateway's database. After the exchange occurs and all topology has been loaded, a gateway uses state FULL for The neighbor. When in FULL, the two gateways periodically exchange packets to ensure that the connection remains intact.

A strong analogy exists between the finite state machine OSPF uses for neighbor status and the TCP finite state machine covered in Chapters 11 and 12. Both finite state machines specify a few basic states for the software, and both machines describe how the software reacts to an arriving message. Both machines also have complex semantic actions associated with each state. Thus, the implementation of the neighbor finite state machine parallels the TCP finite state machine implementation. Although the standard specifies how OSPF operates in each state, the code to handle a given state cannot be segregated into a single procedure — the details permeate much of the software.

19.7 Sending Hello Packets

An example of using state information can be found in procedure `ospf_hsend`, which sends hello packets on a specified interface.

```
/* ospf_hsend.c - ospf_hsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * ospf_hsend - send OSPF hello packet for an interface
```



```
*-----  
*/  
  
int ospf_hsend(ifn)  
int ifn;  
{  
    struct ospf_if          *pif = &ospf_if[ifn];  
    struct ep      *pep;  
    struct ip      *pip;  
    struct ospf      *po;  
    struct ospf_hello *poh;  
    struct ospf_nb     *pnb;  
    int             i, nn;  
  
    pep = ospfhtmpl(pif);  
    wait(pif->if_nbmutex);  
    pip = (struct ip *) pep->ep_data;  
    po = (struct ospf *) pip->ip_data;  
    poh = (struct ospf_hello *) po->ospf_data;  
    pnb = &pif->if_nbtab[1];  
    for (i=0, nn=0; i<MAXNBR; ++i, ++pnb)  
        if (pnb->nb_state >= NBS_INIT) {  
            poh->oh_neighbor[nn++] = pnb->nb_rid;  
            po->ospf_len += sizeof(poh->oh_neighbor[0]);  
        }  
    signal(pif->if_nbmutex);  
    blkcopy(poh->oh_netmask, nif[ifn].ni_mask, IP_ALEN);  
    po->ospf_authtype = net2hs(pif->if_area->ar_authtype);  
    bzero(poh->ospf_auth, AUTHLEN);  
    po->ospf_cksum = 0;  
    po->ospf_cksum = cksum(po, po->ospf_len>>1);  
    blkcopy(po->ospf_auth, pif->if_area->ar_auth, AUTHLEN);  
    blkcopy(pip->ip_src, nif[ifn].ni_ip, IP_ALEN);  
    if (ifn == NI_PRIMARY)  
        ipsend(AllSPFRouters, pep, len, IPT_OSPF,  
               IPP_INCTL, 1);  
    else  
        ipsend(nif[ifn].ni_brc, pep, len, IPT_OSPF,  
               IPP_INCTL, 1);  
}  
  
IPAddr AllSPFRouters = {224, 0, 0, 5};
```



```
IPAddr AllDRouters = {224, 0, 0, 6};
```

Ospf_hsend calls ospfhtmpl to allocate a buffer and fill in header fields for a Hello packet. It then waits on semaphore if_nbmutex to guarantee exclusive access to the interface's neighbor table data structure. After obtaining exclusive use, ospf_hsend examines each entry in the list of neighbors for the interface. For each neighbor in state NBS_INTT or greater, ospf_hsend adds the neighbor to the Hello packet

Once a packet has been constructed, ospf_hsend computes the checksum and calls ipsend to send the packet. Whenever possible, OSPF uses IP multicast to send Hello packets; it transmits them to IP multicast address AllSPFRouters. When multicasting is unavailable, OSPF uses hardware broadcast to send Hello packers. Because our implementation supports multicasting on the primary network interface, ospf_hsend checks the interface to determine whether to multicast or broadcast an outgoing packet.

19.7.1 A Template For Hello Packets

Ospf_hsend calls ospfhtmpl to generate a Hello packet.

```
/* ospfhtmpl.c - ospfhtmpl */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*-----
 * ospfhtmpl - fill in OSPF HELLO packet template
 *-----
 */
struct ep *ospfhtmpl(pic)
struct ospf_if *pic;
{
    struct ep     *ep;
    struct ip     *ip;
    struct ospf    *po;
    struct ospf_hello *poh;

    ep = (struct ep *)getbuf(Net.netpool);
    if ((int)ep == SYSERR)
        return 0;
    ip = (struct ip *)ep->ep_data;
    po = (struct ospf *)ip->ip_data;
```



```
poh = (struct ospf_hello *)po->ospf_data;
po->ospf_version = OSPF_VERSION;
po->ospf_type = T_HELLO;
po->ospf_len = MINHELLOLEN;
po->ospf_rid = pif->if_rid;
po->ospf_aid = pif->if_area->ar_id;
po->ospf_authtype = pif->if_area->ar_authtype;
blkcopy(po->ospf_auth, pif->if_auth, AUTHLEN);
poh->oh_hintv = net2hs(pif->if_hintv);
poh->oh_opts = pif->if_opts;
poh->oh_rdintv = net2hl(pif->if_rdintv);
poh->oh_prio = pif->if_prio;
poh->oh_drid = pif->if_drid;
poh->oh_brid = pif->if_brid;
return pep;
}
```

Ospfhtmpl calls getbuf to allocate a buffer, and then fills in header fields for a Hello packet. Because a packet contains information for a specific interface, the caller must specify the interface as an argument. In the example code, argument pif contains a pointer to the interface structure from which ospfhtmpl extracts information for the header. For example, field if_hintv contains the Hello interval for the interface, which tells the length of time between transmission of Hello packets. Ospfhtmpl copies the data into field oh_hintv of the Hello packet.

19.7.2 The Hello Output Process

The Hello output process handles periodic transmission of Hello packets for all interfaces. File ospfhello.c contains the code.

```
/* ospfhello.c - ospfhello */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

#define OSPFDELTAT10 /* OSPF "stagger" bounds (1/10 secs) */

/*
 * ospfhello - send OSPF hello packets
 */
```



```
*/  
PROCESS ospfhello()  
{  
    struct ospf_if      *pif;  
    int      ifn, rnd;  
  
    /* select initial "random" offset to stagger hello's */  
    rnd = nif[NI_PRIMARY].ni_ip % OSPFDELTA;  
  
    /* do state changes first so we can receive while waiting */  
    for (ifn=0; ifn<Net.nif; ++ifn) {  
        if (ifn == NI_LOCAL)  
            continue;  
        if (nif[ifn].ni_state != NIS_UP)  
            continue;  
        pif = &ospf_if[ifn];  
        switch (pif->if_type) {  
        case IFT_MULTI:  
        case IFT_BROADCAST:  
            if (pif->if_prio > 0) {  
                pif->if_twait = pif->if_rdintv;  
                pif->if_state = IFS_WAITING;  
            } else  
                pif->if_state = IFS_DROTHER;  
            break;  
        case IFT_PT2PT:  
        case IFT_VIRTUAL:  
            pif->if_state = IFS_PT2PT;  
            break;  
        default:  
            break;  
        }  
    }  
  
    while (1) {  
        sleep10(HELLOINTV*10 + rnd);  
        if (++rnd == OSPFDELTA)  
            rnd = -OSPFDELTA;  
  
        for (ifn=0; ifn<Net.nif; ++ifn) {  
            if (ifn == NI_LOCAL)
```



```
        continue;

    if (nif[ifn].ni_state != NIS_UP)
        continue;
    ospf_hsend(ifn);
}
}
}
```

Ospfhello iterates through all network interfaces. For each interface, ospfhello examines the network type and assigns an interface state value^①. Ospfhello assigns the state of nonbroadcast networks the value IFS_PT2PT. For broadcast networks, ospfhello uses the configured priority, if_prio, to choose between interface states IFS_WAITING and IFS_DROTHER. The value IFS_DROTHER means the gateway is ineligible to become a designated router as defined in the next section.

After it initializes the interfaces, ospfhello enters an infinite loop. Each iteration of the loop delays for HELLOINTV seconds, and then calls ospf_hsend to send a Hello packet on each interface. To prevent synchronization among gateways, ospfhello adds a small pseudo-random value, rnd to the delay, and changes rnd for the next iteration.

19.8 Designated Router Concept

Although OSPF gateways flood link state information to other gateways along adjacencies, not all neighboring gateways become adjacent. In particular, to prevent unnecessary traffic on a broadcast network such as an Ethernet, the gateways attached to the network elect a single member of the set to serve as a designated router. Each gateway becomes adjacent to the designated router, but does not become adjacent to other gateways. Thus, only $N-1$ adjacencies form among N gateways attached to a given network,

In addition to serving as a focal point for adjacencies, a designated router connected to a network has one additional responsibility — it sends link state advertisements for the network. To understand the need for network advertisements, consider the graph in Figure 19.4. Several of the nodes represent networks (e.g., the node labeled N4). For shortest path computation to work correctly, a gateway must receive advertisements for all edges in the graph, including each edge that leads from a network to a gateway. Because all gateways on the network become adjacent to the designated router, it receives information about their connections to the network. Furthermore, the Hello protocol allows the designated router to determine whether other gateways remain

^① Note: the concept of an interface state for a local interface is independent of the neighbor states discussed earlier.



available. Thus, the designated router for a given network can send a link state update in which each item corresponds to an edge in the graph that leads from a node that represents a network to a node that represents a gateway.

19.9 Electing A Designated Router

Gateways on a multiaccess network must elect a designated router and a backup designated router. To do so, each gateway runs an election algorithm that examines the list of neighboring gateways that attach to the same network. To become a designated router, a neighbor must have established 2-way communication (i.e., be in state 2-Way or higher), and must have a nonzero priority. The election procedure can require more than one pass through the set of neighbors. After a single pass, if no designated router has been elected, the neighbor elected backup designated router becomes the designated router and the algorithm makes a second pass to elect a new backup designated router. Procedure `if_elect1` makes one pass through the list of neighbors, during which it chooses a designated router and a backup designated router.

```
/* if_elect1.c - if_elect1 */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * if_elect1 - make one neighbor list pass in the election algorithm
 */
int if_elect1(pif, ppdr, ppbr)
struct    ospf_if   *pif;
struct    ospf_nb   **ppdr;
struct    ospf_nb   **ppbr;
{
    unsigned long brid, drid;
    Bool      bdecl, cdecl;
    struct ospf_nb   *pnb, *pdr, *pbr;
    int       i;

    drid = pif->if_drid;
    brid = pif->if_brid;
    bdecl = FALSE;
    pdr = pbr = 0;
```



```
pnb = &pif->if_nbtab[0];
for (i=0; i<=MAXNBR; ++i, ++pnb) {
    if (pnb->nb_state < NBS_2WAY)
        continue;
    if (pnb->nb_rid == pnb->nb_drid) {
        /* neighbor claims designated routership */

        if (pdr) {
            if (pnb->nb_prio < pdr->nb_prio)
                continue;
            if ((pnb->nb_prio == pdr->nb_prio) &&
                (net2hl(pnb->nb_rid) <
                 net2hl(pdr->nb_rid)))
                continue;
        }
        pdr = pnb;
        continue;
    } /* else pnb is a backup candidate */

    cdecl = pnb->nb_rid == pnb->nb_brid;
    if (bdecl && !cdecl)
        continue;
    if (cdecl && !bdecl) {
        bdecl = TRUE;
        pbr = pnb;
        continue;
    }
    if (pbr == 0) {
        pbr = pnb;
        continue;
    }
    if (pnb->nb_prio < pbr->nb_prio)
        continue;
    if ((pnb->nb_prio == pbr->nb_prio) &&
        (pnb->nb_rid < pbr->nb_rid))
        continue;
    pbr = pnb; /* new backup */
}
*ppbr = pbr;
*ppdr = pdr;
}
```



To be elected designated router, a neighbor must be in state 2-Way or higher. As it iterates through the set of neighbors, if_elect1 checks field nb_state to determine the neighbor's state. It then checks to see whether the neighbor claims to be the designated router. If so, the neighbor's router ID in field pnb->nb_rid will be the same as the ID of the designated router reported by the neighbor in field pnb->nb_drid. If the neighbor claims to be the designated router and if_elect1 has another candidate designated router, the two must be compared. If_elect1 chooses the neighbor that advertised the highest priority; when multiple candidates have equal priority, it chooses the one with the higher router ID.

If if_elect1 finds a gateway ineligible to become the designated router, it considers the gateway for backup designated router. To be a candidate for backup designated router, the neighbor must have a nonzero priority and currently not be a designated router. Furthermore, if_elect1 selects candidates by priority. Among neighbors with equal priority, if_elect1 chooses the neighbor with the highest router ID.

Procedure if_elect makes one or two passes through the neighbors as needed to choose both a designated router and a backup designated router. If_elect also handles any changes that result.

```
/* if_elect.c - if_elect */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * if_elect - elect a designated router and backup designated router
 */
if_elect(ifn)
int ifn;
{
    struct ospf_if      *pif = &ospf_if[ifn];
    struct ospf_nb      *pnb, *pdr, *pbr;
    unsigned long        odrid, obrid;

    wait(pif->if_nbmutex);
    odrid = pif->if_drid;
    obrid = pif->if_brid;
    if_elect1(pif, &pdr, &pbr);
```



```
pif->if_brid = pbr ? pbr->nb_rid : 0;
if (pdr)
    pif->if_drid = pdr->nb_rid;
else {
    pif->if_drid = pif->if_brid;
    pif->if_brid = 0;
}
/*
 * if designate or backup has changed in this pass and this
 * router is old or new designate or backup, run again to get
 * a backup designate. Also update DB if we were/are designate
 */
if ((odrid != pif->if_drid &&
    (odrid == pif->if_rid || pif->if_drid == pif->if_rid))) {
    if_elect1(pif, &pdr, &pbr);
    pif->if_brid = pbr ? pbr->nb_rid : 0;
    db_resync(pif);
}
if ((obrid != pif->if_brid &&
    (obrid == pif->if_rid || pif->if_brid == pif->if_rid))) {
    if_elect1(pif, &pdr, &pbr);
    pif->if_brid = pbr ? pbr->nb_rid : 0;
}
signal(pif->if_nbmutex);
if (obrid != pif->if_brid || odrid != pif->if_drid) {
    if (pif->if_drid == pif->if_rid)
        pif->if_state = IFS_DR;
    else if (pif->if_brid == pif->if_rid)
        pif->if_state = IFS_BACKUP;
    else
        pif->if_state = IFS_DROTHER;
    nb_reform(pif);
}
}
```

When it begins, `if_elect` records the IDs of the old designated router and backup designated router. `If_elect` then calls `if_elect1`. Usually, `if_elect1` elects both a designated router and backup designated router. However, if no neighbors qualify for designated router, `if_elect1` will not choose a designated router. In such cases, `if_elect` promotes the backup designated router to the designated router position.



If the election changes the status of the gateway running the algorithm, it must run the election again. In particular, if a neighbor replaces the gateway as designated router, the gateway may have become eligible to serve as backup designated router. Thus, when it calls `if_elect1` the second time, a gateway that is no longer designated router will include itself in the set of candidates eligible to become the backup designated router.

19.10 Reforming Adjacencies After A Change

Each gateway on a network maintains an adjacency with the designated router and backup designated router. Before an election, `if_elect` saves the IDs of the current designated and backup routers. After the election, `if_elect` compares the new and old router IDs to see if they changed. If a change has occurred, `if_elect` records the IDs of the new designated and backup routers in the interface structure, and then calls `nb_reform` to reform adjacencies. File `nb_reform.c` contains the code.

```
/* nb_reform.c - nb_reform */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * nb_reform - reform adjacencies after a DR or BDR change
 */
int nb_reform(pif)
struct ospf_if *pif;
{
    struct ospf_nb     *pnb = &pif->if_nbtab[1];
    int      nn;

    for (nn=0; nn<MAXNBR; ++nn, ++pnb) {
        if (pnb->nb_state >= NBS_EXSTART) {
            if (!nb_aok(pif, pnb)) {
                /* break an adjacency */
                pnb->nb_state = NBS_2WAY;
                nb_clearl(pnb);
            }
        } else if (pnb->nb_state == NBS_2WAY) {
            if (nb_aok(pif, pnb)) {
                /* form an adjacency */
            }
        }
    }
}
```



```
    gettime(&pnb->nb_seq);
    pnb->nb_state = NBS_EXSTART;
    nb_makel(pnb);
    dd_queue(pif, pnb);
}
}
}
}
```

To recalculate adjacencies, nb_reform iterates through the set of neighbors. An adjacency exists, or is currently forming, with each neighbor that has state NBS_EXSTART or greater. Nb_reform calls nb_aok to check whether an adjacency should exist with a specific neighbor. If an existing adjacency has become invalid, nb_reform changes the neighbor's state to NBS_2WAY to break the adjacency. Nb_reform also calls nb_clearl to remove link state advertisements pending transmission to that neighbor. If an adjacency should exist for a neighbor that was previously in state NBS_2WAY, nh_reform moves the neighbor to state NBS_EXSTART, calls nb_makel to create lists on which it will place outgoing link state messages and retransmissions. As the last step of initializing an adjacency, nb_reform calls dd_queue to start sending the database of topology information to the neighbor. The code for nb_clearl and nb_makel follows; a later section discusses dd_queue.

```
/* nb_clearl.c - nb_clearl */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*-----
 * nb_clearl - clear lists of pending messages for a given neighbor
 *-----
 */
int nb_clearl(pnb)
struct    ospf_nb    *pnb;
{
    struct ep *pep;

    while (pep = (struct ep *)deq(pnb->nb_lsal))
        freebuf(pep);
    freeq(pnb->nb_lsal);
    pnb->nb_lsal = EMPTY;
```



```
while (pep = (struct ep *)deq(pnb->nb_dsl))
    freebuf(pep);
freeq(pnb->nb_dsl);
pnb->nb_dsl = EMPTY;

while (pep = (struct ep *)deq(pnb->nb_lsrl))
    freebuf(pep);
freeq(pnb->nb_lsrl);
pnb->nb_lsrl = EMPTY;
}

/* nb_makel.c - nb_makel */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>
#include <q.h>

/*
 * nb_makel - create a new adjacency's retransmission lists
 */
int nb_makel(pnb)
struct    ospf_nb   *pnb;
{
    pnb->nb_lsal = newq(NBMAXLIST, QF_WAIT);
    pnb->nb_dsl = newq(NBMAXLIST, QF_WAIT);
    pnb->nb_lsrl = newq(NBMAXLIST, QF_WAIT);
}
```

19.11 Handling Arriving Hello Packets

When a Hello message arrives, OSPF calls procedure `ospf_hin` to handle it.

```
/* ospf_hin.c - ospf_hin */
```

```
#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>
```



```
struct ospf_nb *nb_add(struct ospf_if *, struct ospf *);
int nb_switch(struct ospf_if *, struct ospf_nb *, struct ep *);

/*-----
 * ospf_hin - handled input of OSPF HELLO packets
 *-----
 */
int
ospf_hin(pep)
struct    ep    *pep
{
    struct ospf_if          *pif = &ospf_if[pep->ep_ifn];
    struct ospf_nb         *pnb, *nb_add();
    struct ip      *pip;
    struct ospf      *po;
    struct ospf_hello *poh;

    pip = (struct ip *)pep->ep_data;
    po = (struct ospf *)((char *)pip + IP_HLEN(pip));
    poh = (struct ospf_hello *)po->ospf_data;

    if (poh->oh_hintv != pif->if_hintv ||
        poh->oh_rdintv != pif->if_rdintv)
        return;
    if (po->ospf_rid == pif->if_rid)
        return; /* one of our own packets */

    pnb = nb_add(pif, po);
    if (pnb == 0)
        return; /* neighbor list overflowed */
    blkcopy(pnb->nb_ipa = pip->ip_src, IP_ALEN);
    if (nb_switch(pif, pnb, pep) == 0)
        return;
    if (poh->oh_prio != pnb->nb_prio) {
        pif->if_event |= IFE_NCHNG;
        pnb->nb_prio = poh->oh_prio;
    }
    if (poh->oh_drid == pnb->nb_rid) { /* Neighbor claims DR */
        if (poh->oh_brid == 0 && pif->if_state == IFS_WAITING)
```



```
    pif->if_event |= IFE_BSEEN;
    else if (pnb->nb_drid != pnb->nb_rid)
        pif->if_event |= IFE_NCHNG;
} else if (pnb->nb_drid == pnb->nb_rid)
    pif->if_event |= IFE_NCHNG;
pnb->nb_drid = poh->oh_drid;
if (poh->oh_brid == pnb->nb_rid) {
    if (pif->if_state == IFS_WAITING)
        pif->if_event |= IFE_BSEEN;
    else if (pnb->nb_brid != pnb->nb_rid)
        pif->if_event |= IFE_NCHNG;
} else if (pnb->nb_brid == pnb->nb_rid)
    pif->if_event |= IFE_NCHNG;
pnb->nb_brid = poh->oh_brid;
}
```

A Hello message can be ignored if either of the neighbor's values for the Hello interval or the neighbor's value for the router dead interval disagrees with the corresponding value in the gateway, or if the message was originally sent from the gateway itself. If the message is valid, ospf_hin calls nb_add to find the sender to the list of neighboring gateways. Ospf_hin then implements the semantics of state transitions according to the standard.

19.12 Adding A Gateway To The Neighbor List

Procedure nb_add adds a neighbor to the list of neighbors.

```
/* nb_add.c - nb_add */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * nb_add - add a neighbor to our neighbor list (update timer if present)
 */
struct ospf_nb *nb_add(PIF pif, PO po)
{
    OSPF_IF *pif;
    OSPF *po;
```



```
{  
    struct ospf_hello *poh = (struct ospf_hello *)po->ospf_data;  
    struct ospf_nb          *pnb, *pnbfree;  
    int             nn;  
  
    wait(pif->if_nbmutex);  
    pnb = &pif->if_nbtab[1];  
    pnbfree = 0;  
    for (nn=0; nn<MAXNBR; ++nn, ++pnb) {  
        if (pnbfree == 0 && pnb->nb_state == NBS_DOWN) {  
            pnbfree = pnb;  
            continue;  
        }  
        if (pnb->nb_rid == po->ospf_rid)  
            break;  
    }  
    if (nn >= MAXNBR)  
        pnb = pnbfree;  
    if (pnb) {  
        if (pnb->nb_state < NBS_INIT) {  
            pnb->nb_rid = po->ospf_rid;  
            pnb->nb_prio = 0;  
            pnb->nb_drid = 0;  
            pnb->nb_brid = 0;  
            pnb->nb_lsal = pnb->nb_dsl = EMPTY;  
            pnb->nb_lsrl = EMPTY;  
            pnb->nb_state = NBS_INIT;  
            pnb->nb_tlastdd = 0;  
        }  
        pnb->nb_lastheard = pif->if_rdintv;  
    }  
    signal(pif->if_nbmutex);  
    return pnb;  
}
```

After obtaining exclusive use of the interface's neighbor table, nb_add searches the list of neighbors sequentially. It records the location of the first unused entry in variable pnbfree, and terminates the search if it encounters the item to be added in the list. If the search results in a new entry, nb_add initializes the state of the entry to NBS_INIT, and fills in other fields.



19.13 Neighbor State Transitions

For each combination of event and neighbor state, the OSPF standard specifies the semantic action as well as a state transition. Events include actions initiated by the local gateway as well as the arrival of messages. Procedure nb_switch handles several actions that can occur when a Hello message arrives.

```
/* nb_switch.c - nb_switch */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * nb_switch - handle neighbor state changes on HELLO packet input
 */
int nb_switch(pif, pnb, pep)
struct ospf_if *pif;
struct ospf_nb *pnb;
struct ep *pep;
{
    struct ip     *pip;
    struct ospf     *po;
    struct ospf_hello *poh;
    Bool         found = FALSE;
    int          nn, maxn;

    pip = (struct ip *)pep->ep_data;
    po = (struct ospf *)((char *)pip + IP_HLEN(pip));
    poh = (struct ospf_hello *)po->ospf_data;

    maxn = (po->ospf_len - MINHELLOLEN) / sizeof(long);
    for (nn=0; nn<maxn; ++nn)
        if (found = (pif->if_rid == poh->oh_neighbor[nn]))
            break;
    if (!found) {
        if (pnb->nb_state >= NBS_2WAY) {
            pnb->nb_state = NBS_INIT;
            nb_clearl(pnb);
        }
    }
}
```



```
        return 0;
    } else if (pnb->nb_state == NBS_INIT) {
        pnb->nb_state = NBS_2WAY;
        if (nb_aok(pif, pnb)) {
            gettime(&pnb->nb_seq); /* set initial seq */
            pnb->nb_state = NBS_EXSTART;
            nb_makel(pnb);
            dd_queue(pif, pnb);
        }
    }
    return 1;
}
```

After computing the number of neighbors advertised in the arriving message, nb_switch iterates through the list. If the receiving gateway does not find itself listed in the advertisement, the neighbor does not maintain 2-way communication. Consequently, the gateway changes the neighbor to state NBS_INIT. If the receiving gateway finds itself listed in the advertisement and 2-way communication was not established before, it moves to state NBS_2WAY, and checks to see whether it should negotiate an adjacency.

19.14 OSPF Timer Events And Retransmissions

The OSPF protocol uses timers to periodically send Hello messages, to timeout retransmissions, and to make transitions among neighbor states when no response has been received. The code arranges to call procedure ospftimer periodically to handle timed events.

```
/* ospftimer.c - ospftimer */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * ospftimer - update neighbor time-out values
 */
int
ospftimer(delay)
    int delay;
```



```
{  
    struct ospf_if      *pif;  
    struct ospf_nb      *pnb;  
    int      ifn, i;  
  
    pif = &ospf_if[0];  
    for (ifn=0; ifn<NIF; ++ifn, ++pif) {  
        switch (pif->if_state) {  
            case IFS_DOWN:  
                continue;  
            case IFS_WAITING:  
                pif->if_twait -= delay;  
                if (pif->if_twait <= 0)  
                    if_elect(ifn);  
                break;  
            default:  
                break;  
        }  
        wait(pif->if_nbmutex);  
        pnb = &ospf_if[ifn].if_nbtab[1];  
        for (i=0; i<MAXNBR; ++i, ++pnb) {  
            if (pnb->nb_state == NBS_DOWN)  
                continue;  
            pnb->nb_lastheard -= delay;  
            if (pnb->nb_lastheard <= 0) {  
                pnb->nb_state = NBS_DOWN;  
                pif->if_event |= IFE_NCHNG;  
            }  
            if (pnb->nb_state == NBS_EXSTART) {  
                pnb->nb_trexmt -= delay;  
                if (pnb->nb_trexmt <= 0)  
                    nb_rexmt(pif, pnb);  
            }  
            if (pnb->nb_state >= NBS_FULL &&  
                pnb->nb_tlastdd > 0) {  
                pnb->nb_tlastdd -= delay;  
                if (pnb->nb_tlastdd < 0)  
                    freebuf(deq(pnb->nb_dsl));  
            }  
            if (headq(pnb->nb_lsrl)) {  
                pnb->nb_tlsr -= delay;  
            }  
        }  
    }  
}
```



```
        if (pnb->nb_tlsr <= 0)
            lsr_xmit(pif, pnb);
    }
}

signal(pif->if_nbmutex);

if (pif->if_event & IFE_NCHNG) {
    if_elect(ifn);
    pif->if_event &= ~IFE_NCHNG;
}
}

}
```

In addition to handling other timed events, ospftimer calls nb_rexmt to retransmit data description messages in state NBS_EXSTART. The retransmission, combined with packet sequencing, ensures reliable synchronization of two gateways' topological databases when they form an adjacency.

```
/* nb_rexmt.c - nb_rexmt */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * nb_rexmt - handle neighbor retransmit timer event
 */
int nb_rexmt(pif, pnb)
struct    ospf_if    *pif;
struct    ospf_nb    *pnb;
{
    if (pnb->nb_state == NBS_EXSTART) {
        dd_xmit(pif, pnb);
        pnb->nb_trexmt = pif->if_rintv;
    }
}
```

19.15 Determining Whether Adjacency Is Permitted

Procedure nb_aok determines whether a gateway should become adjacent with a



neighbor.

```
/* nb_aok.c - nb_aok */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * nb_aok - determine if adjacency with a neighbor is ok
 */
int nb_aok(pif, pnb)
struct    ospf_if   *pif;
struct    ospf_nb   *pnb;
{
    if (pif->if_type == IFT_PT2PT || pif->if_type == IFT_VIRTUAL)
        return TRUE;
    if (pif->if_drid == pif->if_rid || pif->if_brid == pif->if_rid)
        return TRUE;
    if (pif->if_drid == pnb->nb_rid || pif->if_brid == pnb->nb_rid)
        return TRUE;
    return FALSE;
}
```

A gateway always establishes adjacency with any neighbor that is connected by a point-to-point link or a virtual link. For multiaccess networks, the designated router or backup designated router establishes adjacency with all neighbors. Other routers can only become adjacent with a neighbor that is a designated router or a backup designated router. The code checks for each case and returns TRUE to allow adjacency or FALSE to deny it.

19.16 Handling OSPF input

When an IP datagram arrives carrying an OSPF packet, IP calls procedure ospf_in to handle it.

```
/* ospf_in.c - ospf_in */

#include <conf.h>
#include <kernel.h>
```



```
#include <network.h>
#include <ospf.h>

/*
 * ospf_in - deliver an inbound OSPF packet to the OSPF input process
 */
int ospf_in(pni, pep)
struct netif *pni;
struct ep *pep;
{
    /* drop instead of blocking on psend */

    if (pcount(ospf_iport) >= OSPFQLEN) {
        freebuf(pep);
        return SYSERR;
    }
    psend(ospf_iport, (int)pep);
    return OK;
}
```

In our system, the interrupt handler calls `ospf_in` when a packet arrives carrying OSPF. `Ospf_in` calls `psend` to enqueue the packet on port `ospf_iport`, where the OSPF input process finds it. Because procedures called from an interrupt handler cannot block, `ospf_in` must not call `psend` if the port is full. To avoid blocking, `ospf_in` calls `pcount`, and discards the packet if the port is full.

The OSPF input process, which extracts packets from port `ospf_iport`, executes procedure `ospf`.

```
/* ospf.c - ospf */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <proc.h>
#include <ospf.h>

/*
 * ospf - start OSPF processes and become OSPF input process
 */

```



```
PROCESS ospf()
{
    struct    ep    *pep;
    struct    ip    *pip;
    struct    ospf  *po;
    struct    ospf_if  *pif;

    if (!gateway)
        return SYSERR;
    ospfinit();
    ospf_iport = pcreate(OSPFQLEN);
    if (ospf_iport == SYSERR)
        panic("ospf: cannot open ospf port");
    resume(create(ospfhello, OSPFHSTK, OSPFHPRI, OSPFHNAME, OSPFHARGC));

    while (TRUE) {
        pep = (struct ep *)preceive(ospf_iport);
        if (pep == (struct ep *)SYSERR)
            continue;
        pip = (struct ip *)pep->ep_data;
        po = (struct ospf *)((char *)pip + IP_HLEN(pip));
        ospfnet2h(po);
        if (ospfcheck(pep) != OK)
            continue;
        pep->ep_order |= EPO_OSPF;
        switch (po->ospf_type) {
        case T_HELLO:      ospf_hin(pep);
                            break;
        case T_DATADESC:   ospf_ddin(pep);
                            break;
        case T_LSREQ:      lsr_in(pep);
                            break;
        case T_LSUPDATE:   lsu_in(pep);
                            break;
        case T_LSACK:      lsack_in(pep);
                            break;
        default:
                            break;
        }
        pif = &ospf_if[pep->ep_ifn];
        if ((pif->if_state == IFS_WAITING &&
```



```
(pif->if_event&IFE_BSEEN)) ||  
(pif->if_state >= IFS_DROTHER &&  
(pif->if_event&IFE_NCHNG))) {  
    if_elect(pep->ep_ifn);  
    pif->if_event &= ~(IFE_BSEEN|IFE_NCHNG);  
}  
freebuf(pep);  
}  
}  
  
Bool doospf = FALSE;  
int ospfpid = BADPID;  
int ospf_iport;
```

Ospf runs as a separate process. It begins by calling ospfinit to initialize all data structures and pcreate to create a synchronized queue for incoming packets. Ospf also creates a process to generate Hello packets, and then enters an infinite loop.

On each iteration of its main loop, ospf extracts one incoming OSPF message, calls ospfnet2h to convert it to the local byte order, and ospfcheck to verify that the packet is valid. Ospf then examines field ospf_type to determine the type of the packet, and calls an appropriate input routine. Finally, ospf examines the interface over which the packet arrived to determine whether the arrival of the message should trigger election of a new designated router.

19.17 Declarations And Procedures For Link State Processing

File ospf_ls.h contains declarations of packet contents, structures, and constants related to link state messages.

```
/* ospf_ls.h */  
  
/* OSPF Link State Request Packet */  
  
struct ospf_lsr {  
    unsigned long lsr_type; /* Link State Type */  
    unsigned long lsr_lsid; /* Link State Identifier */  
    unsigned long lsr_rid; /* Advertising Router */  
};  
  
#define LSRLEN 12
```



```
/* OSPF Link State Summary */

struct ospf_lss {
    unsigned short lss_age; /* Time (secs) Since Originated */
    unsigned char lss_opts; /* Options Supported */
    unsigned char lss_type; /* LST_* below */
    unsigned long lss_lsid; /* Link State Identifier */
    unsigned long lss_rid; /* Advertising Router Identifier*/
    unsigned long lss_seq; /* Link State Adv. Sequence # */
    unsigned short lss_cksum; /* Fletcher Checksum of LSA */
    unsigned short lss_len; /* Length of Advertisement */
};

#define LSSHDRLEN 20

/* Link State Advertisement Types */

#define LST_RLINK 1      /* Router Link */
#define LST_NLINK 2      /* Network Link */
#define LST_SLINK 3      /* IP Network Summary Link */
#define LST_BRSLINK 4    /* AS Border Router Summary */
#define LST_EXTERN 5      /* AS External Link */

/* Link State Advertisement (min) Lengths */

#define LSA_RLEN (LSSHDRLEN + 4)
#define LSA_NLEN (LSSHDRLEN + 4)

#define LSA_ISEQ 0x80000001

/* LSS Type of Service Entry */

struct tosent {
    unsigned char tos_tos; /* IP Type of Service */
    unsigned char tos_mbz; /* Must Be Zero */
    unsigned short tos_metric; /* Metric for This TOS */
};

/* OSPF Link State Advertisement */

#define MAXLSDLEN 64 /* Max LS Data Len (configurable)
```



```
struct ospf_lsa {
    struct ospf_lss    lsa_lss; /* Link State Adv. Header */
    char      lsa_data[MAXLSDLEN]; /* Link-Type Dependent Data*/
};

/* Convenient Field Translations */

#define  lsa_age        lsa_lss.lss_age
#define  lsa_opts       lsa_lss.lss_opts
#define  lsa_type       lsa_lss.lss_type
#define  lsa_lsid      lsa_lss.lss_lsid
#define  lsa_rid        lsa_lss.lss_rid
#define  lsa_seq        lsa_lss.lss_seq
#define  lsa_cksum     lsa_lss.lss_cksum
#define  lsa_len        lsa_lss.lss_len

/* Router Links Advertisement */

struct  ospf_ra {
    unsigned char ra_opts; /* RAO_* Below */
    unsigned char ra_mbz;   /* Must Be Zero */
    unsigned short  ra_nlinks; /* # of Links This Advertisement*/
    unsigned long   ra_lid;  /* Link ID */
    unsigned long   ra_ipa;  /* Router Interface IP Address */
    unsigned char   ra_type; /* Link Type (RAT_* Below) */
    unsigned char   ra_ntos; /* # of Types-of-Service Entries*/
    unsigned char   ra_metric; /* TOS 0 Metric */
    unsigned long  ra_tosend[1]; /* TOS Entries ra_ntos Times */
};

define  ra_mask        ra_lid      /* For Stub Networks */
#define  RAO_ABR        0x01      /* Router is Area Border Router */
#define  RAO_EXTERN     0x02      /* Router is AS Boundary Router */

#define  RAT_PT2PT1     /* Point-Point Connection */
#define  RAT_TRANSIT    2         /* Connection to Transit Network*/
#define  RAT_STUB       3         /* Connection to Stub Network */
#define  RAT_VIRTUAL    4         /* Virtual Link */

/* Network Links Advertisement */
```



```
struct    ospf_na {
    IPAddr          na_mask; /* Network Mask           */
    unsigned long   na_rid[1]; /* IDs of All Attached Routers */
};

/* Link State Update Packet Format */

struct    ospf_lsu {
    unsigned long   lsu_nads; /* # Advertisements This Packet */
    char      lsu_data[1]; /* 1 or more struct ospf_lsa's */
};

#define  MINLSULEN(MINHDRLEN + 4) /* Base LSU Length           */
```

19.18 Generating Database Description Packets

When a gateway first initiates an adjacency with a neighbor, it places the neighbor in state NBS_EXSTART and begins exchanging database description packets. The exchange continues after the neighbor moves to state NBS_EXCHNG.

Whenever OSPF software needs to generate a database description, it calls procedure dd_queue.

```
/* dd_queue.c - dd_queue */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * dd_queue - generate Data Description packets
 */
int dd_queue(pif, pnb)
struct    ospf_if  pif;
struct    ospf_nb  *pnb;
{
    struct    ep    *pep;
    struct    ip    *pip;
    struct    ospf  *po;
    struct    ospf_dd *pdd;
```



```
    pep = ospfddtmp1(pif);
    if (pep == 0)
        return;
    pip = (struct ip *)pep->ep_data;
    po = (struct ospf *)pip->ip_data;
    pdd = (struct ospf_dd *)po->ospf_data;

    if (pnb->nb_state == NBS_EXSTART) {
        pdd->dd_control = DDC_INIT | DDC_MORE | DDC_MSTR;
        pdd->dd_seq = hl2net(pnb->nb_seq);
        if (enq(pnb->nb_dsl, pep, 0) < 0)
            freebuf(pep);
        dd_xmit(pif, pnb);
        pnb->nb_trexmt = pif->if_rintv;
        return;
    }
    /* else we're in EXCHANGE state */
    lss_build(pif, pnb, pep);
    dd_xmit(pif, pnb);
    if (pnb->nb_master)
        pnb->nb_trexmt = pif->if_rintv;
}
```

Dd_queue calls ospfddtmp1 to allocate a buffer and fill in header fields of the packet. It then examines field nb_state in the neighbor records to determine the neighbor's state. If the neighbor is in state NBS_EXSTART, dd_queue creates an initial packet and enqueues it for transmission. The initial packet has bits DDC_MORE, DDC_MSTR, and DDC_INIT set in field dd_control to tell the neighbor that the packet is the initial database description packet, and to cause the neighbor to negotiate which gateway will serve as master. Ddqueue calls enq to place the initial packet on the database summary list, and then calls dd_xmit to send the packet. Finally, before returning to its caller, dd_queue assigns field nb_trexmt to schedule a retransmission.

After a neighbor acknowledges the initial packet, a gateway changes the neighbors state to NBS_EXCHNG and records which of the two is master. For a neighbor in state NBS_EXCHNG, dd_queue creates and sends a database summary. Dd_queue calls lss_build to construct a summary of its topological database and enqueue it for transmission, and dd_xmit to send the summary. If the sender is the masters, dd_queue schedules a retransmission before returning to its caller.



19.19 Creating A Template

When OSPF needs to create a database description packet, it calls ospfddtmpl to allocate a buffer and fill in the header fields.

```
/* ospfddtmpl.c - ospfddtmpl */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * ospfddtmpl - fill in OSPF Data Description packet template
 */
struct ep *ospfddtmpl(pif)
struct    ospf_if   *pif;
{
    struct ep *pep;
    struct ip *pip;
    struct ospf   *po;
    struct ospf_dd      *pdd;

    pep = (struct ep *)getbuf(Net.netpool);
    if ((int)pep == SYSERR)
        return 0;
    pip = (struct ip *)pep->ep_data;
    po = (struct ospf *)pip->ip_data;
    pdd = (struct ospf_dd *)po->ospf_data;

    po->ospf_version = OSPF_VERSION;
    po->ospf_type = T_DATADESC;
    po->ospf_len = MINDDLEN;
    po->ospf_rid = pif->if_rid;
    po->ospf_aid = pif->if_area->ar_id;
    po->ospf_authtype = pif->if_area->ar_authtype;
    memcpy(po->ospf_auth, pif->if_auth, AUTHLEN);
    pdd->dd_mbz = 0;
    pdd->dd_opts = pif->if_opts;
    pdd->dd_control = 0;
    return pep;
```



}

Ospfddtmpl receives a pointer to the structure for the interface as an argument. It allocates a buffer, and then proceeds to fill in header fields. Fields that contain constants can be assigned symbolic variables (e.g., the OSPF version and packet type). Information for other fields must be extracted from the interface structure (e.g., the IDs of the sending router and the area). Once ospfddtmpl fills in the header, it returns a pointer to the packet.

19.20 Transmitting A Database Description Packet

When two gateways exchange information from their topological databases, they send database description packets and acknowledgements. Procedure dd_xmit transmits both. It sends a database description packet from the database summary list or creates an acknowledgement if the list is empty.

```
/* dd_xmit.c - dd_xmit */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * dd_xmit - transmit pending Database Description packets
 */
int dd_xmit(pif, pnb)
struct    ospf_if   *pif;
struct    ospf_nb   *pnb;
{
    struct    ep     *pephd, *pep;
    struct    ip     *pip;
    struct    ospf   *po;
    struct    ospf_dd *pdd;
    int       len;

    if (pephd = (struct ep *)headq(pnb->nb_dsl)) {
        pep = (struct ep *)getbuf(Net.netpool);
        if ((int)pep == SYSERR)
            return SYSERR;
```



```
/* make a copy */
pip = (struct ip *)pephdr->ep_data;
po = (struct ospf *)pip->ip_data;
len = EP_HLEN + IPMHLEN + po->ospf_len;
blkcopy(pephdr, pephead, len);
pip = (struct ip *)pephdr->ep_data;
po = (struct ospf *)pip->ip_data;
pdd = (struct ospf_dd *)po->ospf_data;
} else {
    /* no DD's to send; create an ACK-only */
    pep = ospfddtmp(pif);
    pip = (struct ip *)pephdr->ep_data;
    po = (struct ospf *)pip->ip_data;
    pdd = (struct ospf_dd *)po->ospf_data;
    if (pnb->nb_master)
        pdd->dd_control = DDC_MSTR;
    else
        pdd->dd_control = 0;
}
pdd->dd_seq = pnb->nb_seq;
po->ospf_authtype = pif->if_area->ar_authtype;
bzero(po->ospf_auth, UATHLEN);
po->ospf_cksum = 0;
po->ospf_cksum = cksum(po, po->ospf_len>>1);
blkcopy(po->ospf_auth, pif->if_area->ar_auth, AUTHLEN);
blkcopy(pip->ip_src, nif[ifn].ni_ip, IP_ALEN);
if (ifn == NI_PRIMARY)
    ipsend(AllSPFRouters, pep, po->ospf_len, IPT_OSPF,
           IPP_INCTL, 1);
else
    ipsend(nif[ifn].ni_brc, pep, po->ospf_len, IPT_OSPF,
           IPP_INCTL, 1);
}
```

Dd_xmit begins by calling headq to examine the first item on the database summary list, field nb_dsl. If the list contains an item, headq returns a pointer to the item; otherwise, it returns zero. If the call to headq finds an item, dd_xmit allocates a buffer to hold an outgoing packet, and copies information from the first item on the list into the packet. If it finds the list empty, dd_xmit calls ospfddtmp to create a packet that contains only an acknowledgement.



After creating a packet, dd_xmit proceeds to fill in the sequence number in field dd_seq, and the authentication type in field ospf_authtype of the OSPF header. OSPF does not include authentication information in the checksum. Therefore, dd_xmit fills in zeroes for the checksum and authentication fields before computing the checksum. Dd_xmit then moves the computed checksum and authentication information into the header, and calls ipsend to transmit the resulting datagram.

19.21 Handling An Arriving Database Description Packet

When two gateways establish an adjacency, they exchange topology information. The gateway with the larger router ID becomes the master, and polls the slave to obtain database information. To handle an incoming database description packet, OSPF calls procedure ospf_ddin.

```
/* ospf_ddin.c - ospf_ddin */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * ospf_ddin - handled input of OSPF Data Description packets
 */
int
ospf_ddin(pep)
struct ep *pep;
{
    struct ospf_if      *pif = &ospf_if[pep->ep_ifn];
    struct ospf_nb      *pnb, *nb_add();
    struct ip *pip;
    struct ospf      *po;

    pip = (struct ip *)pep->ep_data;
    po = (struct ospf *)((char *)pip + IP_HLEN(pip));

    pnb = nb_add(pif, po);
    if (pnb == 0)
        return;
    switch (pnb->nb_state) {
    case NBS_INIT:
```



```
pnb->nb_state = NBS_2WAY;
if (nb_aok(pif, pnb)) {
    pnb->nb_seq++;
    pnb->nb_state = NBS_EXSTART;
    nb_makel(pnb);
    dd_queue(pif, pnb);
}
break;

case NBS_2WAY:
    return 0;
case NBS_EXSTART:
    ddi_exstart(pif, pnb, pep);
    break;
case NBS_EXCHNG:
    ddi_exchng(pif, pnb, pep);
    break;
case NBS_LOADING:
case NBS_FULL:
    ddi_full(pif, pnb, pep);
    break;
}
}
```

Ospf_ddin uses the neighbor's state to determine how to proceed. For neighbors in states less than NBS_2WAY it must determine whether an adjacency is allowed before initiating one. For neighbors in a state greater than NBS_2WAY, ospf_ddin calls an appropriate procedure to handle the packet that has arrived.

19.21.1 Handling A Packet In The EXSTART State

Procedure ddi_exstart handles packets that arrive from a neighbor which currently has state NBS_EXSTART.

```
/* ddi_exstart.c - ddi_exstart */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
-----*
 * ddi_exstart - handle Data Descrip. input for EXSTART state neighbors
```



```
*-----  
*/  
  
int ddi_exstart(pif, pnb, pep)  
struct    ospf_if   *pif;  
struct    ospf_nb   *pnb;  
struct    ep     *pep;  
{  
    struct ip*pip = (struct ip *)pep->ep_data;  
    struct ospf   *po = (struct ospf *)((char *)pip + IP_HLEN(pip));  
    struct ospf_dd   *pdd = (struct ospf_dd *)po->ospf_data;  
    struct ep*peptmp;  
    unsigned int  cbits = DDC_INIT | DDC_MORE | DDC_MSTR;  
  
    if (((pdd->dd_control & cbits) == cbits) &&  
        po->ospf_len == MINDDLEN &&  
        pnb->nb_rid > pif->if_rid) {  
        pnb->nb_master = FALSE;  
        pnb->nb_seq = pdd->dd_seq;  
        pnb->nb_opts = pdd->dd_opts;  
        pnb->nb_state = NBS_EXCHNG;  
    } else if (((pdd->dd_control&(DDC_INIT|DDC_MSTR)) == 0) &&  
        pdd->dd_seq == pnb->nb_seq &&  
        pnb->nb_rid < pif->if_rid) {  
        pnb->nb_master = TRUE;  
        pnb->nb_opts = pdd->dd_opts;  
        pnb->nb_state = NBS_EXCHNG;  
    } else  
        return;  
    if (peptmp = (struct ep *)deq(pnb->nb_ds1))  
        freebuf(peptmp);  
    if (pnb->nb_master)  
        pnb->nb_seq++;  
    lsr_queue(pif, pnb, pep);  
    dd_queue(pif, pnb);  
}
```

Ddi_exstart examines the size of the incoming message as well as the control bits in the message. If a neighbor sends a minimum size packet and the neighbor's ID is larger than the receiving gateway's ID, the receiver moves the neighbor to state NBS_EXCHNG and declares the neighbor the master. Otherwise, if the sequence number on the incoming database description packet matches the initial sequence



number, ddi_exstart accepts the packet, moves the neighbor to state NBS_EXCHNG, and declares itself master. If the neighbor changes state, ddi_exstart calls dd_queue Co enqueue database description packets for transmission.

19.21.2 Handling A Packet In The EXCHNG State

Ospf_ddin calls procedure ddi_exchng to handle a database description packet that arrives from a neighbor in state NBS_EXCHNG.

```
/* ddi_exchng.c - ddi_exchng */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*-----
 * ddi_exchng - handle Data Descrip. input for EXCHANGE state neighbors
 *-----
 */

int ddi_exchng(pif, pnb, pep)
struct    ospf_if   *pif;
struct    ospf_nb   *pnb;
struct    ep     *pep;
{
    struct ip*pip = (struct ip *)pep->ep_data;
    struct ospf   *po = (struct ospf *)((char *)pip + IP_HLEN(pip));
    struct ospf_dd      *pdd = (struct ospf_dd *)po->ospf_data;
    struct ep*peptmp;

    if (((pdd->dd_control & DDC_MSTR) & pnb->nb_master) ||
        (! (pdd->dd_control & DDC_MSTR) & !pnb->nb_master) ||
        (pdd->dd_control & DDC_INIT) ||
        (pdd->dd_opts != pnb->nb_opts))
        return nb_mismatch(pif, pnb);

    if ((pnb->nb_master && pdd->dd_seq == pnb->nb_seq-1) ||
        (!pnb->nb_master && pdd->dd_seq == pnb->nb_seq))
        return 0; /* duplicate */

    if ((!pnb->nb_master && pdd->dd_seq != pnb->nb_seq + 1) ||
        (pnb->nb_master && pdd->dd_seq != pnb->nb_seq))
        return nb_mismatch(pif, pnb);

    if (!pnb->nb_master && lenq(pnb->nb_dsl) == 1) {
        /* slave must save the last packet DEADINTV secs */
    }
}
```



```
    if (pnb->nb_tlastdd == 0)
        pnb->nb_tlastdd = DEADINTV;
    } else if (peptmp = (struct ep *)deq(pnb->nb_dsl))
        freebuf(peptmp); /* has been acked */
    if (pnb->nb_master)
        pnb->nb_seq++;
    else
        pnb->nb_seq = pdd->dd_seq;
    lsr_queue(pif, pnb, pep);
    if ((pdd->dd_control & DDC_MORE) == 0 &&
        leng(pnb->nb_dsl) <= 1) {
        if (headq(pnb->nb_lsrl))
            pnb->nb_state = NBS_LOADING;
        else
            pnb->nb_state = NBS_FULL;
        if (!pnb->nb_master)
            dd_xmit(pif, pnb);
    }
    return 1;
}
dd_xmit(pif, pnb);
}
```

As specified in the standard, ddi_exchng checks the sequence number of the arriving packet. A master accepts a packet that has sequence number equal to the last sequence number the master has sent; a slave accepts a packet that has a sequence number one greater than the last sequence number the slave has sent. A packet with a lower sequence number is a duplicate. The master discards duplicates, but a slave responds by retransmitting the last packet sent. When accepting a packet, a master increments the sequence counter in field nb_seq; a slave makes the sequence number equal to that of the accepted packet.

19.21.3 Handling A. Packet In The FULL State

When ospf_ddin receives a database description packet from a neighbor that currently has state NBS_FULL, it calls procedure ddi_full to handle the packet.

```
/* ddi_full.c - ddi_full */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>
```



```
/*
 * ddi_full - handle DD packet input for FULL and LOADING state neighbors
 */
int ddi_full(pif, pnb, pep)
struct ospf_if *pif;
struct ospf_nb *pnb;
struct ep *pep;
{
    struct ip*pip = (struct ip *)pep->ep_data;
    struct ospf *po = (struct ospf *)((char *)pip + IP_HLEN(pip));
    struct ospf_dd *pdd = (struct ospf_dd *)po->ospf_data;

    if (((pdd->dd_control & DDC_MSTR) & pnb->nb_master) ||
        (!(pdd->dd_control & DDC_MSTR) & !pnb->nb_master) ||
        (pdd->dd_control & DDC_INIT) ||
        (pdd->dd_opts != pnb->nb_opts))
        return nb_mismatch(pif, pnb);
    if (pnb->nb_master && pdd->dd_seq == pnb->nb_seq-1)
        return 0; /* duplicate; master discards */
    if (!pnb->nb_master && pdd->dd_seq == pnb->nb_seq) {
        dd_xmit(pif, pnb); /* duplicate; slave responds */
        return 1;
    }
    /* else, NOT a duplicate; Sequence mismatch */
    return nb_mismatch(pif, pnb);
}
```

When a packet arrives, `ddi_full` compares the master control bit in the packet to field `nb_master` in the local data structure. If the neighbor's assertion of master status disagrees with the receiver's record, `ddi_full` calls `nb_mismatch` to restart the adjacency from the beginning. A master gateway discards duplicate messages that arrive; a slave sends an acknowledgement.

19.22 Handling Link State Request Packets

When forming an adjacency, a pair of gateways exchange summaries of links for which they have information. When an arriving summary specifies a link for which the receiver has no information or a link for which the sender has more recent information,



the receiving gateway generates a link state request message that asks the other gateway to send a link state advertisement. When a link state request message arrives, ospf calls procedure lsr_in to handle it.

```
/* lsr_in.c - lsr_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * lsr_in - handle a received link state request packet
 */
int lsr_in(pep)
struct ep *pep;
{
    struct ip*pipout, *pip = (struct ip *)pep->ep_data;
    struct ospf *poout, *po;
    struct ospf_if *pif = &ospf_if[pep->ep_ifn];
    struct ospf_nb *pnb;
    struct ospf_lsr *plsr;
    struct ospf_lsu *plsu;
    struct ospf_db *pdb, *db_lookup();
    struct ep*pepout, *ospflstmp();
    unsigned i, nlsr, maxlapp;

    if (pif->if_state <= IFS_WAITING)
        return;
    pnb = &pif->if_nbtab[1];
    po = (struct ospf *)((char *)pip+IP_HLEN(pip));
    for (i=0; i<MAXNBR; ++i, ++pnb)
        if (pnb->nb_rid == po->ospf_rid)
            break;
    if (i == MAXNBR || pnb->nb_state < NBS_EXCHNG)
        return;
    maxlapp = (nif[pep->ep_ifn].ni_mtu - IPMHLEN - MINLSULEN);
    maxlapp /= (LSSHDRLEN + MAXLSDLEN);
    pepout = ospflstmp(pif);
    if (pepout == 0)
        return;
```



```
    pipout = (struct ip *)pepout->ep_data;
    poout = (struct ospf *)pipout->ip_data;
    plsu = (struct ospf_lsu *)poout->ospf_data;

    nlsr = (po->ospf_len - MINHDRLEN) / sizeof (struct ospf_lsr);
    plsr = (struct ospf_lsr *)po->ospf_data;
    for (i=0; i<nlsr; ++i, ++plsr) {
        pdb = db_lookup(pif->if_area, plsr->lsr_type,
                        plsr->lsr_lsid);
        if (pdb == 0) {
            freebuf(pepout);
            nb_mismatch(pif, pnb);
            return;
        }
        if (plsu->lsu_nads >= maxlapp) {
            lsa_send(pif, pip->ip_src, pepout);
            if (!(pepout = ospflstimpl(pif)))
                return;
            pipout = (struct ip *)pepout->ep_data;
            poout = (struct ospf *)pipout->ip_data;
            plsu = (struct ospf_lsu *)poout->ospf_data;
        }
        lsa_add(pif, pepout, pdb);
    }
    lsa_send(pif, pip->ip_src, pepout);
}
```

When it begins, lsr_in checks to see that OSPF expects to receive a link state request on the interface over which the message arrived, and then searches the list of neighbors to find the neighbor that sent the message. Lsr_in examines the neighbor's state to ensure that it only accepts link state request messages from neighbors with which an adjacency is being established. After finding the sending neighbor in a valid state, lsr_in iterates through items in the link state request message and adds corresponding link state advertisements from the topological database to the link state message sent to the neighbor in response.

19.23 Building A Link State Summary

Procedure lss_build constructs a link state summary and inserts it on the neighbor's database summary list.



```
/* lss_build.c - lss_build */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

#define LSS_ABORT(par, pnb, pep) { \
    signal(par->ar_dbmutex); \
    if (pep) \
        freebuf(pep); \
    pnb->nb_state = NBS_2WAY; \
    nb_clearl(pnb); }

/*
 * lss_build - build link-state summaries for DD packets
 */
int lss_build(pif, pnb, pep)
struct ospf_if *pif;
struct ospf_nb *pnb;
struct ep *pep;
{
    struct ospf_ar     *par = pif->if_area;
    struct ip *pip = (struct ip *)pep->ep_data;
    struct ospf   *po = (struct ospf *)pip->ip_data;
    struct ospf_dd   *pdd = (struct ospf_dd *)po->ospf_data;
    struct ospf_db   *pdb;
    struct ep       *ospfddtmp();
    int      nls, ifn, maxl spp;

    ifn = pif - &ospf_if[0];
    maxl spp = (nif[ifn].ni_mtu - IPMHLEN - MINDDLEN) / LSSHDRLEN;
    wait(par->ar_dbmutex);
    nls = 0;
    for (pdb = par->ar_dblhead; pdb; pdb = pdb->db_lnext) {
        if (nls >= maxl spp) {
            pdd->dd_control = DDC_MORE;
            if (pnb->nb_master)
                pdd->dd_control |= DDC_MSTR;
            if (enq(pnb->nb_dsl, pep, 0) < 0) {

```



```
LSS_ABORT(par, pnb, pep);
return;
}
pep = ospfddtmp(pif);
if (pep == 0) {
    LSS_ABORT(par, pnb, pep);
    return;
}
pip = (struct ip *)pep->ep_data;
po = (struct ospf *)pip->ip_data;
pdd = (struct ospf_dd *)po->ospf_data;
nls = 0;
}
blkcopy(pdd->dd_lss[nls], &pdb->db_lsa.lsa_lss, LSSHDRLEN);
pdd->dd_lss[nls].lss_len = LSSHDRLEN;
po->ospf_len += LSSHDRLEN;
++nls;
}
pdd->dd_control = 0;
if (pnb->nb_master)
    pdd->dd_control |= DDC_MSTR;
if (enq(pnb->nb_dsl, pep, 0) < 0) {
    LSS_ABORT(par, pnb, pep);
    return;
}
signal(par->ar_dbmutex);
}
```

19.24 OSPF Utility Procedures

After receiving a packet, OSPF converts integer fields in the header to network byte order. Procedure `ospfnet2h` handles the conversion, which is straightforward.

```
/* ospfnet2h.c - ospfnet2h */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*
 * ospfnet2h - convert OSPF header from network to host byte order

```



```
*-----  
*/  
  
struct ospf *  
ospfnet2h(po)  
struct    ospf *po;  
{  
    /* NOTE: only includes fields in the main header */  
  
    po->ospf_len = net2hs(po->ospf_len);  
    po->ospf_cksum = net2hs(po->ospf_cksum);  
    po->ospf_authtype = net2hs(po->ospf_authtype);  
}
```

After converting integer fields in the header of an incoming packet to local byte order, OSPF examines values to determine whether the packet is valid (e.g., whether the checksum is correct). Procedure ospfcheck handles the details.

```
/* ospfcheck.c - ospfcheck */  
  
#include <conf.h>  
#include <kernel.h>  
#include <network.h>  
#include <ospf.h>  
  
/*-----  
 * ospfcheck - check if a packet is a valid OSPF packet  
 *-----  
*/  
  
int ospfcheck(pep)  
struct    ep    *pep;  
{  
    struct    ip    *pip = (struct ip *)pep->ep_data;  
    struct    ospf *po = (struct ospf *)((char *)pip + IP_HLEN(pip));  
    struct    ospf_if  *pif = &ospf_if[pep->ep_ifn];  
  
    if (pif->if_state == IFS_DOWN)  
        return FALSE;  
    if (po->ospf_version != OSPF_VERSION)  
        return FALSE;  
    if (net2hs(po->ospf_authtype) != pif->if_area->ar_authtype)  
        return FALSE;  
    if (pif->if_area->ar_authtype &&
```



```
memcmp(po->ospf_auth, pif->if_area->ar_auth, AUTHLEN))  
    return FALSE;  
  
memset(po->ospf_auth, 0, AUTHLEN);  
if (cksum(po, po->ospf_len>>1))  
    return FALSE;  
return TRUE;  
}
```

Ospfcheck begins by examining the state of the interface. If a system manager has assigned the interface state IFS_DOWN, OSPF cannot use the interface and cannot accept the packet. Ospfcheck rejects packets if the OSPF version number does not agree with the version number of the software, or if the type of authentication used for the arriving packet differs from the type of authentication configured. Furthermore, OSPF will reject the packet if the password sent in the authentication field does not match the password used for the area.

After checking authentication, ospfcheck verifies the checksum. Because the checksum does not include the authentication, ospfcheck replaces the authentication field with zeroes before computing the checksum. As with other TCP/IP protocols, ospfcheck includes the checksum field in the computation, and rejects the packet if the result is nonzero, Ospfcheck produces a Boolean result. If any test of the packet fails, ospfcheck returns FALSE immediately; it returns TRUE for packets that pass all tests.

Another utility procedure handles the situation when two neighboring gateways exchanging information disagree about their status or a sequence number. The master sends a sequence number that the slave uses to acknowledge receipt of a packet. Only one packet can be outstanding at any time, so no packet reordering occurs. If a packet arrives with an incorrect sequence number, OSPF drops the adjacency, moves the neighbor back to state EXSTART, and begins the exchange again.^① Procedure nb_mismatch handles the details.

```
/* nb_mismatch.c - nb_mismatch */  
  
#include <conf.h>  
#include <kernel.h>  
#include <network.h>  
#include <ospf.h>  
  
/*-----  
 * nb_mismatch - handle sequence # mismatch event  
 *-----*/
```

^① The only exception occurs when a complete duplicate copy arrives with the sequence number too low.



```
*/  
  
int nb_mismatch(pif, pnb)  
struct    ospf_if   *pif;  
struct    ospf_nb   *pnb;  
{  
    struct ep *pep;  
  
    pnb->nb_state = NBS_EXSTART;  
  
    /* empty lists */  
    while (pep = (struct ep *)deq(pnb->nb_lsdl))  
        freebuf(pep);  
    while (pep = (struct ep *)deq(pnb->nb_dsl))  
        freebuf(pep);  
    while (pep = (struct ep *)deq(pnb->nb_lsrl))  
        freebuf(pep);  
    pnb->nb_seq++;  
    dd_queue(pif, pnb);  
    return 0;  
}
```

The code is straightforward. Nb_mismatch begins by reducing the neighbor's state to NBS_EXSTART. It then clears three lists of pending requests, Nb_dsl contains database summaries, nb_lsdl contains link state advertisements, and nb_lsrl contains link state requests. After clearing all lists, nb_mismatch increments the sequence number and calls dd_queue to start sending a new database description from the beginning.

19.25 Summary

OSPF uses a link-state algorithm for route propagation. Each participating gateway maintains a directed graph model of the underlying internet in which each node corresponds to a gateway or a multiaccess network. Each gateway periodically broadcasts to all other gateways state information about its links (i.e., edges in the topology graph).

To ensure that all participating gateways receive a copy of each link state message. OSPF maintains connections among pairs of gateways known as adjacencies. OSPF uses Hello messages to discover neighbors, maintain adjacencies, and exchange topology information.

At any time, a neighbor must be in one of eight states. Many of the states control the formation of an adjacency and the exchange of database description packets. The



standard uses a finite state machine to describe the interaction between neighboring gateways and to specify their behavior. Like the TCP finite state machines, the neighbor state machine has complex semantics that cannot be isolated in individual procedures; instead, they permeate the code.

19.26 FOR FURTHER STUDY

Moy [RFC 1247] contains the specification for OSPF Version 2; additional information can be found in [RFC 1245 and RFC 1246].

19.27 EXERCISES

1. Does RIP or OSPF propagate routing changes more rapidly? Why?
2. Should every pair of gateways connected by point-to-point serial connections establish an adjacency? Why or why not?
3. Read the protocol standard to learn about OSPF's concept of a virtual link. When should virtual links be used? What advantages do they have?
4. Consider the election of a designated router. Characterize the circumstances under which the designated router changes. Will the designated router remain stable if one of the routers becomes disconnected and reconnected periodically? Explain.



20 SNMP: MIB Variables, Representations, And Bindings

20.1 Introduction

The Simple Network Management Protocol (SNMP) helps network managers locate and correct problems in a TCP/IP internet. Managers invoke an SNMP client on their local computer (usually a workstation), and use the client to contact one or more SNMP servers that execute on remote machines (usually gateways). SNMP uses a fetch-store paradigm in which each server maintains a set of conceptual variables that include simple statistics, such as a count of packets received, as well as complex variables that correspond to TCP/IP data structures, such as the ARP cache and IP routing tables. SNMP messages either specify that the server should fetch values from variables or store values in variables, and the server translates the requests to equivalent operations on local data structures. Because the protocol does not include other operations, all control must be accomplished through the fetch-store paradigm. In addition to the SNMP protocol, a separate standard for a Management Information Base (MIB) defines the set of variables that SNMP servers maintain as well as the semantics of each variable. MIB variables record the status of each connected network, traffic statistics, counts of errors encountered, and the current contents of internal data structures such as the machine's IP routing table.

SNMP defines both the syntax and meaning of the messages that clients and servers exchange. It uses Abstract Syntax Notation One (ASN.1) to specify both the format of messages and MIB variable names^①. Thus, unlike most other TCP/IP protocols, SNMP messages do not have fixed fields and cannot be defined with fixed structures.

This chapter describes the overall organization of SNMP software. It discusses the set of MIB variables and their ASN.1 names, and shows how a server binds ASN.1 names to corresponding internal variables. It considers the operations that SNMP allows, and shows why a server must maintain the lexical ordering among the ASN.1 names it

^① ASN.1 names are known as object identifiers.



supports. Later chapters consider how a client generates requests and how a server handles them.

20.2 Server Organization And Name Mapping

An SNMP server must accept an incoming request, perform the specified operation, and return a response. Understanding the basics of how servers process messages is important because it helps explain the mapping software shown throughout the remainder of this chapter. Figure 20.1 illustrates the flow of a message through an SNMP server.

Figure 20.1 The flow of an SNMP message through a server. The server repeats the third and fourth steps for each variable the message specifies.

As Figure 20.1 shows, the server first parses the message and translates to internal form. It then maps the MIB variable specification to the local data item that stores the needed information and performs the fetch or store operation. For fetch operations, it replaces the data area in the SNMP message with the value that it fetches. If the message specifies multiple variables, the server iterates through the third and fourth steps for each one. Finally, once all operations have been performed, the server translates the reply from internal form to external form, and returns it to the server.

The next sections describe MIB variables and concentrate on the details of name mapping. Later chapters show the remaining server software.

20.3 MIB Variables

The MIB defines variables that an SNMP server must maintain. To be more precise, the MIB defines a set of conceptual variables that an SNMP server must be able to access. In many cases, it is possible to use conventional variables to store the items the MIB requires. However, in other cases, the internal data structures used by TCP/IP protocols may not exactly match the variables required by the MIB. In such cases,



SNMP must be able to compute the necessary MIB values from available data structures. As an example of using computation in place of a variable, consider how a gateway might store the time a system has been operational. Many systems simply record the time at which the system started, and compute the time the system has been operating by subtracting the startup time from the current time. Thus, SNMP software can simulate a MIB "variable" that contains the time since last startup. It performs the computation whenever a request arrives to read a value from the MIB variable. To summarize:

The MIB defines conceptual variables that do not always correspond directly to the data structures a gateway uses. SNMP software may perform computation to simulate some of the conceptual variables, but the remote site will remain unaware of the computation.

Broadly speaking, variables in the MIB can be partitioned into two classes: simple variables and tables. Simple variables include types such as signed or unsigned integers and character strings. They also include data aggregates that correspond to structures in programming languages like C or records in languages like Pascal. In general, a gateway maintains one instance of each simple variable (e.g., a single integer that counts the total number of datagrams that IP receives). Tables correspond to one-dimensional arrays; a single table can contain multiple instances of a variable. For example, the MIB defines a table that corresponds to the set of network interfaces connected to a machine; the table has one entry for each network interface. The MIB defines other conceptual tables that correspond to The IP routing table on the server's machine, the ARP cache, and the set of TCP connections,

While the size of simple variables is known a priori, the size of a table can change as time proceeds. For example, the size of the table that corresponds to the ARP cache varies from one moment to the next as old entries time out or as new entries are added. At any time, the MIB address translation table has one entry for each binding in the ARP cache. If the time-to-live expires on an ARP binding, the cache management software removes it. The corresponding MIB table will contain one less entry.

20.3.1 Fields Within tables

Each entry in a MIB table can have multiple fields, which may themselves be simple variables or tables. Thus, it is possible to define an elementary data aggregate such as an array of integers or a more complicated one such as an array of pairs of address bindings.

20.4 MIB Variable Names

The MIB uses ASN.1 to name all variables. ASN.1 defines a hierarchical



namespace, so the name of each variable reflects its position in the hierarchy. The point of the ASN.1 hierarchy is to carefully distribute authority to assign names to many organizations. The scheme guarantees that although many organizations assign names concurrently, the resulting names are guaranteed to be unique and absolute. For example, the hierarchy leading to MIB names starts with the International Organization for Standardization (ISO). It follows through the organization subhierarchy, the United States Department of Defense subhierarchy, the Internet subhierarchy, the management subhierarchy, and the MIB subhierarchy. Each part of the hierarchy has been assigned a label, and a name is written as a sequence of labels that denote subhierarchies, with periods separating the labels. The label for the most significant hierarchy appears on the left. Thus, the MIB variable in the ip subhierarchy that counts incoming IP datagrams, ipInReceives, is named

```
i so. org. dod. internet. mgmt. mib. ip. ipInReceives
```

As the example shows, MIB names can be quite long. Of course, names for items in tables will be even longer than names for simple variables because they contain additional labels that encode the index of the table entry and the field desired in that entry.

20.4.1 Numeric Representation Of Names

When sending and receiving messages, SNMP does not store variable names as text strings. Instead, it uses a numeric form of ASN.1 to represent each name. Because the numeric representation is more compact than a textual representation, it saves space in packets.

The numeric form of ASN.1 assigns a unique (usually small) integer to each label in a name, and represents the name as a sequence of integers. For example, the sequence of numeric labels for the name of variable ipInReceives is

```
1. 3. 6. 1. 2. 1. 4. 3
```

When they appear in an SNMP message, the numeric representation of simple variable names has a zero appended to specify that the name represents the only instance of that variable in the MIB, so the exact form becomes

```
1. 3. 6. 1. 2. 1. 4. 3. 0
```

20.5 Lexicographic Ordering Among Names

ASN.1 defines a lexicographic ordering among names that provides a fundamental part of the SNMP functionality. The lexical ordering allows clients to ask a server for the set of currently available variables and to search a table without knowing its size.

The lexical ordering among names is similar to the ordering of words in a



dictionary, and is defined using their numeric representations. If the numeric representations are identical, we say that the names are lexically equal. Otherwise, the ordering is determined by comparing the labels. If one of the two names is an exact prefix of the other, we say that the shorter one is lexically less than the longer one. If neither is a prefix of the other and they are not identical, there must be at least one label that differs. Let $n_1, n_2, n_3\dots$ denote the numeric labels in name n and let $m_1, m_2, m_3\dots$ denote the numeric labels of m . Find the first label, i , for which n_i differs from m_i . If n_i is less than m_i , we say that n is lexically less than m . Otherwise, we say that m is lexically greater than n . To summarize;

Lexicographic ordering among names of MIB variables is defined using the numeric representation of ASN.1 object identifiers. Two names are lexically equal if they have identical representations. A name is lexically less than another if it is a prefix, or if it has a numerically lower value in the first label that differs.

20.6 Prefix Removal

Because SNMP software only needs to handle MIB variables, and because all MIB variable names begin with the same prefix, the software can eliminate needless computation and save space by representing names internally with the common prefix removed. In particular, each name in a packet must begin with the sequence for MIB variables:

i so. org. dod. internet. mgmt. mi b

or, numerically;

1. 3. 6. 1. 2. 1

Once an SNMP server examines the prefix to insure that the name does indeed refer to a MIB variable, it can ignore the prefix, and use only the remainder of the name internally. As we will see, doing so saves time and keeps internal representations smaller. Similarly, a client can save space by adding the common prefix when it is ready to send a message. We can summarize:

SNMP software improves performance by storing and manipulating suffixes of MIB names. It adds the common prefix before sending a name in a message to another machine, and removes the common prefix (after verifying it) when a message arrives.



20.7 Operations Applied To MIB Variables

Before examining the data structures SNMP software uses to store information, it is important to consider how the server will use those data structures. A client can issue three basic commands to a server. Two of the commands are obvious and require a straightforward mapping. The client sends a set-request to assign a value to a variable. It sends a get-request to fetch the value currently stored in a variable. Before it can perform the request, the server must map the numerically encoded ASN.1 names found in the incoming request into the appropriate internal variables that store values for those names.

Clients can also issue a get-next-request command. Unlike set-request or get-request commands, a get-next-request does not specify the name of an item to retrieve. Instead, it specifies a name, and asks the server to respond with the name and value of the variable that occurs next in the lexical sequence. The server finds the next variable with a name lexically greater than the specified name, and performs a get-request operation on that variable to obtain the value.

The get-next-request command is especially useful for accessing values in a table of unknown size. A client can continually issue get-next-request commands and have the server move through values in the table automatically. Each request specifies the name of the variable returned in the previous response, allowing the server to specify the name of the next item in its response. The process of stepping through entries one at a time is called walking the table.

20.8 Names For Tables

To facilitate use of the get-next-request command, some MIB names correspond to entire tables instead of individual items. These names do not have individual values, and clients cannot use them directly in get-request commands. However, the client can specify the name of a table in a get-next-request command to retrieve the first item in the table. For example, the MIB defines the name

iso.org.dod.internet.mgmt.mib.ip.ipAddrTable

to refer to a conceptual table of IP addresses by which the server's machine^① is known. The numeric equivalent is

1.3.6.1.2.1.4.20

A get-request command that uses a table name will fail because the name does not correspond to an individual item. However, the name is important because it allows clients to find the first item in a table without knowing the name of the previous MIB

^① The standard uses the term entity agent to refer to the machine on which a server executes.



variable. The client issues a get-next-request using the table name to extract the first item in the table. It uses the name returned in the response, and then issues get-next-request commands to step through items in the table one at a time.

One final rule complicates the implementation of data structures that support get-next-request commands: a get-next-request always skips to the next simple variable in the lexicographic ordering. More important, the current contents of variables available at a given server determine the set of names that the server skips. In particular, a get-next-request command always skips an empty table. So, if a server's ARP cache happens to be empty when the client sends a get-next-request for it, the server will skip to the lexicographically next, possibly unrelated, variable. To summarize:

Servers only return the values of simple variables in response to a get-next-request command; they must skip empty tables when the request arrives.

As a consequence, the server cannot simply use a lexically ordered list of MIB variables to determine which variable satisfies a get-next-request command. Instead, it must contain code that examines items in the lexical ordering, skips any that are empty, and finds the first simple variable in the next nonempty item.

20.9 Conceptual Threading Of The Name Hierarchy

If MIB variables are arranged in a hierarchy according to their ASN.1 names, they define a tree. The definition of lexically next can be thought of as a set of threads in the tree as Figure 20.2 illustrates.



Figure 20.2 Part of the conceptual naming tree for MIB variables. Dashed lines show the set of threads in the tree that define the lexically next item. The lexical ordering skips nodes that correspond to tables.

20.10 Data Structure For MIB Variables

SNMP must keep information about each conceptual MIB variable. The implementation uses an array to store the information, where each item in the array corresponds to a single MIB variable. Structure `mib_info`, found in file `mib.h` defines the contents of each item.

```
/* mib.h */

/* mib information structure: 1 per mib entry; Initialized in snmib.c */
struct mib_info {
    char *mi_name; /* name of mib variable in English */
    char *mi_prefix; /* prefix in English (e.g., "tcp.") */
    struct oid mi_objid; /* object identifier */
    int mi_vartype; /* type: integer, aggregate, octet str */
    Bool mi_writable; /* is this variable writable? */
    Bool mi_varleaf; /* is this a leaf with a single value? */
    int (*mi_func)(); /* function to implement get/set/getnext*/
```



```
int mi_param; /* parameter used with function      */
struct mib_info *mi_next; /* pointer to next var. in lexi-order*/
};

extern struct mib_info mib[]; /* array with one entry per MIB variable*/
extern int mib_entries; /* number of entries in mib array */

/* Information about MIB tables. Contains functions to implement      */
/* operations upon variables in the tables.                         */
struct tab_info {
    int (*ti_get)(); /* get operation           */
    int (*ti_getf)(); /* get first operation     */
    int (*ti_getn)(); /* getnext operation       */
    int (*ti_set)(); /* set operation           */
    int (*ti_match)(); /* match operation: is a given oid   */
                       /* found in the current table? */
    struct mib_info *ti_mip; /* pointer to mib information record */
};

extern struct tab_info tabtab[]; /* table of MIB table information */

#define LEAF 1 /* This MIB variable is a leaf.      */
#define NLEAF 0 /* This MIB variable is not a leaf. */

#define T_TCPTABLE 0x0 /* var is the TCP conn. table */
#define T_RTTABLE 0x1 /* var is the routing table */
#define T_AETABLE 0x2 /* var is the address entry tbl */
#define T_ATTABLE 0x3 /* var is the addr translat tbl */
#define T_IFTABLE 0x4 /* var is the interface table */
/*#define T_EGPTABLE 0x5 */ /* var is the egp table */
#define T_AGGREGATE 0x6 /* var is an aggregate */

/* this type specifies in mib.c that the object is a table. The value is
   different than any of the ASN.1 types that SNMP uses. */
#define T_TABLE 01
```

Each item in the array contains sufficient information to identify a variable, including its ASN.1 name (object identifier), its type, and whether it is writable.



20.10.1 Using Separate Functions To Perform Operations

Most of the SNMP server data structures focus on providing efficient name mapping. When an ASN.1 name arrives in a request, the server must be able to recognize it, and call a procedure that will honor the request. Instead of trying to encode all information about how to satisfy a request, the implementation invokes a function. It passes the function three arguments: the request, a parameter (usually a memory address), and a pointer to the mib entry for the name. Field `mi_func` in structure `mib_info` contains the address of the function to call for a given variable.

Many of the functions needed to access variables are quite straightforward: they merely translate from the internal data representation to the ASN.1 format used in SNMP messages. For such cases, it is possible to encode the type of conversion and the address of the variable in the mib entry, and arrange to have a single procedure handle the conversion. However, if the server does not have an explicit representation available for a given MIB variable, requests to fetch or store a value from that variable may require more computation than merely converting representations. Certainly, requests to access items in conceptual MIB tables require computation to translate from the MIB name to the local data structures used to store the information. Finally, SNMP must provide a way to compute which item satisfies a get-next-request command. To keep the implementation uniform while providing sufficient generality to accommodate all requests, the implementation always uses a function to handle requests. We can summarize:

The amount of computation required to apply a given request to a given MIB variable depends on the variable and the request. Using a separate function to handle access requests keeps the implementation uniform, and allows the server to use a single mechanism to handle all requests.

20.11 A Data Structure For Fast Lookup

As we will see later, array `mib` contains the bindings between ASN.1 names (object identifiers) and internal variables. If sequential search were used, finding an entry could be time consuming. To make name binding efficient, the software uses an auxiliary hash table as Figure 20.3 illustrates.



Figure 20.3 The mib array and hash table snhtab that speeds MIB variable lookup. Arrows on the right hand side show lexical links used for get-next-requests; the lexical order skips nonleaf items.

The technique used is known as bucket hashing. The hash table itself consists of an array of pointers. Each pointer gives the address of a linked list of nodes that represent the MIB variables that hash to that address. To find the information for a MIB variable, the software computes the hash function using the numeric representation of the variable name, selects a linked list from the hash table, and searches the list. Each entry on the list contains a pointer to the MIB variable it represents as well as a pointer to the next item on the list.

In practice, a simple hashing scheme works well. Of the 89^① variables defined in the MIB, hashing into an array of 101 positions produces lists with an average length slightly greater than 1. Thus, most lookups can be found immediately and do not require the software to search a linked list.

^① Our example code specifies only 82 items because it does not include EGP variables.



20.12 Implementation Of The Hash Table

File snhash.h contains declarations of the hash table and nodes on the linked lists.

```
/* snhash.h */

struct snhnode {           /* hash table node structure      */
    struct mib_info *sh_mip;   /* points to a mib record      */
    struct snhnode *sh_next;   /* next node in this hash bucket*/
};

#define S_HTABSIZE 101        /* hash table size - a prime #  */
#define S_HTRADIX 21          /* hash table radix            */
#define S_NUMTABS 5            /* number of table objects   */

extern struct snhnode *snhtab[]; /* the hash table      */
```

Structure snhnode defines the contents of a node on one of the linked lists. Each node only contains a pointer to an entry in the mib array and a pointer to the next node on the list.

20.13 Specification Of MIB Bindings

File snmib.c contains code that initializes the mib array to contain one entry per MIB variable. The entry contains all information about the MIB object, including the numerical representation of its ASN.1 name, the address of an internal variable that contains the value associated with the MIB variable, and the address of a function that can be called to perform SNMP operations on the variable. Note that the pointer to the next item in the lexicographically ordered list of variables is set to zero in each of these static declarations. Although the lexical pointers are not initialized in the declaration, all MIB objects must be specified in reverse lexicographic order. SNMP software will initialize the lexical pointers at system startup.

```
/* snmib.c */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>
```



```
#include <snmpvars.h>

extern int sntable(), snleaf();

extern struct oid SysObjectID;

/* All table and leaf variables that are found in the MIB */
struct mib_info mib[] = {
    { "system", "", { {1}, 1 }, T_AGGREGATE,
        FALSE, NLEAF, 0, 0, 0 },
    { "sysDescr", "system.", { {1, 1, 0}, 3 }, ASN1_OCTSTR,
        FALSE, LEAF, snleaf, (int) &SysDescr, 0 },
    { "sysObjectID", "system.", { {1, 2, 0}, 3 }, ASN1_OBJID,
        FALSE, LEAF, snleaf, (int) &SysObjectID, 0 },
    { "sysUpTime", "system.", { {1, 3, 0}, 3 }, ASN1_TIMETICKS,
        FALSE, LEAF, snleaf, (int) &SysUpTime, 0 },
    { "if", "", { {2}, 1 }, T_AGGREGATE,
        FALSE, NLEAF, 0, 0, 0 },
    { "ifNumber", "if.", { {2, 1, 0}, 3 }, ASN1_INT,
        FALSE, LEAF, snleaf, (int) &IfNumber, 0 },
    { "ifTable", "if.", { {2, 2}, 2 }, T_AGGREGATE,
        TRUE, NLEAF, 0, 0, 0 },
    { "ifEntry", "if.ifTable.", { {2, 2, 1}, 3 }, T_TABLE,
        TRUE, NLEAF, sntable, (int) &tabtab[T_IFTABLE], 0 },
    { "at", "", { {3}, 1 }, T_AGGREGATE,
        TRUE, NLEAF, 0, 0, 0 },
    { "atTable", "at.", { {3, 1}, 2 }, T_AGGREGATE,
        TRUE, NLEAF, 0, 0, 0 },
    { "atEntry", "at.atTable.", { {3, 1, 1}, 3 }, T_TABLE,
        TRUE, NLEAF, sntable, (int) &tabtab[T_ATTABLE], 0 },
    { "ip", "", { {4}, 1 }, T_AGGREGATE,
        TRUE, NLEAF, 0, 0, 0 },
    { "ipForwarding", "ip.", { {4, 1, 0}, 3 }, ASN1_INT,
        FALSE, LEAF, snleaf, (int) &IpForwarding, 0 },
    { "ipDefaultTTL", "ip.", { {4, 2, 0}, 3 }, ASN1_INT,
        TRUE, LEAF, snleaf, (int) &IpDefaultTTL, 0 },
    { "ipInReceives", "ip.", { {4, 3, 0}, 3 }, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (int) &IpInReceives, 0 },
    { "ipInHdrErrors", "ip.", { {4, 4, 0}, 3 }, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (int) &IpInHdrErrors, 0 },
    { "ipInAddrErrors", "ip.", { {4, 5, 0}, 3 }, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (int) &IpInAddrErrors, 0 },
```



```
{ "ipForwDatagrams", "ip.", { {4, 6, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpForwDatagrams, 0},
{ "ipInUnknownProtos", "ip.", { {4, 7, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpInUnknownProtos, 0},
{ "ipInDiscards", "ip.", { {4, 8, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpInDiscards, 0},
{ "ipInDelivers", "ip.", { {4, 9, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpInDelivers, 0},
{ "ipOutRequests", "ip.", { {4, 10, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpOutRequests, 0},
{ "ipOutDiscards", "ip.", { {4, 11, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpOutDiscards, 0},
{ "ipOutNoRoutes", "ip.", { {4, 12, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpOutNoRoutes, 0},
{ "ipReasmTimeout", "ip.", { {4, 13, 0}, 3}, ASN1_INT,
    FALSE, LEAF, snleaf, (int) &IpReasmTimeout, 0},
{ "ipReasmReqds", "ip.", { {4, 14, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpReasmReqds, 0},
{ "ipReasmOKs", "ip.", { {4, 15, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpReasmOKs, 0},
{ "ipReasmFails", "ip.", { {4, 16, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpReasmFails, 0},
{ "ipFragOKs", "ip.", { {4, 17, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpFragOKs, 0},
{ "ipFragFails", "ip.", { {4, 18, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpFragFails, 0},
{ "ipFragCreates", "ip.", { {4, 19, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IpFragCreates, 0},
{ "ipAddrTable", "ip.", { {4, 20}, 2}, T_AGGREGATE,
    FALSE, NLEAF, 0, 0, 0},
{ "ipAddrEntry", "ip.ipAddrTable.", { {4, 20, 1}, 3}, T_TABLE,
    FALSE, NLEAF, sntable, (int) &tabtab[T_AETABLE], 0},
{ "ipRoutingTable", "ip.", { {4, 21}, 2}, T_AGGREGATE,
    TRUE, NLEAF, 0, 0, 0},
{ "ipRouteEntry", "ip.ipRoutingTable.", { {4, 21, 1}, 3}, T_TABLE,
    TRUE, NLEAF, sntable, (int) &tabtab[T_RTTPABLE], 0},
{ "icmp", "", { {5}, 1}, T_AGGREGATE,
    FALSE, NLEAF, 0, 0, 0},
{ "icmpInMsgs", "icmp.", { {5, 1, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInMsgs, 0},
{ "icmpInErrors", "icmp.", { {5, 2, 0}, 3}, ASN1_COUNTER,
```



```
    FALSE, LEAF, snleaf, (int) &IcmpInErrors, 0},
{ "icmpInDestUnreachs", "icmp.", { {5, 3, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInDestUnreachs, 0},
{ "icmpInTimeExclds", "icmp.", { {5, 4, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInTimeExclds, 0},
{ "icmpInParmProbs", "icmp.", { {5, 5, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInParmProbs, 0},
{ "icmpInSrcQuenchs", "icmp.", { {5, 6, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInSrcQuenchs, 0},
{ "icmpInRedirects", "icmp.", { {5, 7, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInRedirects, 0},
{ "icmpInEchos", "icmp.", { {5, 8, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInEchos, 0},
{ "icmpInEchoReps", "icmp.", { {5, 9, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInEchoReps, 0},
{ "icmpInTimestamps", "icmp.", { {5, 10, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInTimestamps, 0},
{ "icmpInTimestampReps", "icmp.", { {5, 11, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInTimestampReps, 0},
{ "icmpInAddrMasks", "icmp.", { {5, 12, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInAddrMasks, 0},
{ "icmpInAddrMaskReps", "icmp.", { {5, 13, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpInAddrMaskReps, 0},
{ "icmpOutMsgs", "icmp.", { {5, 14, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutMsgs, 0},
{ "icmpOutErrors", "icmp.", { {5, 15, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutErrors, 0},
{ "icmpOutDestUnreachs", "icmp.", { {5, 16, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutDestUnreachs, 0},
{ "icmpOutTimeExclds", "icmp.", { {5, 17, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutTimeExclds, 0},
{ "icmpOutParmProbs", "icmp.", { {5, 18, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutParmProbs, 0},
{ "icmpOutSrcQuenchs", "icmp.", { {5, 19, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutSrcQuenchs, 0},
{ "icmpOutRedirects", "icmp.", { {5, 20, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutRedirects, 0},
{ "icmpOutEchos", "icmp.", { {5, 21, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutEchos, 0},
{ "icmpOutEchoReps", "icmp.", { {5, 22, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutEchoReps, 0},
```



```
{ "icmpOutTimestamps", "icmp.", { {5, 23, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutTimestamps, 0},
{ "icmpOutTimestampReps", "icmp.", { {5, 24, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutTimestampReps, 0},
{ "icmpOutAddrMasks", "icmp.", { {5, 25, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutAddrMasks, 0},
{ "icmpOutAddrMaskReps", "icmp.", { {5, 26, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &IcmpOutAddrMaskReps, 0},
{ "tcp", "", { {6}, 1}, T_AGGREGATE,
    FALSE, NLEAF, 0, 0, 0},
{ "tcpRtoAlgorithm", "tcp.", { {6, 1, 0}, 3}, ASN1_INT,
    FALSE, LEAF, snleaf, (int) &TcpRtoAlgorithm, 0},
{ "tcpRtoMin", "tcp.", { {6, 2, 0}, 3}, ASN1_INT,
    FALSE, LEAF, snleaf, (int) &TcpRtoMin, 0},
{ "tcpRtoMax", "tcp.", { {6, 3, 0}, 3}, ASN1_INT,
    FALSE, LEAF, snleaf, (int) &TcpRtoMax, 0},
{ "tcpMaxConn", "tcp.", { {6, 4, 0}, 3}, ASN1_INT,
    FALSE, LEAF, snleaf, (int) &TcpMaxConn, 0},
{ "tcpActiveOpens", "tcp.", { {6, 5, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &TcpActiveOpens, 0},
{ "tcpPassiveOpens", "tcp.", { {6, 6, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &TcpPassiveOpens, 0},
{ "tcpAttemptFails", "tcp.", { {6, 7, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &TcpAttemptFails, 0},
{ "tcpEstabResets", "tcp.", { {6, 8, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &TcpEstabResets, 0},
{ "tcpCurrEstab", "tcp.", { {6, 9, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &TcpCurrEstab, 0},
{ "tcpInSegs", "tcp.", { {6, 10, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &TcpInSegs, 0},
{ "tcpOutSegs", "tcp.", { {6, 11, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &TcpOutSegs, 0},
{ "tcpRetransSegs", "tcp.", { {6, 12, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &TcpRetransSegs, 0},
{ "tcpConnTable", "tcp.", { {6, 13}, 2}, T_AGGREGATE,
    FALSE, NLEAF, 0, 0, 0},
{ "tcpConnEntry", "tcp.tcpConnTable.", { {6, 13, 1}, 3}, T_TABLE,
    FALSE, NLEAF, sntable, (int) &tabtab[T_TCPTABLE], 0},
{ "udp", "", { {7}, 1}, T_AGGREGATE,
    FALSE, NLEAF, 0, 0, 0},
{ "udpInDatagrams", "udp.", { {7, 1, 0}, 3}, ASN1_COUNTER,
```



```
    FALSE, LEAF, snleaf, (int) &UdpInDatagrams, 0},
{ "udpNoPorts", "udp.", { {7, 2, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &UdpNoPorts, 0},
{ "udpInErrors", "udp.", { {7, 3, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &UdpInErrors, 0},
{ "udpOutDatagrams", "udp.", { {7, 4, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf, (int) &UdpOutDatagrams, 0}
};

int mib_entries = sizeof(mib) / sizeof(struct mib_info);

/* Funcs that implement get, getfirst, getnext, set, and match for tables */
extern int
    stc_get(),    stc_getf(),    stc_getn(),    stc_set(),
    stc_match(),   srt_get(),    srt_getf(),    srt_getn(),
    srt_set(),    srt_match(),   sae_get(),    sae_getf(),
    sae_getn(),   sae_set(),    sae_match(),   sat_get(),
    sat_getf(),   sat_getn(),   sat_set(),    sat_match(),
    sif_get(),    sif_getf(),   sif_getn(),   sif_set(),
    sif_match();

struct tab_info tabtab[] = {
    { stc_get, stc_getf, stc_getn, stc_set, stc_match, 0, },
    { srt_get, srt_getf, srt_getn, srt_set, srt_match, 0, },
    { sae_get, sae_getf, sae_getn, sae_set, sae_match, 0, },
    { sat_get, sat_getf, sat_getn, sat_set, sat_match, 0, },
    { sif_get, sif_getf, sif_getn, sif_set, sif_match, 0 }
};
```

20.14 Internal Variables Used In Bindings

File snmpvars.h declares the types of internal variables used in MIB bindings.

```
/* snmpvars.h */

/* System & Interface MIB */

extern char *SysDescr[], SysContact[], SysName[], SysLocation[];
extern unsigned
    SysUpTime, SysServices, IfNumber;

/* IP MIB */
```



```
extern unsigned
    IpForwarding, IpDefaultTTL, IpInReceives, IpInHdrErrors,
    IpInAddrErrors, IpForwDatagrams, IpInUnknownProtos, IpInDiscards,
    IpInDelivers, IpOutRequests, IpOutDiscards, IpOutNoRoutes,
    IpReasmTimeout, IpReasmReqds, IpReasmOKs, IpReasmFails, IpFragOKs,
    IpFragFails, IpFragCreates, IpRouting Discards;

/* ICMP MIB */
extern unsigned
    IcmpInMsgs, IcmpInErrors, IcmpInDestUnreachs, IcmpInTimeExcds,
    IcmpInParmProbs, IcmpInSrcQuenches, IcmpInRedirects, IcmpInEchos,
    IcmpInEchoReps, IcmpInTimestamps, IcmpInTimestampReps,
    IcmpInAddrMasks, IcmpInAddrMaskReps, IcmpOutMsgs, IcmpOutErrors,
    IcmpOutDestUnreachs, IcmpOutTimeExcds, IcmpOutParmProbs,
    IcmpOutSrcQuenches, IcmpOutRedirects, IcmpOutEchos,
    IcmpOutEchoReps, IcmpOutTimestamps, IcmpOutTimestampReps,
    IcmpOutAddrMasks, IcmpOutAddrMaskReps;

/* UDP MIB */
extern unsigned
    UdpInDatagrams, UdpNoPorts, UdpInErrors, UdpOutDatagrams;

/* TCP MIB */
extern unsigned
    TcpRtoAlgorithm, TcpRtoMin, TcpRtoMax, TcpMaxConn, TcpActiveOpens,
    TcpPassiveOpens, TcpAttemptFails, TcpEstabResets, TcpCurrEstab,
    TcpInSegs, TcpOutSegs, TcpRetransSegs;
```

20.15 Hash Table Lookup

Function getmib uses the hash table to find the MIB entry that corresponds to a given name (object identifier). It calls function hashoid to compute the hash of the name. To form an integer value that represents the object id, hashoid treats the identifier as a sequence of digits in radix S_HTRADIX. It iterates through the object identifier, multiplying by the radix and adding a new "digit" at each step.

The source code for both getmib and hashoid can be found in file snhash.c, which also contains the hash table initialization procedure hashinit.

```
/* snhash.c - getmib, hashoid, hashinit */

#include <conf.h>
#include <kernel.h>
```



```
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <snhash.h>

struct    snhnode   *snhtab[S_HTABSIZ];

extern struct tab_info tabtab[];

/*-----
 * getmib - find mib record for the given object id
 *-----
 */
struct mib_info *getmib(oip)
struct oid      *oip;
{
    struct snhnode     *hp;
    int      loc, i;

    loc = hashoid(oip);      /* try the regular hash table */
    for (hp = snhtab[loc]; hp; hp = hp->sh_next)
        if (oidequ(oip, &hp->sh_mip->mi_objid))
            return hp->sh_mip;
    for (i = 0; i < S_NUMTABS; ++i) /* try the table table */
        if (blkequ(tabtab[i].ti_mip->mi_objid.id, oip->id,
                   tabtab[i].ti_mip->mi_objid.len * 2))
            return tabtab[i].ti_mip;
    return 0;
}

/*-----
 * hashoid - hash the object id
 *-----
 */
int hashoid(oip)
struct oid      *oip;
{
    register unsigned tot;
    register int   i;

    for (tot = 0, i = oip->len - 1; i >= 0; i--)

```



```
    tot = tot * S_HTRADIX + oip->id[i];
    return tot % S_HTABSIZ;
}

/*-----
 * hashinit - initialize the hash table
 *-----
 */
hashinit()
{
    int i;

    register struct snhnode **ht;
    register struct mib_info *mp;
    struct mib_info *lastnodep;
    struct snhnode *hp;
    int loc, tabtabct;

    /* clear the hash table */
    ht = snhtab;
    for (i = 0; i < S_HTABSIZ; i++)
        *ht++ = 0;
    lastnodep = 0;

    tabtabct = 0;
    for (i=0, mp = &mib[mib_entries - 1]; i<mib_entries; i++, mp--) {
        loc = hashoid(&mp->mi_objid);
        hp = (struct snhnode *) getmem(sizeof(struct snhnode));
        hp->sh_mip = mp;
        hp->sh_next = snhtab[loc];
        snhtab[loc] = hp;
        mp->mi_next = lastnodep;
        /* (node == table) ==> insert into array of tables */

        if (mp->mi_vartype == T_TABLE)
            tabtab[tabtabct++].ti_mip = mp;
        if (mp->mi_varleaf || mp->mi_vartype == T_TABLE)
            lastnodep = mp;
    }
}
```



20.16 SNMP Structures And Constants

File snmp.h contains the definitions of data structures and symbolic constants used throughout the code,

```
/* snmp.h - strequ, oidequ */

#define SNMPD          snmpd      /* SNMP server code           */
extern int       SNMPD();    /* SNMP server daemon         */
#define SNMPSTK        8000      /* SNMP server stack size    */
#define SNMPPRI        20        /* SNMP server priority      */
#define SNMPDNAM      "snmpd"   /* SNMP server daemon name   */

#define SMAXOBJID 32      /* max # of sub object ids  */
#define OJBSUBIDTYPE unsigned short /* type of sub object ids   */

/* strequ - return TRUE if strings x and y are the same           */
#define strequ(x,y)  (strcmp((x), (y)) == 0)

#define u_char        unsigned char

struct oid {           /* object identifier           */
    OJBSUBIDTYPE id[SMAXOBJID]; /* array of sub-identifiers */
    int      len;     /* length of this object id */
};

/*
 * oidequ - check if the lengths of the oid's are the same, then check
 *           the contents of the oid's
 */
#define oidequ(x,y) ((x)->len == (y)->len && \
                  blkequ((x)->id, (y)->id, (y)->len * 2))

/* Structure that contains the value of an SNMP variable.           */
struct snval {
    unsigned char sv_type; /* variable type           */
    union {
        /* value of var is one of these */
        int sv_int;      /* variable is one of: integer, */
                         /* counter, gauge, timeticks */
        struct {
            /* variable is a (octet) string */
            char *sv_str; /* string's contents */
            int sv_len;   /* string's length */
        }
    }
};
```



```
        } sv_str;

    struct    oid sv_oid;    /* variable is an object id */
    IPAddr    sv_ipaddr;    /* variable is an IP address      */
} sv_val;
};

/* Functions to access parts of the above snval structure      */

#define SVTYPE(bl) ((bl)->sb_val.sv_type)
#define SVINT(bl)   ((bl)->sb_val.sv_val.sv_int)
#define SVSTR(bl)   ((bl)->sb_val.sv_val.sv_str.sv_str)
#define SVSTRLEN(bl) ((bl)->sb_val.sv_val.sv_str.sv_len)
#define SVOID(bl)   ((bl)->sb_val.sv_val.sv_oid.id)
#define SVOIDLEN(bl) ((bl)->sb_val.sv_val.sv_oid.len)
#define SVIPADDR(bl) ((bl)->sb_val.sv_val.sv_ipaddr)

/*
 * Each snblist node contains an SNMP binding in one of 2 forms: ASN.1
 * encoded form or internal form. The bindings list is doubly-linked
 */
struct snbentry {

    struct    oid sb_oid;        /* object id in internal form      */
    struct    snval sb_val;      /* value of the object            */
    u_char    *sb_alstr;        /* ASN.1 string containing the    */
                               /* object id and its value      */
    int     sb_alslen;         /* length of the ASN.1 string    */
    struct    snbentry *sb_next; /* next node in the bind list    */
    struct    snbentry *sb_prev; /* previous node in the list     */
};

/* Structure that holds a complete description of an SNMP request      */
struct req_desc {

    u_char    reqtype;        /* request type                  */
    u_char    reqid[10];       /* request identifier           */
    int     reqidlen;         /* length of the identifier     */
    u_char    err_stat;        /* error status                 */
    u_char    err_idx;         /* error index                  */
    int     err_stat_pos;      /* position of error status in */
                               /* the ASN.1 encoding          */
    int     err_idx_pos;       /* position of error index     */
    int     pductype_pos;      /* position of pdu type        */
    struct    snbentry *bindlf; /* front of bindings list      */
};
```



```
    struct    snbentry *bindle; /* end of bindings list      */

};

#define SNMPMAXSZ  U_MAXLEN /* max SNMP request size      */
#define SNMPPORT   161       /* SNMP server UDP port      */

/* SNMP error types
#define SNMP_OK        0       /* no error                  */
#define SERR_TOO_BIG   1       /* reply would be too long  */
#define SERR_NO_SUCH   2       /* no such object id exists */
#define SERR_BAD_VALUE 3       /* bad value in a set operation */

#define SVERS_LEN     1       /* SNMP version is 1 byte long */
#define SVERSION      0       /* current SNMP version      */

#define SCOMM_STR    "public" /* SNMP community string      */

/* operations to be applied to an SNMP object
#define SOP_GET       1       /* get operation              */
#define SOP_GETN      2       /* getnext operation          */
#define SOP_GETF      3       /* get first operation         */
#define SOP_SET       4       /* set operation              */

/* standard version and community string -- backwards */
static char SNVCBACK[] = {
    'c', 'i', 'l', 'b', 'u', 'p', 0x06, 0x04 /* ASN1_OCTSTR */,
    SVERSION, SVERS_LEN, 0x02 /* ASN1_INT */
};

#define SNVCLEN      sizeof(SNVCBACK) /* length of SNVCBACK      */

/* SNMP client return codes
#define SCL_OK        0       /* good response -- no errors */
#define SCL_OPENF     1       /* open fails                */
#define SCL_WRITEF    2       /* write fails               */
#define SCL_NORESP   3       /* no response from server  */
#define SCL_READF    4       /* read fails                */
#define SCL_BADRESP  5       /* bad response              */
#define SCL_BADREQ  6       /* bad request               */

/* Table specific constants
#define SNUMF_AETAB   4       /* 4 fields in an Addr Entry
```



```
#define SNUMF_ATTAB      3      /* 3 fields in an Addr Transl. Entry */
#define SNUMF_IFTAB      21     /* 21 fields in an Interface Entry */
#define SNUMF_RTTAB      10      /* 10 fields in a Route Table Entry */
#define SNUMF_TCTAB      5       /* 5 fields in a TCP Connection Entry */

#define SAE_OIDLEN      3      /* all sae variables start with 4.20.1      */
#define SAT_OIDLEN 3      /* all sat variables start with 3.1.1*/
#define SIF_OIDLEN 3      /* all sif variables start with 2.2.1*/
#define SRT_OIDLEN 3      /* all srt variables start with 4.21.1      */
#define STC_OIDLEN 3      /* all stc variables start with 6.13.1      */
```

ASN.1 defines the exact representation for all objects sent in an SNMP message, including variable names (object identifiers), integers, sequences, IP addresses, and SNMP commands. Usually, ASN.1 represents each object with a type, length, and value. The type, which specifies what kind of object follows, distinguishes between integers, commands, and counters. The length specifies the number of octets in the representation, and the value consists of the octets that comprise the object.

20.17 ASN.1 Representation Manipulation

File asn1.h contains definitions of symbolic constants used to specify ASN.1 types. Most are self-explanatory. The sequence type is used to denote a repetition of items (e.g., sequence of integers), and can be thought of as corresponding to an array in a programming language. The NULL type is used when no value is needed.

```
/* asn1.h - A1_SIGNED */

/* constants for parsing an SNMP packet, according to ASN.1 */

/* ASN.1 object types */

#define ASN1_SEQ          0x30 /* sequence object      */
#define ASN1_INT          0x02 /* integer             */
#define ASN1_OCTSTR        0x04 /* octet string        */
#define ASN1_NULL          0x05 /* null                */
#define ASN1_OBJID         0x06 /* object identifier   */
#define ASN1_IPADDR        0x40 /* ip address          */
#define ASN1_COUNTER        0x41 /* counter             */
#define ASN1_GAUGE          0x42 /* gauge               */
#define ASN1_TIMETICKS      0x43 /* time ticks          */

/* Protocol Data Unit types -- SNMP specific */
```



```
#define PDU_GET          0xA0 /* get request           */
#define PDU_GETN         0xA1 /* get-next request      */
#define PDU_RESP         0xA2 /* response             */
#define PDU_SET          0xA3 /* set request           */
#define PDU_TRAP         0xA4 /* trap message          */

/* Constants used for conversion of objects to/from ASN.1 notation */
#define CHAR_BITS   8       /* number of bits per char */
#define CHAR_HIBIT 0x80     /* octet with the high bit set */
#define BYTE2_HIBIT 0x8000   /* 2 bytes with high bit set */
#define BYTE3_HIBIT 0x800000 /* 3 bytes with high bit set */
#define BYTE4_HIBIT 0x8000000 /* 4 bytes with high bit set */

#define A1_SIGNED(x) ((x) == ASN1_INT)

/* the standard MIB prefix - 1.3.6.1.2.1 */
extern char MIB_PREFIX[];

/* the standard MIB prefix is encoded by ASN.1 into 5 octets */
#define MIB_PREF_SZ    5
```

20.17.1 Representation Of Length

ASN.1 uses two representations for the length field in an object specification as Figure 20.4 illustrates. If the object requires fewer than 128 octets, ASN.1 uses a short form in which a single octet encodes the object length. Because the 8-bit binary representations of values less than 128 have the high-order bit set to zero, programs use the high-order bit to check for the short form.

Figure 20.4 The ASN.1 encoding of object lengths. The short form in (a) is used to represent lengths less than 128, while the long-form in (b) is used to represent longer lengths. The high-order bit of the first byte distinguishes the two forms.

In the long form, ASN.1 uses a multiple-octet integer to encode a length. The first octet has the high-order bit set (to specify long form), and contains an integer K in the low-order 7 bits ($K > 1$). The next K octets contain a binary integer that specifies the length of the object that follows. Thus, to extract a long-form length, a program first



reads the single-octet, finds K, and then reads a K-octet binary number. Function alreadlen performs the operation. Function alwrtielen, found in the same file, creates the ASN.1 encoding of a length. It handles the short form by storing the length directly, and it handles the long form for integers that require either one or two octets (i.e., it only handles integers less than 65,556). Restricting an SNMP object to a length of less than 64K is reasonable because a larger object could not fit into an IP datagram.

```
/* alrlen.c - alreadlen, alwritelen */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <asn1.h>

/*
 *-----*
 * alreadlen - read and return the length of an ASN.1 encoded object
 *-----*
 */
int alreadlen(pack, lenlen)
unsigned char *pack;
int      *lenlen; /* length of length specification */
{
    int totlen;
    int i;

    /* if the high bit is NOT set, then len is in short form */
    if (!((*pack) & CHAR_HIBIT)) {
        *lenlen = 1;
        return (*pack) & ~CHAR_HIBIT; /* use only low bits */
    }
    /*
     * else, using long form where bit 7 = 1, and bits 6 - 0 encode
     * the number of subsequent octets that specify the length
     */
    *lenlen = (*pack++ & ~CHAR_HIBIT) + 1;

    for (i = 0, totlen = 0; i < (*lenlen) - 1; i++)
        totlen = (totlen << CHAR_BITS) | (int) *pack++;
    return totlen;
}
```



```
/*
 *-----*
 * alwritelen - write the length of an object in ASN.1 encoded form
 *-----*
 */
int alwritelen(pp, len)      /* return number of bytes required */
u_char    *pp;
int len;
{
    /* if len < 128 then use short form */
    if (len < CHAR_HIBIT) {
        *pp = len;
        return 1;
    }
    /* use long form, where bit 7 = 1, and bits 6 - 0 encode the
       number of subsequent octets that specify the length */
    if (len <= 255) {
        *pp++ = CHAR_HIBIT | 1;
        *pp = len & 0xff;
        return 2;
    }
    /* else, assume len <= 65535 (2^16 - 1) */
    *pp++ = CHAR_HIBIT | 2;
    *pp++ = len >> CHAR_BITS;
    *pp = len & 0xff;
    return 3;
}
```

20.17.2 Converting Integers To ASN.1 Form

ASN.1 represents all values as variable-length fields. To encode an integer, one must find the number of significant (nonzero) bytes in it, and then copy that many bytes into the encoding. Functions alreadint and alwriteint perform the translation.

```
/* alrwint.c - alreadint, alwriteint */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <asn1.h>

#define h2asn blkcopy
```



```
/*-----
 * alreadint - convert an ASN.1 encoded integer into a machine integer
 *-----
 */
int alreadint(pack, len)
u_char    *pack;
int  len;
{
    register int  tot;
    u_char        neg;
    int          tlen;

    if ((tlen = len) > sizeof(int))
        return 0;
    tot = *pack & ~CHAR_HIBIT;
    neg = *pack & CHAR_HIBIT;
    for (tlen--, pack++ ; tlen > 0 ; tlen--, pack++)
        tot = (tot << CHAR_BITS) | (int) *pack;
    if (neg)
        tot -= (1 << ((len * CHAR_BITS) - 1));
    return tot;
}

/*-----
 * alwriteint - convert an integer into its ASN.1 encoded form
 *-----
 */
int alwriteint(val, buffp, altype)
int  val;
u_char    *buffp;
int  altype;
{
    unsigned tmp, numbytes;
    register u_char    *bp;

    bp = buffp;
    tmp = val;
    if (A1_SIGNED(altype) && val < 0)
        tmp = -val;
    if (tmp < (unsigned) CHAR_HIBIT)
        *bp++ = numbytes = (u_char) 1;
```



```
        else if (tmp < (unsigned) BYTE2_HIBIT)
            *bp++ = numbytes = (u_char) 2;
        else if (tmp < (unsigned) BYTE3_HIBIT)
            *bp++ = numbytes = (u_char) 3;
        else if (tmp < (unsigned) BYTE4_HIBIT)
            *bp++ = numbytes = (u_char) 4;
        else { /* 5 bytes for unsigned with high bit set */
            *bp++ = (u_char) 5; /* length */
            *bp++ = (u_char) 0;
            numbytes = 4;
        }
        h2asn(bp, ((char *) &val) + (sizeof(int) - numbytes), numbytes);
        bp += numbytes;
        return bp - buffp;
    }
}
```

20.17.3 Converting Object Ids To ASN.1 Form

Internally, the software uses structure oid (defined in file snmp.h) to store an ASN.1 object id. Procedures a1readoid and a1writeoid convert from standard ASN.1 to the internal representation and vice versa.

```
/* alrwoi.c - a1readoid, a1writeoid */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <asn1.h>

char MIB_PREFIX[] = { 0x2b, 0x6, 0x1, 0x2, 0x1 }; /* 1.3.6.1.2.1 */

/*
*-----*
* a1readoid - convert an ASN.1 encoded object id into internal form
*-----*
*/
int a1readoid(pack, objidlen, objid)
unsigned char *pack;
int      objidlen;
struct oid   *objid;
{
    int val;
    u_char   *pp;
```



```
objid->len = 0;
pp = pack;

/* verify the required 1.3.6.1.2.1 prefix */
if (! blkequ(MIB_PREFIX, pp, MIB_PREF_SZ))
    return SYSERR;
pp += MIB_PREF_SZ;

for (; pp < pack + objidlen; objid->len++) {
    if (! (*pp & CHAR_HIBIT)) {
        objid->id[objid->len] = *pp++;
        continue;
    }
    /*
     * using long form, where bits 6 - 0 of each
     * octet are used; (bit 7 == 0) ==> last octet
     */
    val = 0;
    do
        val = (val << 7) | (int) (*pp & ~CHAR_HIBIT);
    while (*pp++ & CHAR_HIBIT); /* high bit set */
    objid->id[objid->len] = val;
}
return OK;
}

/*
 * alwriteoid - convert an object id into ASN.1 encoded form
 */
int alwriteoid(packp, oidp)
unsigned char *packp;
struct oid    *oidp;
{
    register u_char      *pp;
    int          i;
    u_char       *objidp, *lenp;

    pp = packp;
    lenp = pp++; /* save location of objid len */
```



```
objidp = pp;
/* prepend the standard MIB prefix. */
blkcopy(pp, MIB_PREFIX, MIB_PREF_SZ);
pp += MIB_PREF_SZ;

for (i=0; i < oidp->len; i++)
    if (oidp->id[i] < CHAR_HIBIT)      /* short form */
        *pp++ = oidp->id[i];
    else {                                /* long form */
        if (oidp->id[i] >= (u_short) (BYTE2_HIBIT >> 1))
            *pp++ = (u_char) (oidp->id[i] >> 14) |
                CHAR_HIBIT;
        *pp++ = (u_char) (oidp->id[i] >> 7) | CHAR_HIBIT;
        *pp++ = (u_char) (oidp->id[i] & ~CHAR_HIBIT);
    }
*lenp = pp - objidp;    /* assign the length of the objid */
return pp - packp;
}
```

Because all object ids that the client and server manipulate begin with the same prefix, the software can improve efficiency by removing the prefix on input and adding the prefix on output. The ASN.1 representation makes prefix recognition difficult because it encodes the first two numeric labels of an object identifier in a single octet. Thus, the prefix:

1. 3. 5. 1. 2. 1

is encoded into a 5-octet string that begins with 43^① (hexadecimal 0x2b) followed by octets that contain 6, 1, 2 and 1.

If any label in the object id is greater than 127^②, ASN.1 uses an extended representation to store the value. Only 7 bits in each octet contain data; ASN.1 uses the high-order bit to mark the end of subidentifiers. Figure 20.5 illustrates the encoding.

^① ASN.1 combines the first two subidentifiers, a and b, using the expression $a*40+b$.

^② None of the ASN.1 names permanently assigned to MIB variables currently has a numeric label greater than 127. However, object identifiers for some tables can contain larger values.



Figure 20.5 Part of an ASN.1 object identifier, shown in decimal, binary, and encoded form. The ASN.1 encoding uses multiple octets to encode integers greater than 127. A zero in the high-order bit of an octet marks the end of a subidentifier.

20.17.4 A Generic Routine For Converting Values

Now that we have reviewed low-level routines that convert values between the internal representation and the ASN.1 representation used in SNMP messages, we can understand how they are used. Function alreadval takes an argument that specifies an ASN.1 object type as well as a pointer to a value. It uses the type to decide which conversion routine to use to translate the item from ASN.1 to internal form. A related routine, alwriteval performs the translation for output. Both routines use the functions described earlier in this chapter to perform the conversion.

```
/* alrwval.c - alreadval, alwriteval */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <asn1.h>

/*
 * alreadval - convert object in ASN.1 encoded form into internal form
 */
int alreadval(val, type, vallen, pack)
struct snval *val;
int type;
int vallen;
unsigned char *pack;
```



```
{  
    val->sv_type = type;  
  
    switch (type) {  
        case ASN1_INT:  
        case ASN1_COUNTER:  
        case ASN1_GAUGE:  
        case ASN1_TIMETICKS:  
            val->sv_val.sv_int = alreadint(pack, vallen);  
            return OK;  
        case ASN1_OCTSTR:  
            val->sv_val.sv_str.sv_len = vallen;  
            val->sv_val.sv_str.sv_str = (char *) pack;  
            return OK;  
        case ASN1_NULL:  
            return OK;  
        case ASN1_OBJID:  
            return alreadoid(pack, vallen, &(val->sv_val.sv_oid));  
        case ASN1_IPADDR:  
            blkcopy(val->sv_val.sv_ipaddr, pack, vallen);  
            return OK;  
        default:  
            return SYSERR;  
    }  
}  
  
/*-----  
 * alwriteval - convert the value of a variable into ASN.1 equivalent.  
 *-----  
 */  
int alwriteval(bl, bp) /* Return number of bytes required. */  
struct snbentry *bl;  
u_char *bp;  
{  
    u_char *origbp;  
  
    origbp = bp;  
    *bp++ = SVTYPE(bl);  
  
    switch(SVTYPE(bl)) {  
        case ASN1_INT:
```



```
case ASN1_COUNTER:
case ASN1_GAUGE:
case ASN1_TIMETICKS:
    bp += alwriteint(SVINT(bl), bp, SVTYPE(bl));
    break;
case ASN1_NULL:
    *bp++ = (u_char) 0;
    break;
case ASN1_OCTSTR:
    bp += alwritelen(bp, SVSTRLEN(bl));
    blkcopy(bp, SVSTR(bl), SVSTRLEN(bl));
    bp += SVSTRLEN(bl);
    freemem(SVSTR(bl), SVSTRLEN(bl));
    break;
case ASN1_IPADDR:
    *bp++ = IP_ALEN;
    blkcopy(bp, SVIPADDR(bl), IP_ALEN);
    bp += IP_ALEN;
    break;
case ASN1_OBJID:
    bp += alwriteoid(bp, &bl->sb_val.sv_val.sv_oid);
    break;
default:
    break;
}

return bp - origbp;
}
```

20.18 Summary

SNMP allows managers to interrogate or control gateways. A manager invokes client software that contacts one or more SNMP servers that operate on remote gateways. SNMP uses a fetch/store paradigm in which the server maintains a conceptual set of variables that map onto TCP/IP data structures in the gateway.

The Management Information Base (MIB) specifies a set of variables that gateways must maintain. SNMP uses ASN.1 syntax to represent messages, and ASN.1 object identifiers to name MIB variables. When a server receives a message, it must map the numeric representation of the ASN.1 variable names into local variables that store the corresponding values. Our implementation keeps an array of MIB variables, but uses a hash table to speed lookup. In addition, it uses pointers to maintain the lexical ordering



among names needed for get-next-request operations.

We examined functions that convert values from the ASN.I representation used in SNMP messages to the internal form used by the gateway. Although ASN.1 syntax is tedious, it is not difficult.

20.19 FOR FURTHER STUDY

Case, Fedor, Schoffstall, and Davin [RFC 1157] contains the standard for SNMP. ISO [May 87a] and [May 87b] contain the standard for ASN.1 and specify the encoding. McCloghrie and Rose [RFC 1156] specifies the MIB, while McCloghrie and Rose [RFC 1155] contains the SMI rules for naming MIB variables. Rose [RFC 1158] proposes a MIB-II for use with SNMP.

20.20 EXERCISES

1. Suppose the current set of MIB variables were numbered sequentially from 1 through 89 instead of assigned ASN.1 object identifiers. How much code could be eliminated?
2. Compare the ASN.1 hierarchical naming scheme to the numbering scheme suggested in the previous exercise. What are the advantages and disadvantages of each scheme?
3. What are the advantages and disadvantages of assigning pointers for the lexicographic ordering among MIB variables in the declarations in file snmib.c?
4. Consider macro oidequ defined in file snmp.h. Why does it check the object lengths explicitly?
5. Read the protocol specification to find out what the community field represents. Why does the server send and expect the value public in this field?
6. Read the standard to find out how the first two labels of an ASN.1 object identifier are encoded into a single octet. Why does ASN.1 specify such an encoding?
7. Explain why procedures like a1readint are inherently machine dependent. How might a1readint change for a different architecture?



21 SNMP: Client And Server

21.1 Introduction

The previous chapter described SNMP software, and showed data structures used to look up and bind ASN.1 names for MIB variables to variables on the local gateway. This chapter describes the implementation of an SNMP server, and shows how it parses messages. In addition, it examines an SNMP client that shares many of the utility procedures used by the server. The next chapter continues the discussion by concentrating on procedures that implement fetch and store operations for specific tables.

21.2 Data Representation In The Server

In principle, an SNMP server is simple. It consists of a single process that repeatedly waits for an incoming message, performs the specified operations, and returns the result. The most important part of the message consists of a sequence of get, set, and get-next requests^①, along with the ASN.1 object identifier to which each should be applied. The server looks up each identifier in the sequence, and applies the specified operation to it.

In practice, many small details complicate the code. The message itself uses ASN.1 representation for all fields, so the server cannot use a fixed structure to describe the message format. Instead, it must step through the message, parsing each field as it goes. Furthermore, because each value is represented in ASN.1 format, the server must translate them to the appropriate internal form.

To simplify the procedures that implement individual fetch or store operations, our server uses the local machine's native encoding for values like integers. It parses each incoming message, and translates the fields from ASN.1 representation to an internal data structure. The server manipulates the internal form, and translates it back to the

^① Throughout the remainder of the text, we will abbreviate get-request, set-request, and get-next-request to get, set, and get-next.



required ASN.1 representation before sending a reply. We can summarize:

To optimize performance, the SNMP server translates incoming messages from ASN.1 representation to an internal, fixed-field form that stores most values using the native hardware representation. It performs operations using the internal representation, and translates back to the ASN.1 representation before sending a reply.

21.3 Server Implementation

Procedure snmpd implements the main SNMP server algorithm. File `snmp.h`^① contains declarations for the data structures it uses.

```
/* snmpd.c - snmpd */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <asn1.h>

/*
 * snmpd - open the SNMP port and handle incoming queries
 */
int snmpd()
{
    int      snmpdev, len;
    struct xgram  *query;
    struct req_desc    rqd;

    sninit();
    query = (struct xgram *) getmem(sizeof (struct xgram));
    /* open the SNMP server port */
    if ((snmpdev = open(UDP, ANYFPORT, SNMPPORT)) == SYSERR)
        return SYSERR;
    while (TRUE) {
        /*
         * In this mode, give read the size of xgram, it returns
         * number of bytes of *data* in xgram.
```

^① See page 461 for a listing of `snmp.h`.



```
*/  
len = read(snmpdev, query, sizeof(struct xgram));  
/* parse the packet into the request desc. structure */  
if (snparse(&rqd, query->xg_data, len) == SYSERR) {  
    snfreebl(rqd.bindlf);  
    continue;  
}  
/* convert ASN.1 representations to internal forms */  
if (sna2b(&rqd) == SYSERR) {  
    snfreebl(rqd.bindlf);  
    continue;  
}  
if (snrslv(&rqd) == SYSERR) {  
    query->xg_data[rqd.pdutype_pos] = PDU_RESP;  
    query->xg_data[rqd.err_stat_pos] = rqd.err_stat;  
    query->xg_data[rqd.err_idx_pos] = rqd.err_idx;  
    if (write(snmpdev, query, len) == SYSERR)  
        return SYSERR;  
    snfreebl(rqd.bindlf);  
    continue;  
}  
len = mksnmp(&rqd, query->xg_data, PDU_RESP);  
if (len == SYSERR) {  
    query->xg_data[rqd.pdutype_pos] = PDU_RESP;  
    query->xg_data[rqd.err_stat_pos] = rqd.err_stat;  
    query->xg_data[rqd.err_idx_pos] = rqd.err_idx;  
    if (write(snmpdev, query, len) == SYSERR)  
        return SYSERR;  
    snfreebl(rqd.bindlf);  
    continue;  
}  
if (write(snmpdev, query, len) == SYSERR)  
    return SYSERR;  
snfreebl(rqd.bindlf);  
}  
}
```

Snmpd begins by opening the UDP port SNMP uses (constant SNMPPORT in the code). It then enters an infinite loop in which it calls read to wait for the next incoming message. When a message arrives, snmpd calls snparse to parse and convert the message to its internal form, and store it in the request data structure (req_desc). In addition to



extracting fields in the header, snmpd calls function sna2b to extract the sequence of object identifiers from the message and convert them into a linked list. Nodes on the list each correspond to one binding: they are defined by structure snbentry in file snmp.h. During the conversion, sna2b translates each ASN.1 object identifier to an internal representation.

Once it has converted the message and list of names to internal form, snmpd calls snrslv to resolve the query. Resolution consists of performing the specified get, set, or get-next operation for each identifier in the list. After completing the resolution, snmpd calls mksnmp to form a reply message, and write to send the reply to the client. Once the server finishes sending a reply, it calls snfreebl to free the linked list of names, and returns to the beginning of the main loop to await the next incoming message.

If an error prevents successful resolution, the server creates an error reply by storing an error type code and an error index in the message. The error type code gives the reason for the error, and the error index specifies the name in the query that caused the error.

21.4 Parsing An SNMP Message

Function snparse decodes an SNMP message by extracting its fields. Because the ASN.1 representation allows each field to have a variable size, the task is tedious. The parser must move a pointer through all fields in the message, finding the field length and extracting the value.

```
/* snparse.c - snparse */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <asn1.h>

/*
 * snparse - convert the ASN.1-encoded SNMP packet into internal form
 */
int snparse(rqdp, snmppack, len)
struct req_desc    *rqdp;
u_char            *snmppack;
int      len;
{
    struct snbentry *bl, *lastbl;
    register u_char *packp;
```



```
int      totpacklen, commlen, pdulen, totbindlen, lenlen;
int      varbindlen;
u_char   *packendp;

packp = snmppack;
packendp = snmppack + len;

/* sequence operator and total packet length */
if (*packp++ != ASN1_SEQ ||
    (totpacklen = alreadlen(packp, &lenlen)) < 0)
    return SYSERR;
packp += lenlen;

/* verify total length, version, community */
if (packendp != packp + totpacklen ||
    *packp++ != ASN1_INT ||
    *packp++ != SVERS_LEN ||
    *packp++ != SVERSION ||
    *packp++ != ASN1_OCTSTR ||
    (commlen = alreadlen(packp, &lenlen)) < 0)
    return SYSERR;
packp += lenlen;
if (strncmp(packp, SCOMM_STR, commlen) != 0)
    return SYSERR;
packp += commlen;

/* PDU type and length */
if (*packp == PDU_TRAP)
    return SYSERR;
rqdp->pdutype_pos = packp - snmppack;
rqdp->reqtype = *packp++;
if ((pdulen = alreadlen(packp, &lenlen)) < 0)
    return SYSERR;
packp += lenlen;

/* verify PDU length */
if (packendp != packp + pdulen)
    return SYSERR;
/* request id */
if (*packp++ != ASN1_INT ||
    (rqdp->reqidlen = alreadlen(packp, &lenlen)) < 0)
```



```
        return SYSERR;

    packp += lenlen;
    blkcopy(rqdp->reqid, packp, rqdp->reqidlen);
    packp += rqdp->reqidlen;

    /* error status */
    if (*packp++ != ASN1_INT || *packp++ != 1)
        return SYSERR;

    rqdp->err_stat = *packp;
    rqdp->err_stat_pos = packp++ - snmppack;

    /* error index */
    if (*packp++ != ASN1_INT || *packp++ != 1)
        return SYSERR;

    rqdp->err_idx = *packp;
    rqdp->err_idx_pos = packp++ - snmppack;

    /* sequence of variable bindings */
    if (*packp++ != ASN1_SEQ ||
        (totbindlen = alreadlen(packp, &lenlen)) < 0)
        return SYSERR;
    packp += lenlen;

    /* verify variable bindings length */
    if (packendp != packp + totbindlen)
        return SYSERR;

    /* build doubly-linked bindings list; fill in only sb_alstr's */
    rqdp->bindlf = rqdp->bindle = (struct snbentry *) NULL;
    do {
        bl = (struct snbentry *) getmem(sizeof(struct snbentry));
        bl->sb_next = 0;
        bl->sb_prev = 0;
        if (rqdp->bindlf) {
            lastbl->sb_next = bl;
            bl->sb_prev = lastbl;
            lastbl = bl;
        } else
            lastbl = rqdp->bindlf = bl;
        bl->sb_alstr = packp;
        if (*packp++ != ASN1_SEQ ||
            (varbindlen = alreadlen(packp, &lenlen)) < 0)
```



```
        return SYSERR;

    packp += lenlen;
    bl->sb_alslen = varbindlen;
    packp += varbindlen;
} while (packp < packendp);
/* check that the entire packet has now been parsed */
if (packp != packendp)
    return SYSERR;
rqdp->bndle = lastbl;
return OK;
}
```

An SNMP message begins with a sequence operator and a total message length. If the length does not agree with the length of the datagram, snparse returns an error code. Snparse then verifies that the second field is an integer that contains the correct SNMP version number. The third field is known as a community field, and is used for authentication. Our implementation honors the community string public, which is the standard value for servers that do not require clients to authenticate themselves.

Snparse verifies that the fourth field does not specify a trap operation (i.e., that it is a get, set, or a get-next request). Servers, not clients, generate trap messages.

Snparse checks the fifth field to make sure it correctly specifies the length of the remaining message, and then extracts the request identification from the sixth field. It also verifies that the error status and error index in the seventh and eighth fields have not been set.

After checking the error status and index fields, snparse reaches a sequence of bindings. In both requests and responses, each binding consists of a pain. In a get request or get-next request, each binding specifies a variable name (ASN.1 object identifier) and the associated value NULL. In a response, the server replaces NULL values with the values requested by the client. In a set request, the bindings specify nonnull values for each name; the server assigns these values to the specified variables.

Snparse iterates through the sequence of bindings and allocates an snbentry node for each binding. Each node contains a pointer to the ASN.1 representation of the object identifier in the original message. The node is linked into the list that has a head and tail pointer in the message's req_desc structure as Figure 21.1 illustrates.

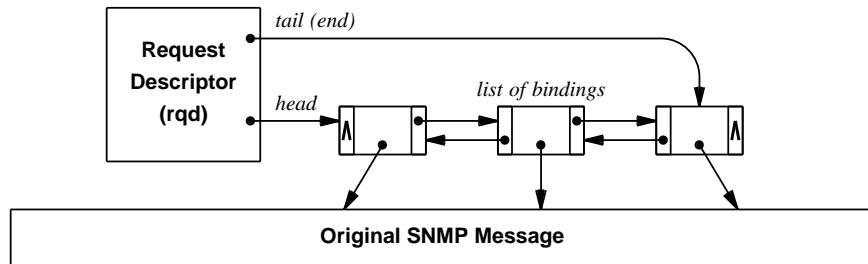


Figure 21.1 The data structure created by snparse. Each node on the binding list corresponds to a MIB variable. The node points to an ASN.1 object identifier found in the original message.

Once snparse finishes extracting the expected fields from the message, it verifies that no additional octets remain.

21.5 Converting ASN.1 Names In The Binding List

After snparse finishes extracting fields from an incoming message, structure req_desc contains the contents. To make the object names easier to access, the server calls function sna2b to convert object identifiers and values from the message into fixed-length internal representations. Sna2b iterates through the linked list of bindings, and parses the ASN.1 syntax for each. It calls a1readoid to extract the suffix of the object id and store it in field sb_oid of the node. It also extracts the associated value from the original message and stores it in field sb_val.

```
/* sna2b.c - sna2b */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <asn1.h>

/*
 * sna2b - convert an ASN.1 encoded binding into internal form
 */
int sna2b(rqdp)
struct req_desc *rqdp;
{
```



```
register u_char      *sp;
struct snbentry *bl;
int      lenlen, objidlen, vallen;
u_char      type;

for (bl = rqdp->bindlf; bl; bl = bl->sb_next) {
    sp = bl->sb_alstr;
    /* match the sequence operator and length of bindings */
    if (*sp++ != ASN1_SEQ || alreadlen(sp, &lenlen) < 0)
        return SYSERR;
    sp += lenlen;
    /* object identifier type, length, objid */
    if (*sp++ != ASN1_OBJID ||
        (objidlen = alreadlen(sp, &lenlen)) < 0)
        return SYSERR;
    sp += lenlen;
    if (alreadoid(sp, objidlen, &bl->sb_oid) == SYSERR)
        return SYSERR;
    sp += objidlen;
    /* object's value */
    type = *sp++;
    if ((vallen = alreadlen(sp, &lenlen)) < 0)
        return SYSERR;
    sp += lenlen;
    if (alreadval(&bl->sb_val, type, vallen, sp) == SYSERR)
        return SYSERR;
    sp += vallen;
    bl->sb_alslen = 0;
}
return OK;
}
```

21.6 Resolving A Query

Once an incoming message has been converted to internal form, the server calls snrslv to resolve it.

```
/* snrslv.c - snrslv */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```



```
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/* Set the error status and error index in a request descriptor. */
#define seterr(errval)      rqdp->err_stat = errval;    \
                        rqdp->err_idx = i;

/*-----
 * snrslv - resolve the list of specified variable bindings
 *-----
 */

snrslv(rqdp)
struct req_desc *rqdp;
{
    struct snbentry *bl;
    struct mib_info *np, *getmib();
    int i, op, err;

    for (bl = rqdp->bindlf, i = 1; bl; bl = bl->sb_next, i++) {
        /* use getmib to look up object id */
        if ((np = getmib(&bl->sb_oid)) == 0) {
            seterr(SERR_NO_SUCH);
            return SYSERR;
        }
        /* call function to apply specified operation */
        if (np->mi_func == 0) { /* objid is an aggregate */
            /* only getnext allows nonexistent names */
            if (rqdp->reqtype != PDU_GETN) {
                seterr(SERR_NO_SUCH);
                return SYSERR;
            }
            /* use getfirst for getnext on an aggregate */
            if (err = ((*np->mi_next->mi_func)
                       (bl, np->mi_next, SOP_GETF))) {
                seterr(err);
                return SYSERR;
            }
        } else { /* function in table ==> single item or table */
            switch (rqdp->reqtype) {
            case PDU_GET: op = SOP_GET; break;
```



```
        case PDU_GETN:      op = SOP_GETN;      break;
        case PDU_SET: op = SOP_SET; break;
    }
    /* use getfirst for getnext on table entry */
    if (oidequ(&bl->sb_oid, &np->mi_objid) &&
        np->mi_vartype == T_TABLE) {
        if (op == SOP_GETN)
            op = SOP_GETF;
    }
    if (err = ((*np->mi_func)(bl, np, op))) {
        seterr(err);
        return SYSERR;
    }
}
return OK;
}
```

Resolution consists of applying the operation specified in the request to each variable on the binding list. Thus, snrslv iterates through the binding list. It calls getmib to look up each object identifier in the mib array. If no match can be found, snrslv returns an error.

Once it has found an entry for the object, snrslv examines field mi_func in the entry. If the field contains NULL, the entry corresponds to an aggregate instead of a variable. Because aggregate names can only be used in get-next requests, snrslv returns an error unless the requested operation is get-next. If field mi_func does not contain NULL, it contains the address of a function that interprets operations for the variable.

21.7 Interpreting The Get-Next Operation

In the case of a get-next request, snrslv changes the operation to get-first, and applies it to the lexically next item in the MIB. Get-first is part of the implementation, and not part of the SNMP protocol. To understand why get-first arises, consider the semantics of get-next carefully. When get-next specifies an object identifier, the server must apply the request to the MIB variable that lexically follows the specified name. If the server follows the lexical pointer in the MIB table to find the next name, and then attempts to apply get-next to the new item, an infinite iteration results. On the other hand, if the server looks up the identifier, follows the lexical pointer to the next item, and applies the get operation, the operation will fail if the lexically next item corresponds to an empty table instead of a simple variable. Thus, the server must follow lexical pointers



until it finds a simple variable. The example code uses the get-first operation to do exactly that. When applied to a simple variable, get-first is the same as get. When applied to an aggregate, however, get-first is the same as get-next. To summarize:

The get-first operation accommodates get-next requests for an arbitrary lexical ordering of MIB items by finding the next variable in the lexical order to which the get operation applies. It then applies the get request to that item.

To understand how snrslv uses get-first, study the code again. If the specified name corresponds to an aggregate, snrslv finds field mi_func empty, so it applies get-first to the lexically next item on the list. If it finds a table (mi_vartype contains T_TABLE), snrslv changes a get-next operation into a get_first before calling the function that implements operations.

21.8 Indirect Application Of Operations

We have seen that snrslv consults the mib array to determine which function to call to apply an operation to a given variable. It passes an operation code, a pointer to the node on the binding list, and a pointer to the mib entry that corresponds to the variable. The purpose of using indirection is to avoid the duplication that results from building separate functions for each MIB variable. Instead, a few functions contain all the code needed for error checking and common operations. For example, function snleaf handles operations on simple variables.

```
/* snleaf.c - snleaf */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 * snleaf - perform the requested operation on the leaf SNMP variable
 */
int snleaf(bindl, mip, op)
struct snbentry *bindl;
struct mib_info *mip;
int      op;
```



```
{  
    int      len;  
    char     *strp;  
    struct oid   *oip;  
  
    if (op == SOP_GETN) {  
        if (mip->mi_next)  
            return((*mip->mi_next->mi_func)  
                   (bindl, mip->mi_next, SOP_GETF));  
        return SERR_NO_SUCH;  
    }  
    if (op == SOP_SET) {  
        if (! mip->mi_writable)  
            return SERR_NO_SUCH;  
        switch(mip->mi_vartype) {  
            case ASN1_INT:  
                if (SVTYPE(bindl) != ASN1_INT)  
                    return SERR_BAD_VALUE;  
                if (mip->mi_param == 0)  
                    return SERR_NO_SUCH;  
                *((int *) mip->mi_param) = SVINT(bindl);  
                break;  
            case ASN1_OCTSTR:  
                if (SVTYPE(bindl) != ASN1_OCTSTR)  
                    return SERR_BAD_VALUE;  
                strp = *(char **) mip->mi_param;  
                blkcopy(strp, SVSTR(bindl), SVSTRLEN(bindl));  
                *(strp + SVSTRLEN(bindl)) = '\0';  
                break;  
            case ASN1_OBJID:  
                if (SVTYPE(bindl) != ASN1_OBJID)  
                    return SERR_BAD_VALUE;  
                oip = (struct oid *) mip->mi_param;  
                oip->len = SVSTRLEN(bindl);  
                blkcopy(oip->id, SVSTR(bindl), oip->len * 2);  
                break;  
        }  
        return SNMP_OK;  
    }  
    if (op == SOP_GETF) {  
        /* put the correct objid into the binding list. */  
    }  
}
```



```
bindl->sb_oid.len = mip->mi_objid.len;
blkcopy(bindl->sb_oid.id, mip->mi_objid.id,
        mip->mi_objid.len * 2);
}

SVTYPE(bindl) = mip->mi_vartype;

switch(mip->mi_vartype) {
case ASN1_INT:
case ASN1_TIMETICKS:
case ASN1_GAUGE:
case ASN1_COUNTER:
    SVINT(bindl) = *((int *) mip->mi_param);
    break;
case ASN1_OCTSTR:
    strp = *(char **) mip->mi_param;
    if (strp == NULL) {
        SVSTRLEN(bindl) = 0;
        SVSTR(bindl) = NULL;
        break;
    }
    len = SVSTRLEN(bindl) = strlen(strp);
    SVSTR(bindl) = (char *) getmem(len);
    blkcopy(SVSTR(bindl), strp, len);
    break;
case ASN1_OBJID:
    oip = (struct oid *) mip->mi_param;
    SVOIDLEN(bindl) = oip->len;
    blkcopy(SVOID(bindl), oip->id, oip->len * 2);
    break;
}
return SNMP_OK;
}
```

Snleaf tests the operation argument to choose an action. It handles get-next operations by invoking get-first on the succeeding item in the lexical order. It handles set operations by storing the value from the binding list node in the appropriate variable in memory (field mi_param in the mib table specifies the memory address). It handles get (and get-first) requests by copying the value from the appropriate variable in memory into the binding list node.



21.9 Indirection For Tables

In the mib array, entries that correspond to tables use field mi_param to store a pointer to the table's tabtab entry. The tabtab entry for a given table contains pointers to functions that implement each of the SNMP operations on that table. Thus, procedure sntable uses the tabtab entry and the specified operation to choose and invoke a function that implements that operation on a specific table.

```
/* sntable.c - sntable */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>

/*-----
 * sntable - call function to operate on a table-embedded variable
 *-----
 */

int sntable(bindl, mip, op)
struct snbentry      *bindl;
struct mib_info      *mip;
int      op;
{
    int  numifaces = Net.nif - 1;

    /*
     * mip->mi_param holds a pointer to an entry in tabtab that
     * contains the pointers to functions for each table
     */
    switch (op) {
    case SOP_GET:
        return (*mip->mi_param->ti_get)(bindl, numifaces);
    case SOP_GETF:
        return (*mip->mi_param->ti_getf)(bindl, mip, numifaces);
    case SOP_SET:
        return (*mip->mi_param->ti_set)(bindl, mip, numifaces);
    case SOP_GETN:
        return (*mip->mi_param->ti_getn)(bindl, mip, numifaces);
    }
    return SYSERR;
```



}

21.10 Generating A Reply Message Backward

Once the server has resolved the entries in a request, it creates a reply and sends the reply back to the client. Reply messages have the same format as request messages, with each field using an ASN.1 representation. The representation requires the header to contain the message length, which cannot be known until the representation of each field has been computed. Furthermore, because the size of the message length field itself depends on the size of the remainder of the message, it is impossible to know how much space to skip for the length field when constructing the message.

To simplify message construction, our code avoids the problem of unknown lengths by building the message backward. It generates fields in reverse order, and within fields, it generates octets in reverse order. Thus, once the entire message has been generated, it is simply reversed for transmission. To summarize:

The ASN.1 representation makes message generation difficult because all fields, including message length fields, use a variable-size format. To simplify construction, the message is generated backward and reversed before transmission.

Procedure mksnmp handles creation of both request messages and response messages, and both the client and server invoke it.

```
/* mksnmp.c - mksnmp */

#include <conf.h>
#include <kernel.h>
#include <sleep.h>
#include <network.h>
#include <snmp.h>
#include <asn1.h>

#define SNMAXHLEN 32 /* length of a "maximum" SNMP header */

u_char    snmpbuff[SNMPMAXSZ]; /* global scratch buffer */

/*-----
 * mksnmp - make an snmp packet and return its length
 *-----
 */
```



```
int mksnmp(rqdp, snmppack, pdutype)
struct req_desc      *rqdp;
u_char          *snmppack;
u_char          pdutype;
{
    register u_char          *pp, *cp;
    struct  snbentry *bl;
    u_char          tmpbuff[40];
    int             len, mtu, estlen;

    pp = snmpbuff;
    if (rqdp->reqidlen == 0) { /* if id len == 0, get new reqid */
        blkcopy(rqdp->reqid, (char *) &clktime, sizeof(clktime));
        rqdp->reqidlen = sizeof(clktime);
    }
    snb2a(rqdp); /* convert bindings to ASN.1 notation */

    /* check total length of the packet to be created */
    mtu = IP_MAXLEN;

    /* add up total length of ASN.1 representations of variables */
    for (estlen=0, bl=rqdp->bindlf; estlen<mtu && bl; bl=bl->sb_next)
        estlen += bl->sb_alslen;
    /*
     * if too long, or if adding the header makes it too long,
     * set error status to tooBig and return
     */
    if (bl || (estlen + SNMAXHLEN >= mtu)) {
        rqdp->err_stat = SERR_TOO_BIG;
        rqdp->err_idx = 0;
        return SYSERR;
    }
    /* go backwards through snbentry, writing out the bindings */
    for (bl=rqdp->bindle; bl; bl=bl->sb_prev) {
        cp = &bl->sb_alstr[bl->sb_alslen-1];
        while (cp >= bl->sb_alstr)
            *pp++ = *cp--;
    }
    /* write the length of the bindings and an ASN1_SEQ type */
    len = alwritelen(tmpbuff, pp - snmpbuff);
    for (cp = &tmpbuff[len-1]; cp >= tmpbuff; )
```



```
*pp++ = *cp--;
*pp++ = ASN1_SEQ;

/* write the error index and error status -- 1 byte integers */
*pp++ = (u_char) rqdp->err_idx;
*pp++ = (u_char) 1;
*pp++ = ASN1_INT;
*pp++ = (u_char) rqdp->err_stat;
*pp++ = (u_char) 1;
*pp++ = ASN1_INT;

/* write the request id, its length, and its type */
for (cp = &rqdp->reqid[rqdp->reqidlen-1]; cp >= rqdp->reqid; )
    *pp++ = *cp--;
*pp++ = rqdp->reqidlen;
*pp++ = ASN1_INT;

/* write the packet length and pdutype */
len = alwritelen(tmpbuff, pp - snmpbuff);
for (cp = &tmpbuff[len-1]; cp >= tmpbuff; )
    *pp++ = *cp--;
*pp++ = pdutype;

/* write the community and the version */
blkcopy(pp, SNVBACK, SNVCLEN);
pp += SNVCLEN;

/* write the total packet length */
len = alwritelen(tmpbuff, pp - snmpbuff);
for (cp = &tmpbuff[len-1]; cp >= tmpbuff; )
    *pp++ = *cp--;
*pp++ = ASN1_SEQ;

/* reverse the entire finished packet */
for (--pp, cp = snmppack; pp >= snmpbuff; )
    *cp++ = *pp--;
return cp - snmppack;
}
```

Mksnmp begins by checking the request id field (reqid). In a response, the id will contain whatever value the client sent in the request. For a request generated by a client,



the id field will be zero, so mksnmp uses the current time of day as a unique id.

After checking the request id, mksnmp calls snb2a to convert all object identifiers in the binding list to ASN.1 form. It then adds the resulting lengths and an estimate of the message header size to obtain an estimate of the total message length. If the estimate exceeds the maximum UDP datagram buffer size, mksnmp returns an error.

The formation of an outbound message parallels the recognition of an incoming message. Mksnmp takes fixed fields from structure req_desc, and converts each to its ASN.1 representation. Finally, it reverses the message.

21.11 Converting From Internal Form to ASN.1

Procedure snb2a provides the inverse of sna2; it converts an object identifier from internal form to its ASN.1 representation.

```
/* snb2a.c - snb2a */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <asn1.h>

extern u_char snmpbuff[];

/*-----
 * snb2a - convert the list of bindings from internal form into ASN.1
 *-----
 */
int snb2a(rqdp)
struct req_desc      *rqdp;
{
    register u_char *bp;
    int      len;
    struct snbentry   *bl;
    u_char      *ap;

    for (bl = rqdp->bindlf; bl; bl = bl->sb_next) {
        bp = snmpbuff;      /* used for temporary working space */
        *bp++ = ASN1_OBJID;
        bp += alwriteoid(bp, &bl->sb_oid);
        bp += alwriteval(bl, bp);
    }
}
```



```
/*
 * We need to allocate bytes in sb_alstr but can't do it
 * until we know how many bytes it takes to write the
 * length of the binding, so we write that length into
 * snmpbuff at the end of the binding. Then we can alloc
 * space, and transfer the data.
 */
len = alwritelen(bp, bp - snmpbuff);
bl->sb_alslen = bp - snmpbuff + len + 1;
ap = bl->sb_alstr = (u_char *) getmem(bl->sb_alslen);
*ap++ = ASN1_SEQ;
blkcopy(ap, bp, len); /* write in the length spec.*/
ap += len;
blkcopy(ap, snmpbuff, bp - snmpbuff);
}
}
```

21.12 Utility Functions Used By The Server

The server invokes utility procedure snfreebl to free the linked list of bindings after it sends a reply back to the client that issued a request.

```
/* snfreebl.c - snfreebl */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>

/*-----
 * snfreebl - free memory used for ASN.1 strings and snbentry nodes
 *-----
 */
snfreebl(bl)
struct snbentry     *bl;
{
    register struct snbentry     *pbl;

    if (bl == 0)
        return;
    for (pbl = bl, bl = bl->sb_next; bl; pbl = bl, bl = bl->sb_next) {
        freemem(pbl->sb_alstr, pbl->sb_alslen);
```



```
    freemem(pbl, sizeof(struct snbentry));
}
freemem(pbl->sb_alstr, pbl->sb_alslen);
freemem(pbl, sizeof(struct snbentry));
}
```

Given a pointer to the binding list, snfreebl moves along it and deallocates both the memory used to hold the ASN.1 form of the binding and the node itself.

21.13 Implementation Of An SNMP Client

An SNMP client must generate and send a request to a server, wait for a response, and verify that the response matches the request. Procedure snclient performs the client function. It accepts as an argument the address of a request descriptor that contains the information in the message, including the desired operation and a list of bindings to which the operation should be applied. It calls mksnmp to generate a message in ASN.1 representation, open to open a descriptor that can be used to send datagrams, and write to send the message.

```
/* snclient.c - snclient */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>

/*
 * snclient - send an SNMP request and wait for the response
 */
int snclient(rqdp, fport, stdout)
struct req_desc    *rqdp;
char      *fport;
int       stdout;
{
    struct snbentry    *bindl;
    u_char        buff[SNMPMAXSZ], reqidsave[10], reqidsavelen;
    int        snmpdev, len;

    rqdp->reqidlen = 0;
    rqdp->err_stat = 0;
```



```
rqdp->err_idx = 0;

if ((len = mksnmp(rqdp, buff, rqdp->reqtype)) == SYSERR)
    return SCL_BADREQ;
blkcopy(reqidsave, rqdp->reqid, rqdp->reqidlen);
reqidsavelen = rqdp->reqidlen;

/* open the SNMP port and put into data mode */
if ((snmpdev = open(UDP, fport, ANYLPORT)) == SYSERR ||
    control(snmpdev, DG_SETMODE, DG_DMODE | DG_TMODE) == SYSERR) {
    close(snmpdev);
    return SCL_OPENF;
}
if (write(snmpdev, buff, len) == SYSERR) {
    close(snmpdev);
    return SCL_WRITEF;
}
/* retry once, on timeout */
if ((len = read(snmpdev, buff, SNMPMAXSZ)) == TIMEOUT)
    len = read(snmpdev, buff, SNMPMAXSZ);
if (len == TIMEOUT) {
    close(snmpdev);
    return SCL_NORESP;
} else if (len == SYSERR) {
    close(snmpdev);
    return SCL_READF;
}
if (snparse(rqdp, buff, len) == SYSERR) {
    close(snmpdev);
    return SCL_BADRESP;
}
if (reqidsavelen != rqdp->reqidlen ||
    ! blkequ(reqidsave, rqdp->reqid, reqidsavelen)) {
    close(snmpdev);
    return SCL_BADRESP;
}
/* convert the sb_alstr's to objid's and their values */
if (sna2b(rqdp) == SYSERR) {
    close(snmpdev);
    return SCL_BADRESP;
}
```



```
    close(snmpdev);
    return SCL_OK;
}
```

Because UDP is unreliable, an SNMP client must implement its own strategy for timeout and retransmission. Our example client implements timeout, but only one retransmission. To do so, snclient calls control to place the UDP descriptor in timed mode (DG_TMODE). In timed mode, read operations either return a datagram or the special value TIMEOUT if the timer expires before any datagram arrives. If snclient does not receive a response within two timeout periods, it closes the descriptor and returns an error code.

If snclient does receive a response, it calls snparse to convert the response into internal form. It then compares the id field of the response to the id field of the request to verify that the message is a response to the request that was sent. If so, snclient calls sna2b to translate the ASN.1 representation of each object identifier to its internal form.

21.14 Initialization Of Variables

Procedure sninit initializes all simple counters and variables used by SNMP that are not initialized as part of the normal system startups. Sninit also calls hashinit to initialize the hash table used to optimize MIB name lookup. Of course, MIB variables that correspond to tables (e.g., the routing table) are initialized by the TCP/IP software.

```
/* sninit.c - sninit */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ctype.h>
#include <snmp.h>
#include <mib.h>
#include <snmpvars.h>

extern char vers[];

struct oid SysObjectID = { {0}, 1};

/* System & Interface MIB */
char      *SysDescr[256], SysContact[256], SysName[256], SysLocation[256];
unsigned SysUpTime, SysServices, IfNumber;
```



```
/* IP MIB */
unsigned IpForwarding, IpDefaultTTL, IpInReceives, IpInHdrErrors,
IpInAddrErrors, IpForwDatagrams, IpInUnknownProtos, IpInDiscards,
IpInDelivers, IpOutRequests, IpOutDiscards, IpOutNoRoutes,
IpReasmTimeout, IpReasmReqds, IpReasmOKs, IpReasmFails, IpFragOKs,
IpFragFails, IpFragCreates, IpRoutingDiscards;

/* ICMP MIB */
unsigned IcmpInMsgs, IcmpInErrors, IcmpInDestUnreachs, IcmpInTimeExcds,
IcmpInParmProbs, IcmpInSrcQuenches, IcmpInRedirects, IcmpInEchos,
IcmpInEchoReps, IcmpInTimestamps, IcmpInTimestampReps,
IcmpInAddrMasks, IcmpInAddrMaskReps, IcmpOutMsgs, IcmpOutErrors,
IcmpOutDestUnreachs, IcmpOutTimeExcds, IcmpOutParmProbs,
IcmpOutSrcQuenches, IcmpOutRedirects, IcmpOutEchos,
IcmpOutEchoReps, IcmpOutTimestamps, IcmpOutTimestampReps,
IcmpOutAddrMasks, IcmpOutAddrMaskReps;

/* UDP MIB */
unsigned UdpInDatagrams, UdpNoPorts, UdpInErrors, UdpOutDatagrams;

/* TCP MIB */
unsigned TcpRtoAlgorithm, TcpRtoMin, TcpRtoMax, TcpMaxConn,
TcpActiveOpens, TcpPassiveOpens, TcpAttemptFails, TcpEstabResets,
TcpCurrEstab, TcpInSegs, TcpOutSegs, TcpRetransSegs;

int snmpinitialized = FALSE;

/*
 * sninit - initialize the data structures for the SNMP server and client
 */
sninit()
{
    int i;

    if (snmpinitialized)
        return; /* if SNMP data structures already initialized */
    snmpinitialized = TRUE;
    hashinit();

    /* initialize most SNMP variables */
}
```



```
strcpy(SysDescr, vers);      strcpy(SysContect, CONTACT);
strcpy(SysLocation, LOCATION);  getname(SysName);  strcpy(SysDescr, vers);

IfNumber = Net.nif - 1;

/* non-zero int/count initializations */
IpDefaultTTL = IP_TTL;
IpReasmTimeout = IP_FTTL;
TcpRtoAlgorithm = 4;
TcpRtoMin = TCP_MINRXT*10;
TcpRtoMax = TCP_MAXRXT*10;
TcpMaxConn = Ntcp;
}
```

21.15 Summary

An SNMP server accepts incoming requests, performs the operation specified, and returns the result to the client. Although conceptually simple, the server code is dominated by conversions between the ASN.1 representation used in SNMP messages and the internal representation of values. The server parses an incoming message and converts it into a structure that uses fixed-format fields. It then uses the mib array to map from an object identifier and operation to a function that performs the specified operation. The server uses a separate data structure that specifies the functions for each table. By using indirection, the server can avoid having a separate function for each variable.

Client software shares most of the procedures used by the server. The client forms a message, sends it, and waits for a response. It must implement timeout, and if needed, retransmission. When a response arrives, the client parses the message and converts fields from the ASN.1 representation to an internal, fixed-field format. The client also compares the id field in a response to the id field in the request to insure they match.

21.16 FOR FURTHER STUDY

Case, Fedor, Schoffstall, and Davin [RFC 1157] describes the operation of a client and server,

21.17 EXERCISES

1. Procedure snparse encodes the SNMP message format in a sequence of statements that parse it. Can you devise a scheme that encodes the format in a



data structure instead? (Hint: think of a conventional compiler.)

2. The snmib data structure provides only one pointer for a function that implements get, set, and get-next operations. Compare this design to an alternative in which the structure contains a separate pointer for each operation similar to an entry in array tabtab. What are the advantages of each implementation?
3. Estimate the percentage of the code devoted to parsing and converting messages between ASN.1 and the internal representation. What percentage of the total SNMP code does it represent? Do the same for IP. What conclusions can you draw?
4. When the client specifies a get-next operation and gives the name of a data aggregate, snrslv follows the lexical pointer and invokes the function on the next item (using expression `*np->mi_next->mi_func`). Explain why snrslv does not need to check whether field `mi_func` is empty.
5. Design a user interface for the client that allows a manager to use SNMP without knowing what variables are available in the MIB.
6. How long should a client wait for a response to an SNMP request? How many times should a client retransmit?
7. The set of bindings in an SNMP get request specify pairs consisting of object identifiers and the value NULL. Why does the protocol include NULL values when they are unnecessary?
8. Read the SNMP protocol specification. Does it specify the range of values for individual variables like `IpForwarding`? If it does, give examples of variables and possible values. Does the server shown here check for values out of range in an SNMP set request? Explain.



22 SNMP: *Table Access Functions*

22.1 Introduction

The previous chapter examined an SNMP server, and showed how it used the function pointer in a mib entry to invoke an operation indirectly. When a message arrives, the server converts it to internal form, and stores it in the request descriptor structure. The request structure contains a pointer to a linked list of object names to which the specified operation must be applied. For simple MIB variables like integers, access is straightforward. Underlying access functions merely copy a value between an internal data structure used by the TCP/IP software and the binding list node. Once the server has performed the specified operation, it translates the request descriptor, along with items in the binding list, back into external form, and sends a reply.

This chapter concentrates on the underlying functions that handle get, set, and get-next requests for tables. It shows how the software maps from conceptual MIB tables into the data structures used by TCP/IP, and how underlying access functions implement operations on the tables. Finally, it examines additional data structures needed by SNMP that implement the entries in conceptual tables that do not correspond to existing data structures.

22.2 Table Access

Unlike simple variables that map to a location in memory, tables require additional software that maps the conceptual SNMP table into the corresponding internal data structure. For MIB tables, the server provides a mechanism that allows each table to have three functions that implement get, set, and get-next operations. As Chapter 19 shows, server software uses indirection to choose the correct access function by



following a pointer in the tabtab array^①; the array contains a separate pointer for each operation.

22.3 Object Identifiers For Tables

The five entries in array tabtab that correspond to tables do not contain full object identifiers. Instead, they only contain a prefix of the object identifier for the table. The reason is simple: the complete object identifier for an item in a table includes a prefix that identifies the table itself, as well as a suffix that identifies a particular entry in the table and a specific field within that entry. When checking an identifier, getmib^② first checks to see if it matches a simple variable by looking it up in the hash table. Hash table lookups use exact match, comparison. If no exact match can be found, getmib compares the identifier to the set of prefixes that correspond to MIB tables.

Once a prefix match has been found, the server invokes an underlying access function indirectly. The access function parses the suffix of the object identifier, and uses it to select an entry in the table as well as a field within that entry. For many tables, the MIB uses an IP address to select an entry. The IP address is encoded in the object identifier by including its dotted decimal representation. The next section provides an example.

22.4 Address Entry Table Functions

The MIB defines a conceptual address entry table that corresponds to the set of IP addresses for the interfaces on a machine. Each item in the table has four fields: an IP address (ipAdEntAddr), a network interface index (ipAdEntIfIndex), a subnet mask (ipAdEntMask), and a broadcast address (ipAdEntBcastAddr). Although the TCP/IP software does not have a data structure defined exactly this way, the network interface array, nif, contains the needed information.

To identify an item in the conceptual address entry table, client software creates an ASN.1 object identifier with a prefix that specifies the table, and a suffix that specifies an individual field within a specific table entry. For example, the object identifier

1. 3. 6. 1. 2. 1. 4. 20. 1. 1. 128. 10, 2. 3

specifies the standard MIB prefix (1.3.6.1.2.1), the ip subhierarchy (4), the ipAddrTable (20), an ipAddrEntry (1), a field within that entry (1), and an IP address used as the index for the entry (128.10.2.3). Thus, we can think of the object identifier as representing:

^① Page 452 contains the listing of file snmib.c in which tabtab is defined.

^② Page 459 contains the listing of file snhash.c in which getmib appears.



standard. MI B. prefix ip. ipAddrTable. ipAddrEntry. field. IP address

The access software for each table includes a matching function that tests whether a given object exists in the table. As we have seen, when a query arrives that specifies a variable in a table, the server uses a prefix of the object identifier to select the appropriate table, and then calls the table's matching function to decide whether the specified item exists. For example, function sae_match performs the test for the address entry table.

```
/* sae_match.c - sae_match */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>
#include <snhash.h>

/*
 *-----*
 * sae_match - check if a variable exists in the IP Address Entry Table
 *-----*
 */
int sae_match(bindl, iface, field, numifaces)
struct snbentry *bindl;
int      *iface;
int      *field;
int      numifaces;
{
    int oidi;

    oidi = SAE_OIDLEN; /* skip over fixed part of objid */

    if ((*field = bindl->sb_oid.id[oidi++]) > SNUMF_AETAB)
        return SYSERR;
    for (*iface = 1; *iface <= numifaces; (*iface)++)
        if (soipequ(&bindl->sb_oid.id[oidi],
                    nif[*iface].ni_ip, IP_ALEN))
            break;
    if (*iface > numifaces)
        return SYSERR;
    oidi += IP_ALEN;
    if (oidi != bindl->sb_oid.len) /* verify oidi at end of objid */
```



```
    return SYSERR;
    return OK;
}
```

The code is straightforward. Sae_match skips the part of the object identifier that identifies the address entry table and the address entry structure. It then extracts the integer that specifies which field of the table entry is desired, and stores the value at the location given by argument field. If the value specified is out of the valid range, sae_match returns an error code to indicate that the object identifier does not correspond to a valid table entry.

Once it has found a valid field specification, sae_match iterates through all network interfaces, comparing the remaining four values of the object identifier to the IP address of each interface. If it finds a match, sae_match stores the interface number in the address given by argument iface, and returns a code that specifies it found a match. If no match can be found, sae_match returns an error code that specifies the object identifier does not correspond to a valid entry.

22.4.1 Get Operation For The Address Entry Table

Procedure sae_get implements the get operation for an item in the address entry table.

```
/* sae_get.c - sae_get */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 *-----*
 * sae_get - perform a get on a variable in the IP Address Entry Table
 *-----*
 */
int sae_get(bindl, numifaces)
struct snbentry *bindl;
int      numifaces;
{
    int iface, field;

    if (sae_match(bindl, &iface, &field, numifaces) == SYSERR)
```



```
        return SERR_NO_SUCH;

    switch (field) {
        case 1: /* ipAdEntAddr */
            SVTYPE(bindl) = ASN1_IPADDR;
            blkcopy(SVIPADDR(bindl), nif[iface].ni_ip, IP_ALEN);
            return SNMP_OK;

        case 2: /* ipAdEntIfIndex */
            SVTYPE(bindl) = ASN1_INT;
            SVINT(bindl) = iface;
            return SNMP_OK;

        case 3: /* ipAdEntNetMask */
            SVTYPE(bindl) = ASN1_IPADDR;
            blkcopy(SVIPADDR(bindl), nif[iface].ni_mask, IP_ALEN);
            return SNMP_OK;

        case 4: /* ipAdEntBcastAddr */
            SVTYPE(bindl) = ASN1_INT;
            SVINT(bindl) = (nif[iface].ni_brc[IP_ALEN - 1] & 0x01);
            return SNMP_OK;

        case 5: /* ipAdEntReasmMaxSize */
            SVTYPE(bindl) = ASN1_INT;
            SVINT(bindl) = (nif[iface].ni_maxreasm);
            return SNMP_OK; default:
            break;
    }
    return SERR_NO_SUCH;
}
```

Sae_get uses sae_match to find the interface that matches the object identifier. If the identifier does not correspond to a valid entry, it returns an error. If it finds a match, sae_get proceeds to access the desired information.

The switch statement chooses one of the fields in the conceptual table entry, using the field code set by sae_match. The code that implements a given field stores both a type and a value in the binding list node, and returns to the caller.

22.4.2 Get-First Operation For The Address Entry Table

Function sae_getf implements the get-first operation for items in the address entry table.

```
/* sae_getf.c - sae_getf */
```

```
#include <conf.h>
```



```
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 *-----*
 *  sae_getf - perform a getfirst on a variable in the IPAddr Entry Table
 *-----*
 */
sae_getf(bindl, mip, numifaces)
struct snbentry      *bindl;
struct mib_info      *mip;
int      numifaces;
{
    int  iface, oidi;

    iface = sae_findnext(-1, numifaces);

    /* write the objid into the bindings list and call get func */
    blkcopy(bindl->sb_oid.id, mip->mi_objid.id, mip->mi_objid.len*2);
    oidi = mip->mi_objid.len;

    bindl->sb_oid.id[oidi++] = (u_short) 1;    /* field */
    sip2ocpy(&bindl->sb_oid.id[oidi], nif[iface].ni_ip);
    bindl->sb_oid.len = oidi + IP_ALEN;

    return sae_get(bindl, numifaces);
}
```

To find the lexically first entry in the table, sae_getf calls function sae_findnext, passing it -1 as a starting interface. Sae_findnext finds the first interface and returns its index. Once it knows the interface number, sae_getf computes the correct object identifier for the interface, and uses it to replace the identifier in the binding list node. To construct the identifier, sae_getf inserts 1 into the field value (to identify the first field), and calls procedure sip2ocpy to copy the IP address of the entry into the object identifier. Finally, sae_getf invokes sae_get to obtain the requested information.

22.4.3 Get-Next Operation For The Address Entry Table

Function sae_getn implements the get-next operation for items in the address entry



table.

```
/* sae_getn.c - sae_getn */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 *-----*
 * sae_getn - perform a getnext on a variable in the IPAddr Entry Table
 *-----*
 */

int sae_getn(bindl, mip, numifaces)
struct snbentry    *bindl;
struct mib_info    *mip;
int      numifaces;
{
    int field, iface, oidi;

    if (sae_match(bindl, &iface, &field, numifaces) == SYSERR)
        return SERR_NO_SUCH;
    if ((iface = sae_findnext(iface, numifaces)) == -1) {
        iface = sae_findnext(-1, numifaces);
        if (++field > SNUMF_AETAB)
            return (*mip->mi_next->mi_func)
                (bindl, mip->mi_next, SOP_GETF);
    }
    /* The fixed part of the objid is correct. Update the rest */
    oidi = SAE_OIDLEN;

    bindl->sb_oid.id[oidi++] = (u_short) field;
    sip2ocpy(&bindl->sb_oid.id[oidi], nif[iface].ni_ip);
    bindl->sb_oid.len = oidi + IP_ALEN;

    return sae_get(bindl, numifaces);
}
```

Implementation of the get-next operation requires three steps. First, sae_getn uses sae_match to find the entry in the table specified by the object identifier in the binding



list node. Second, it follows the lexical order defined for the table to locate the "next" item. Third, it applies the get operation,

To find the lexically next item in the table, sae_getn calls function sae_findnext, passing it as an argument the interface at which to start. If sae_findnext returns a valid interface, no further searching is required. However, if sae_findnext returns the value -1, it means that no more entries exist in the table beyond the one specified by the object id.

When sae_getn reaches the end of the table, it must increment the field value and move back to the lexically first item in the table. Ultimately, when it increments past the final field in the last table entry, sae_getn invokes the get-first function on the lexically next item in the MIB. For the case where incrementing the field results in a valid value, sae_getn must find the lexically first entry in the table. To find the lexically first item, it calls sae_findnext, passing it -1 as the starting interface.

22.4.4 Incremental Search In The Address Entry Table

Procedure sae_findnext searches the address entry table to find the item that follows a given item in the lexical order.

```
/* sae_findn.c - sae_findnext */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 * sae_findnext - find the next interface in the lexical ordering
 *-----*
 */
int sae_findnext(iface, numifaces)
int iface;
int numifaces;
{
    int i, nextif;

    for (nextif = -1, i = 1; i <= numifaces; ++i) {
        if (iface >= 0 &&
            blkcmp(nif[i].ni_ip,nif[iface].ni_ip,IP_ALEN) <= 0)
            continue;
        if (nextif < 0 ||
            blkcmp(nif[i].ni_ip,nif[nextif].ni_ip,IP_ALEN) < 0)
            nextif = i;
    }
}
```



```
    return nextif;
}
```

We think of argument iface as a starting interface. When iface contains -1, sae_findnext starts from the beginning, and locates the table item that has the lexicographically smallest object identifier (i.e., the "first" item in the table in lexicographic order). When argument iface contains a positive value, sae_findnext starts with that interface and locates the entry that follows it in the lexicographic order (i.e., the "next" item in the table after iface).

If sae_findnext locates the desired item in the table, it returns its index. Otherwise, it returns -1 to indicate that the starting interface is last in the lexicographic ordering.

22.4.5 Set Operation For The Address Entry Table

Because the address entry table does not permit managers to change entries, the implementation of set is trivial. Function sae_set merely returns an error if called.

```
/* sae_set.c - sae_set */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>

/*
 *-----*
 * sae_set - return error: the IP Address Entry Table is read-only
 *-----*
 */
sae_set()
{
    return SERR_NO_SUCH;
}
```

22.5 Address Translation Table Functions

The MIB defines a conceptual address translation table that corresponds to the ARP cache. Each entry in the table has three fields: the index of the network interface from which the entry was obtained (atIfIndex), the physical address in the entry (atPhyAddress), and the IP address in the entry (atNetAddress). The ASN.1 name for an item in the table encodes the IP address field of the entry.

The general form of an object identifier for an address translation table entry is



standard-MIB-prefix.at.atTable.atEntry.field.interface.1.IPAddress

A prefix identifies the table and address table structure, while the remaining octets specify the field in an entry, the interface for an entry, a type (1), and an IP address,

When an object identifier specifies an item in the address translation table, the SNMP server checks only a prefix of the identifier that identifies the table. It then calls table-specific functions that must parse the remainder of the object identifier and verify that it corresponds to a valid table entry. Function sat_match compares the object identifier suffix with entries in the table to see if any entry matches the specified name.

```
/* sat_match.c - sat_match */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>

/*
 * sat_match - check if a variable exists in the Addr Translation Table
 */
int sat_match(bindl, iface, entry, field, numifaces)
    struct snbentry      *bindl;
    int          *iface;
    int          *entry;
    int          *field;
    int          numifaces;
{
    int oidi;
    struct arpentry *pae;

    oidi = SAT_OIDLEN;
    if ((*field = bindl->sb_oid.id[oidi++]) > SNUMF_ATTAB)
        return SYSERR;
    if ((*iface = bindl->sb_oid.id[oidi++]) > numifaces)
        return SYSERR;
    oidi++;      /* skip over the 1 */
    /*
     * oidi now points to IPAddr. Read it and match it against
     * the correct arp cache entry to get entry number
     */
    for (*entry = 0; *entry < ARP_TSIZE; (*entry)++) {
```



```
    pae = &arphtable[*entry];
    if (pae->ae_state != AS_FREE &&
        pae->ae_pni == &nif[*iface] &&
        soipequ(&bindl->sb_oid.id[oidi],pae->ae_pra,IP_ALEN))
        break;
    }
    if (*entry >= ARP_TSIZE)
        return SYSERR;
    if (oidi + IP_ALEN != bindl->sb_oid.len)
        return SYSERR; /* oidi is not at end of objid */
    return OK;
}
```

After extracting the field and interface specifications, sat_match compares the four-octet IP address to the IP address stored in each entry in the ARP cache. It returns OK if a match is found, and SYSERR otherwise.

22.5.1 Get Operation For The Address Translation Table

Function sat_get implements the get operation for the address translation table. After calling sat_match to find the correct ARP entry, it uses a switch statement to select the requested field, and copies the information into the request descriptor.

```
/* sat_get.c - sat_get */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 * sat_get - do a get on a variable in the Address Translation Table
 */
int sat_get(bindl, numifaces)
struct snbentry *bindl;
int      numifaces;
{
    int iface, entry, field;
```



```
if (sat_match(bindl, &iface, &entry, &field, numifaces) == SYSERR)
    return SERR_NO_SUCH;
switch(field) {
case 1:      /* atIfIndex */
    SVTYPE(bindl) = ASN1_INT;
    SVINT(bindl) = iface;
    return SNMP_OK;
case 2:      /* atPhysAddress */
    SVTYPE(bindl) = ASN1_OCTSTR;
    SVSTR(bindl) = (char *) getmem(EP_ALEN);
    blkcopy(SVSTR(bindl), arptable[entry].ae_hwa,
            EP_ALEN);
    SVSTRLEN(bindl) = EP_ALEN;
    return SNMP_OK;
case 3:      /* atNetAddress */
    SVTYPE(bindl) = ASN1_IPADDR;
    blkcopy(SVIPADDR(bindl), arptable[entry].ae_pra,
            IP_ALEN);
    return SNMP_OK;
default:
    break;
}
return SERR_NO_SUCH;
}
```

22.5.2 Get-First Operation For The Address Translation Table

Function sat_getf implements the get-first operation for the address translation table.

```
/* sat_getf.c - sat_getf */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>
#include <snhash.h>

/*-----
 * sat_getf - do a getfirst on a variable in the Address Translation Table
 *-----
```



```
*/  
  
int sat_getf(bindl, mip, numifaces)  
struct snbentry      *bindl;  
struct mib_info      *mip;  
int      numifaces;  
{  
    int iface, entry, oidi;  
  
    for (iface=1; iface <= numifaces; ++iface)  
        if ((entry = sat_findnext(-1, iface)) >= 0)  
            break;  
    if (iface > numifaces) { /* no active interface found */  
        if (mip->mi_next)  
            return (*mip->mi_next->mi_func)  
                (bindl, mip->mi_next, SOP_GETF);  
        return SERR_NO_SUCH;           /* no next node */  
    }  
    blkcopy(bindl->sb_oid.id, mip->mi_objid.id, mip->mi_objid.len*2);  
    oidi = mip->mi_objid.len;  
  
    bindl->sb_oid.id[oidi++] = (u_short) 1;           /* field */  
    bindl->sb_oid.id[oidi++] = (u_short) iface;  
    bindl->sb_oid.id[oidi++] = (u_short) 1;  
    sip2ocpy(&bindl->sb_oid.id[oidi], arptable[entry].ae_pra);  
    bindl->sb_oid.len = oidi + IP_ALEN;  
  
    return sat_get(bindl, numifaces);  
}
```

The conceptual address translation table partitions entries into sets that correspond to individual interfaces, and places all entries from a given interface adjacent in the lexicographic ordering. Thus, to find the lexically first entry, sat_getf must iterate through each possible interface, one at a time. On each iteration, it calls sat_findnext to see if the cache contains any entries for the given interface. As soon as sat_findnext reports that it has found a valid entry, sat_getf stops the search. However, if the ARP cache is completely empty, the iteration will continue until all interfaces have been examined. In such cases, sat_getf applies the get-first operation to the lexically next item in the MIB (provided one exists).

If it finds a nonempty entry in The ARP cache, sat_getf constructs an object identifier that corresponds to the entry, and calls sat_get to extract the value. When



constructing the object identifier, it uses field 1 because the object identifier for the first field will be lexically least.

22.5.3 Get-Next Operation For The Address Translation Table

Function sat_getn provides the get-next operation for the address translation table.

```
/* sat_getn.c - sat_getn */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 * sat_getn - do a getnext on a variable in the Address Translation Table
 */
int sat_getn(bindl, mip, numifaces)
struct snbentry      *bindl;
struct mib_info      *mip;
int      numifaces;
{
    int entry, iface, field, oidi, i;

    if (sat_match(bindl, &iface, &entry, &field, numifaces) == SYSERR)
        return SERR_NO_SUCH;
    for (i=1; field <= SNUMF_ATTAB && i<=numifaces; ++i) {
        if ((entry = sat_findnext(entry, iface)) >= 0)
            break;
        if (++iface > numifaces) {
            iface = 1;
            ++field;
        }
    }
    if (entry < 0)
        return (*mip->mi_next->mi_func)
            (bindl, mip->mi_next, SOP_GETF);
    oidi = SAT_OIDLEN; /* 3.1.1 */

    bindl->sb_oid.id[oidi++] = (u_short) field;
```



```
bindl->sb_oid.id[oidi++] = (u_short) iface;
bindl->sb_oid.id[oidi++] = (u_short) 1;
sip2ocpy(&bindl->sb_oid.id[oidi], arptable[entry].ae_pra);
bindl->sb_oid.len = oidi + IP_ALEN;

return sat_get(bindl, numifaces);
}
```

Sat_getn uses sat_match to find the table entry that matches the specified object identifier. It then calls function sat_findnext to search the ARP cache for the next valid entry for that same interface. If no such entry exists in the ARP cache, it tries finding an entry for the next interface. If it exhausts all possible interfaces, sat_getn increments the field number, and begins searching the table again. The iteration terminates either because sat_findnext has found a valid entry that follows the starting entry, or because no such entry exists. If sat_getn has exhausted all possible interfaces without finding an entry, it applies the get-first operation to the object that follows the address translation table in the lexical ordering. Otherwise, sat_getn constructs an object identifier for the new entry, and calls sat_get to extract the value.

22.5.4 Incremental Search In The Address Entry Table

Function sat_findnext searches for an entry in the ARP cache according to the lexical order imposed by the MIB.

```
/* sat_findn.c - sat_findnext, satcmp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*-----
 * sat_findnext - for given iface, find next resolved arp entry
 *-----
 */
int sat_findnext(entry, iface)
int entry;
int iface;
{
    int i, next;

    next = -1;
    for (i = 0; i < ARP_TSIZE; ++i) {
```



```
struct arpentry    *pae = &arphtable[i];

if (pae->ae_state == AS_FREE ||
    pae->ae_pni != &nif[iface] ||
    (entry >= 0 && satcmp(pae, &arphtable[entry]) <= 0))
    continue;

if (next < 0 || satcmp(pae, &arphtable[next]) < 0)
    next = i;
}

return next;
}

/*-----
 * satcmp - compare two ARP table entries in SNMP lexicographic order
 *-----
 */

int satcmp(pael, pae2)
struct    arpentry *pael, *pae2;
{
    int  rv;

    if (rv = (pael->ae_prlen - pae2->ae_prlen))
        return rv;
    return blkcmp(pael->ae_pra, pae2->ae_pra, pael->ae_prlen);
}
```

Sat_findnext uses argument entry as an index that specifies a starting point, and argument iface to select entries for a single interface. It searches the ARP cache, looking only at valid entries that correspond to interface iface. If argument entry contains -1, sat_findnext remembers the first entry that matches iface. Once it has a candidate entry, sat_findnext only replaces that value if it finds another entry greater than the initial entry and less than the candidate.

22.5.5 Order From Chaos

The MIB address translation table illustrates an interesting idea: that it is possible to define structure and order for any system data structure. Although our implementation of ARP stores all entries for all network interfaces in a single cache, the MIB defines the conceptual table to be indexed by interface, as if ARP caches existed for each interface. The address table access functions make it appear that separate tables exist by imposing an order on how items can be searched. Thus, when accessing the address translation



table, an SNMP client remains completely unaware of the underlying implementation.

22.5.6 Set Operation For The Address Translation Table

Unlike the address entry table shown above, the address translation table allows managers to assign values to variables as well as fetch them. Function sat_set provides the set operation.

```
/* sat_set.c - sat_set */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 *-----*
 * sat_set - do a set on a variable in the Address Translation Table
 *-----*
 */
int sat_set(bindl, mip, numifaces)
struct snbentry    *bindl;
struct mib_info    *mip;
int      numifaces;
{
    int iface, entry, field;

    if (sat_match(bindl, &iface, &entry, &field, numifaces) == SYSERR)
        return SERR_NO_SUCH;
    switch (field) {
    case 1:      /* atIfIndex */
        if (SVTYPE(bindl) != ASN1_INT)
            return SERR_BAD_VALUE;
        if (SVINT(bindl) <= 0)
            return SERR_BAD_VALUE;
        if (SVINT(bindl) > numifaces)
            return SERR_BAD_VALUE;
        arptable[entry].ae_pni = &nif[SVINT(bindl)];
        return SNMP_OK;
    case 2:      /* atPhysAddress */
        if (SVTYPE(bindl) != ASN1_OCTSTR)
            return SERR_BAD_VALUE;
```



```
    if (SVSTRLEN(bindl) != EP_ALEN)
        return SERR_BAD_VALUE;

    blkcopy(arptable[entry].ae_hwa, SVSTR(bindl), EP_ALEN);
    return SNMP_OK;

case 3:      /* atNetAddress */
    if (SVTYPE(bindl) != ASN1_IPADDR)
        return SERR_BAD_VALUE;
    blkcopy(arptable[entry].ae_pra, SVIPADDR(bindl), IP_ALEN);
    return SNMP_OK;

default:
    break;
}

return SERR_NO_SUCH;
}
```

Sat_set checks the value type to verify that it matches the object type to which it must be assigned. In most cases, sat_set also checks to see that the value is in the legal range before making the assignment.

22.6 Network Interface Table Functions

The MIB defines a conceptual network interface table that holds information about each network interface. The server uses entries in array nif^① to store the information needed for the conceptual MIB table.

22.6.1 Interface Table ID Matching

ASN.1 object identifiers for items in the network interface table are constructed like identifiers for the tables examined previously. A prefix of the name specifies the table, while the suffix specifies a field within the table and the network interface. The general form is

```
standard-MIB-prefix.interfaces.interfaceTable.interfaceEntry.field.interface
```

Function sif_match decodes the suffix and verifies that the values are valid.

```
/* sif_match.c - sif_match */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

^① See page 28 for a listing of file netif.h that contains the declaration of nif.



```
#include <snmp.h>
#include <mib.h>

/*
 * sif_match - check if a variable exists in the Interfaces Table.
 */
int sif_match(bindl, iface, field, numifaces)
struct snbentry      *bindl;
int      *iface;
int      *field;
int      numifaces;
{
    int oidi;

    oidi = SIF_OIDLEN;
    if ((*field = bindl->sb_oid.id[oidi++]) > SNUMFIFTAB)
        return SYSERR;
    if ((*iface = bindl->sb_oid.id[oidi++]) > numifaces)
        return SYSERR;
    if (oidi != bindl->sb_oid.len)
        return SYSERR; /* oidi is not at end of objid */
    return OK;
}
```

22.6.2 Get Operation For The Network Interface Table

Because the MIB interface table defines 21 fields per entry, operations like get and set require more code than the tables examined earlier. For example, function sif_get provides the get operation.

```
/* sif_get.c - sif_get */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 * sif_get - perform a get on a variable in the Interfaces Table
 */

```



```
*/  
  
int sif_get(bindl, numifaces)  
struct snbentry *bindl;  
int numifaces;  
{  
    int iface, field, sl;  
  
    if (sif_match(bindl, &iface, &field, numifaces) == SYSERR)  
        return SERR_NO_SUCH;  
    switch (field) {  
    case 1: /* ifIndex */  
        SVTYPE(bindl) = ASN1_INT;  
        SVINT(bindl) = iface;  
        return SNMP_OK;  
    case 2: /* ifDescr */  
        SVTYPE(bindl) = ASN1_OCTSTR;  
        SVSTRLEN(bindl) = sl = strlen(nif[iface].ni_descr);  
        SVSTR(bindl) = (char *) getmem(sl);  
        blkcopy(SVSTR(bindl), nif[iface].ni_descr, sl);  
        return SNMP_OK;  
    case 3: /* ifType */  
        SVTYPE(bindl) = ASN1_INT;  
        SVINT(bindl) = nif[iface].ni_mtype;  
        return SNMP_OK;  
    case 4: /* ifMtu */  
        SVTYPE(bindl) = ASN1_INT;  
        SVINT(bindl) = nif[iface].ni_mtu;  
        return SNMP_OK;  
    case 5: /* ifSpeed */  
        SVTYPE(bindl) = ASN1_GAUGE;  
        SVINT(bindl) = nif[iface].ni_speed;  
        return SNMP_OK;  
    case 6: /* ifPhysAddress */  
        SVTYPE(bindl) = ASN1_OCTSTR;  
        SVSTR(bindl) = (char *) getmem(nif[iface].ni_hwa.ha_len);  
        blkcopy(SVSTR(bindl), nif[iface].ni_hwa.ha_addr,  
            SVSTRLEN(bindl) = nif[iface].ni_hwa.ha_len);  
        return SNMP_OK;  
    case 7: /* ifAdminStatus */  
        SVTYPE(bindl) = ASN1_INT;  
        SVINT(bindl) = nif[iface].ni_admstate;
```



```
        return SNMP_OK;

case 8:      /* ifOperStatus */
    SVTYPE(bindl) = ASN1_INT;
    SVINT(bindl) = nif[iface].ni_state;
    return SNMP_OK;

case 9:      /* ifLastChange */
    SVTYPE(bindl) = ASN1_TIMETICKS;
    SVINT(bindl) = nif[iface].ni_lastchange;
    return SNMP_OK;

case 10:     /* ifInOctets */
    SVTYPE(bindl) = ASN1_COUNTER;
    SVINT(bindl) = nif[iface].ni_ioctets;
    return SNMP_OK;

case 11:     /* ifInUcastPkts */
    SVTYPE(bindl) = ASN1_COUNTER;
    SVINT(bindl) = nif[iface].ni_iucast;
    return SNMP_OK;

case 12:     /* ifInNUcastPkts */
    SVTYPE(bindl) = ASN1_COUNTER;
    SVINT(bindl) = nif[iface].ni_inuicast;
    return SNMP_OK;

case 13:     /* ifInDiscards */
    SVTYPE(bindl) = ASN1_COUNTER;
    SVINT(bindl) = nif[iface].ni_idiscard;
    return SNMP_OK;

case 14:     /* ifInErrors */
    SVTYPE(bindl) = ASN1_COUNTER;
    SVINT(bindl) = nif[iface].ni_ierrors;
    return SNMP_OK;

case 15:     /* ifInUnknownProtos */
    SVTYPE(bindl) = ASN1_COUNTER;
    SVINT(bindl) = nif[iface].ni_iunkproto;
    return SNMP_OK;

case 16:     /* ifOutOctets */
    SVTYPE(bindl) = ASN1_COUNTER;
    SVINT(bindl) = nif[iface].ni_octets;
    return SNMP_OK;

case 17:     /* ifOutUcastPkts */
    SVTYPE(bindl) = ASN1_COUNTER;
    SVINT(bindl) = nif[iface].ni_oucast;
    return SNMP_OK;
```



```
case 18: /* ifOutNUcastPkts */
    SVTYPE(bindl) = ASN1_COUNTER;
    SVINT(bindl) = nif[iface].ni_onucast;
    return SNMP_OK;

case 19: /* ifOutDiscards */
    SVTYPE(bindl) = ASN1_COUNTER;
    SVINT(bindl) = nif[iface].ni_odiscard;
    return SNMP_OK;

case 20: /* ifOutErrors */
    SVTYPE(bindl) = ASN1_COUNTER;
    SVINT(bindl) = nif[iface].ni_oerrors;
    return SNMP_OK;

case 21: /* ifOutQLen */
    SVTYPE(bindl) = ASN1_GAUGE;
    SVINT(bindl) = lenq(nif[iface].ni_outq);
    return SNMP_OK;

#ifndef notyet
case 22: /* ifSpecific */
    SVTYPE(bindl) = ASN1_OBJID;
    SV
#endif

default:
    break;
}

return SERR_NO_SUCH;
}
```

As the code shows, the interface table has conceptual fields for the hardware type, the maximum transfer unit, the physical address, and for counters such as the number of input and output errors that have occurred. The code translates each conceptual field into a corresponding field of the array that the local software uses to store interface information.

22.6.3 Get-First Operation For The Network Interface Table

Function sif_getf defines the get-first operation for the network interface table.

```
/* sif_getf.c - sif_getf */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```



```
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 * sif_getf - perform a getfirst on a variable in the Interfaces Table
 */
int sif_getf(bindl, mip, numifaces)
struct snbentry    *bindl;
struct mib_info   *mip;
int      numifaces;
{
    int  oidi;

    blkcopy(bindl->sb_oid.id, mip->mi_objid.id, mip->mi_objid.len*2);
    oidi = mip->mi_objid.len;

    bindl->sb_oid.id[oidi++] = (u_short) 1;           /* field */
    bindl->sb_oid.id[oidi++] = (u_short) 1;           /* interface */
    bindl->sb_oid.len = oidi;

    return sif_get(bindl, numifaces);
}
```

Unlike tables discussed earlier, the network interface table does not use data values as an index. Instead, we think of the interface table as a one-dimensional array indexed by an integer between 1 and the maximum number of interfaces. For such a table, the implementation of the get-first operation is extremely simple. Sif_getf constructs an object identifier that specifies field 1 of interface 1 in the network interface table. It then calls the get operation to extract the value.

22.6.4 Get-Next Operation For The Network Interface Table

Function sif_getn provides the get-next operation for the network interface table.

```
/* sif_getn.c - sif_getn */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
```



```
#include <mib.h>

/*
 * sif_getn - perform a getnext on a variable in the Interfaces Table.
 */
sif_getn(bindl, mip, numifaces)
struct snbentry      *bindl;
struct mib_info      *mip;
int      numifaces;
{
    int oidi, field, iface;

    if (sif_match(bindl, &iface, &field, numifaces) == SYSERR)
        return SERR_NO_SUCH;
    if (++iface > numifaces) {
        iface = 2;
        if (++field > SNUMFIFTAB)
            return (*mip->mi_next->mi_func)
                (bindl, mip->mi_next, SOP_GETF);
    }
    oidi = SIF_OIDLEN; /* 2.2.1 */

    bindl->sb_oid.id[oidi++] = (u_short) field;
    bindl->sb_oid.id[oidi++] = (u_short) iface;
    bindl->sb_oid.len = oidi;

    return sif_get(bindl, numifaces);
}
```

Because object identifiers for table entries can be constructed without knowing the table contents, sif_getn does not need to search. It first increments the interface number. If the value exceeds the maximum number of interfaces, sif_getn sets the interface number to 1 and increments the field number. If the field number exceeds the maximum number of fields, no additional entries exist, so sif_getn applies the get-first operation to the next variable in the MIB lexical ordering.

As with other tables, the get-next operation requires an application of the get operation after the next object has been found. Thus, if sif_getn finds a valid successor, it constructs an object identifier for the new item, and then calls sif_get to extract the value.



22.6.5 Set Operation For The Network Interface Table

The network interface table allows managers to set the administrative status field; other fields only allow read access. As a result, the implementation returns an error code for all other fields. Function sif_set contains the code,

```
/* sif_set.c - sif_set */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 *-----*
 * sif_set - perform a set on a variable in the Interfaces Table
 *-----*
 */
int sif_set(bindl, mip, numifaces)
struct snbentry    *bindl;
struct mib_info    *mip;
int      numifaces;
{
    int iface, field;

    if (sif_match(bindl, &iface, &field, numifaces) == SYSERR)
        return SERR_NO_SUCH;
    /* the only settable object here is ifAdminStatus (ifEntry 7) */
    if (field != 7)
        return SERR_NO_SUCH;
    if (SVTYPE(bindl) != ASN1_INT)
        return SERR_BAD_VALUE;
    if (SVINT(bindl) < 0 || SVINT(bindl) > 1)
        return SERR_BAD_VALUE;
    /* value is OK, so install it */
    nif[iface].ni_admstate = SVINT(bindl);
    return SNMP_OK;
}
```

After verifying that the request specifies field ifAdminStatus, sif_set checks the value to insure it is a positive integer, and checks the interface specification to insure it



specifies a valid interface. If the value is valid, sif_set makes the assignment.

22.7 Routing Table Functions

The MIB defines a conceptual table that corresponds to a gateway's IP routing table. Like object identifiers for items in the address entry table, an object id for the routing table encodes both a field designator and an IP address that SNMP uses as an index into the table. The object identifiers have the following general form:

```
standard-MIB-prefix.ip.ipRoutingTable.ipRouteEntry.field.IPdestaddr
```

The IPdestaddr portion of the identifier gives a 4-octet IP address used to identify the route.

Matching function srt_match extracts the field specification, and matches the suffix of the object id against the IP address of a routing table entry.

```
/* srt_match.c - srt_match */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>

/*-----
 * srt_match - check if a variable exists in the current Routing Table
 *-----
 */
int srt_match(bindl, rtp, rtl, field)
struct snbentry    *bindl;
struct route   **rtp;
int        *rtl;
int        *field;
{
    int oidi, i;
    Bool found;

    oidi = SRT_OIDLEN;
    if ((*field = bindl->sb_oid.id[oidi++]) > SNUMF_RTTAB)
        return SYSERR;
    /* oidi points to IP address to match in the routing table. */
    for (found = FALSE, i = 0; !found && i < RT_TSIZ; i++)
        for (*rtp = rttable[i]; *rtp; *rtp = (*rtp)->rt_next)
```



```
    if (found = soipequ(&bindl->sb_oid.id[oidi],  
                        (*rtp)->rt_net, IP_ALEN))  
        break;  
  
    if (!found || /* not there */  
        oidi + IP_ALEN != bindl->sb_oid.len) /* not end of object id*/  
        return SYSERR;  
  
    *rtl = i - 1;  
    return OK;  
}
```

Srt_match assigns argument field the field identifier from the object id, and searches the IP routing table for an entry that matches the IP address given in the object id. The search iterates through all locations of the routing table, and follows the linked list of routes that extends from each. Srt_match returns OK if it finds an exact match, and returns SYSERR otherwise.

22.7.1 Get Operation For The Routing Table

The MIB routing table variable contains ten fields. The obvious fields correspond to the destination IP address (ipRouteDest), the address of the next-hop for that destination (ipRoureNextHop), and the index of the interface over which traffic will be sent to the next hop (ipRouteIfIndex). The table also contains fields that specify the protocol that installed the route (ipRouteProto), the type of route (ipRouteType), and the time-to-live value for the route (ipRouteAge). Function srt_get implements the get operation for the routing table. It calls srt_match to find a route that matches the specified object identifier, and then uses the specified field to select the correct piece of code to satisfy the request.

```
/* srt_get.c - srt_get */  
  
#include <conf.h>  
#include <kernel.h>  
#include <network.h>  
#include <snmp.h>  
#include <mib.h>  
#include <asn1.h>  
  
/*-----  
 * srt_get - perform a get on a variable in the Routing Table  
 *-----  
 */
```



```
int srt_get(bindl, numifaces)
struct snbentry    *bindl;
int      numifaces;
{
    struct    route *rtp;
    int   rtl, field;

    if (srt_match(bindl, &rtp, &rtl, &field) == SYSERR)
        return SERR_NO_SUCH;
    switch(field) {
    case 1:      /* ipRouteDest */
        SVTYPE(bindl) = ASN1_IPADDR;
        blkcopy(SVIPADDR(bindl), rtp->rt_net, IP_ALEN);
        return SNMP_OK;
    case 2:      /* ipRouteIfIndex */
        SVTYPE(bindl) = ASN1_INT;
        SVINT(bindl) = rtp->rt_ifnum;
        return SNMP_OK;
    case 3:      /* ipRouteMetric1 */
        SVTYPE(bindl) = ASN1_INT;
        SVINT(bindl) = rtp->rt_metric;
        return SNMP_OK;
    case 4:      /* ipRouteMetric2 */
    case 5:      /* ipRouteMetric3 */
    case 6:      /* ipRouteMetric4 */
        SVTYPE(bindl) = ASN1_INT;
        SVINT(bindl) = -1;
        return SNMP_OK;
    case 7:      /* ipRouteNextHop */
        SVTYPE(bindl) = ASN1_IPADDR;
        blkcopy(SVIPADDR(bindl), rtp->rt_gw, IP_ALEN);
        return SNMP_OK;
    case 8:      /* ipRouteType */
        SVTYPE(bindl) = ASN1_INT;
        if (rtp->rt_metric)
            SVINT(bindl) = 4; /* remote */
        else
            SVINT(bindl) = 3; /* direct */
        return SNMP_OK;
    case 9:      /* ipRouteProto */
        SVTYPE(bindl) = ASN1_INT;
```



```
    SVINT(bindl) = 1;           /* other */
    return SNMP_OK;

case 10:      /* ipRouteAge */
    SVTYPE(bindl) = ASN1_INT;
    SVINT(bindl) = rtp->rt_ttl;
    return SNMP_OK;

case 11:      /* ipRouteMask */
    SVTYPE(bindl) = ASN1_IPADDR;
    blkcopy(SVIPADDR(bindl), rtp->rt_mask, IP_ALEN);
    return SNMP_OK;

case 12:      /* ipRouteMetric5 */
    SVTYPE(bindl) = ASN1_INT;
    SVINT(bindl) = -1;
    return SNMP_OK;

default:
    break;
}
return SERR_NO_SUCH;
}
```

Srt_get handles most requests as expected, by accessing the appropriate field of the IP routing structure. Because the local routing table does not include multiple metrics, srt_get returns -1 for requests that correspond to routing metrics 2 through 4. Srt_get always returns the code for other (1) when a client requests field ipRouteProto. Although some routes may have been installed by the local system, by RIP, or by ICMP, srt_get has no way of knowing because the local routing table does not distinguish among them. This is an example where the conceptual MIB table includes variables that not only do not exist, but also cannot be computed without making significant changes in the underlying system.

22.7.2 Get-First Operation For The Routing Table

Function srt_getf implements the get-first operation for the routing table.

```
/* srt_getf.c - srt_getf */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>
```



```
/*
 *-----*
 * srt_getf - perform a getfirst on a variable in the Routing Table
 *-----*
 */

int srt_getf(bindl, mip, numifaces)
struct snbentry    *bindl;
struct mib_info   *mip;
int      numifaces;
{
    int      rtl, oidi;
    struct route  *rtp, *srt_findnext();

    rtl = -1; /* use first field, first route */
    if ((rtp = srt_findnext(rtp, &rtl)) == 0) {
        if (mip->mi_next)
            return (*mip->mi_next->mi_func)
                (bindl, mip->mi_next, SOP_GETF);
        return SERR_NO_SUCH; /* no next node */
    }
    blkcopy(bindl->sb_oid.id, mip->mi_objid.id, mip->mi_objid.len*2);
    oidi = mip->mi_objid.len;

    bindl->sb_oid.id[oidi++] = (u_short) 1;           /* field */
    sip2ocpy(&bindl->sb_oid.id[oidi], rtp->rt_net);
    bindl->sb_oid.len = oidi + IP_ALEN;

    return srt_get(bindl, numifaces);
}
```

When searching for a route, `srt_getf` must be sure to select one that has the lexically smallest object identifier. It calls function `srt_findnext` to scan the table and extract such a route. If the table is empty, `srt_findnext` returns -1, and `srt_getf` invokes the get-first operation on the next item in the MIB lexical order. If it finds an item, `srt_getf` creates the correct object id, and invokes the get operation on that item.

22.7.3 Get-Next Operation For The Routing Table

The get-next operation for the routing table differs from the get-next operation for previous tables in one significant way: the IP routing table may contain multiple routes for a given destination. The reason is simple: although the MIB defines a destination



address as the key for table lookup, the routing table can contain multiple routes for a single key. In particular, the routing table can contain a host-specific route, a subnet-specific route, and a network-specific route for each destination. Consider what happens if the table contains all three for some destination D. A get-next for D finds the host-specific route, and then uses the "next" route in the table. Unfortunately, the next route happens to be the subnet-specific route, which has the same destination IP address. If get-next constructs an object identifier for the response, it will be identical to the object identifier used in the request. Thus, a subsequent get-next will match the first route entry again, and the client will be usable to move through the table.

To handle the problem of ambiguous destination addresses, our implementation of get-next ignores routing table entries that have a destination address equal to that in the object identifier supplied. We can summarize:

Using only the destination IP address as a key for the MIB routing table prevents complete table access because an IP routing table may have multiple routes for a given destination address. When performing the get-next operation, our implementation skips multiple routing table entries that have the same destination address as the request.

Function srt_getn implements the get-next operation for the routing table. It calls function srt_findnext to handle the problem of address ambiguity.

```
/* srt_getn.c - srt_getn */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 *-----*
 * srt_getn - perform a getnext on a variable in the Routing Table
 *-----*
 */
int srt_getn(bindl, mip, numifaces)
struct snbentry *bindl;
struct mib_info *mip;
int      numifaces;
{
    struct    route    *rtp, *srt_findnext();
    int      rtl, field, oidi;
```



```
if (srt_match(bindl, &rtp, &rtl, &field) == SYSERR)
    return SERR_NO_SUCH;
if ((rtp = srt_findnext(rtp, &rtl)) == 0) {
    rtp = (struct route *) NULL;
    rtl = 0; /* set route hash table list to 0 */
    rtp = srt_findnext(rtp, &rtl);
    if (++field > SNUMF_RTTAB)
        return (*mip->mi_next->mi_func)
            (bindl, mip->mi_next, SOP_GETF);
}
oidi = SRT_OIDLEN; /* 4.21.1 */

bindl->sb_oid.id[oidi++] = field;
sip2ocpy(&bindl->sb_oid.id[oidi], rtp->rt_net, IP_ALEN);
bindl->sb_oid.len = oidi + IP_ALEN;

return srt_get(bindl, numifaces);
}
```

22.7.4 Incremental Search In The Routing Table

Srt_findnext searches the routing table for the "next" entry in lexicographic order.

```
/* srt_findn.c - srt_findnext */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 *-----*
 * srt_findnext - find next route in the lexicographic ordering
 *-----*
 */
struct route *srt_findnext(rtp, rtl)
struct route *rtp;
int      *rtl;
{
    struct route *nextrtp, *trtp;
    int      i, nextrtl;

    for (i = 0, nextrtl = -1; i < RT_TSIZE; i++)
        for (trtp = rttable[i]; trtp; trtp = trtp->rt_next) {
```



```
    if (*rtl >= 0 &&
        blkcmp(trtp->rt_net,rtp->rt_net,IP_ALEN)<=0)
        continue;

    if (nextrtl < 0 || blkcmp(trtp->rt_net,
                               nextrtp->rt_net, IP_ALEN) < 0) {
        nextrtp = trtp;
        nextrtl = i;
    }
}

if (nextrtl == -1) /* no next route found */
    return 0;
*rtl = nextrtl;
return nextrtp;
}
```

When called with argument rtp equal to -1, srt_findnext locates the lexically least entry; when called with a specific route, it finds the following route by skipping multiple routes that have the same destination address.

22.7.5 Set Operation For The Routing Table

Function srt_set implements the set operation for the routing table.

```
/* srt_set.c - srt_set */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 * srt_set - perform a set operation on a variable in the Routing Table
 */
int srt_set(bindl, mip, numifaces)
struct snbentry    *bindl;
struct mib_info    *mip;
int      numifaces;
{
    struct    route    *rtp;
```



```
int      rtl, field;

if (srt_match(bindl, &rtp, &rtl, &field) == SYSERR)
    return SERR_NO_SUCH;
switch (field) {
case 1:      /* ipRouteDest */
    if (SVTYPE(bindl) != ASN1_IPADDR)
        return SERR_BAD_VALUE;
    blkcopy(rtp->rt_net, SVIPADDR(bindl), IP_ALEN);
    return SNMP_OK;
case 2:      /* ipRouteIfIndex */
    if (SVTYPE(bindl) != ASN1_INT || SVINT(bindl) <= 0 || SVINT(bindl) > numifaces)
        return SERR_BAD_VALUE;
    rtp->rt_ifnum = SVINT(bindl);
    break;
case 3:      /* ipRouteMetric1 */
    if (SVTYPE(bindl) != ASN1_INT || SVINT(bindl) < 0)
        return SERR_BAD_VALUE;
    rtp->rt_metric = SVINT(bindl);
    break;
case 4:      /* ipRouteMetric2 */
case 5:      /* ipRouteMetric3 */
case 6:      /* ipRouteMetric4 */
    break;
case 7:      /* ipRouteNextHop */
    if (SVTYPE(bindl) != ASN1_IPADDR)
        return SERR_BAD_VALUE;
    blkcopy(rtp->rt_gw, SVIPADDR(bindl), IP_ALEN);
    break;
case 8:      /* ipRouteType */
/* route type is invalid (2) ==> should remove route
   from routing table */
    if (SVTYPE(bindl) != ASN1_INT || SVINT(bindl) < 1 || SVINT(bindl) > 4)
        return SERR_BAD_VALUE;
    if (SVINT(bindl) == 2) /* route invalid */
        (void) rtdel(rtp->rt_net, rtp->rt_mask);
    break;
case 9:      /* ipRouteProto */
    break;
```



```
case 10: /* ipRouteAge */
    if (SVTYPE(bindl) != ASN1_INT || SVINT(bindl) < 0)
        return SERR_BAD_VALUE;
    rtp->rt_ttl = SVINT(bindl);
    break;
default:
    return SERR_NO_SUCH;
}
return SNMP_OK;
}
```

For most fields of the routing table, srt_set translates assignment requests into appropriate assignments to fields in the routing table. Assignments to fields that the local software does not provide (e.g., multiple routing metrics) have no effect. Srt_set checks most values to insure that they are valid before assigning them.

Srt_set contains one interesting special case. The protocol standard specifies that assigning invalid to field ipRouteType means that the route should be removed from the table. If the request assigns invalid (2) to the ipRouteType field, srt_set calls rtDel to remove the route.

22.8 TCP Connection Table Functions

The MIB defines a table that contains all active TCP connections. Object identifiers for the TCP connection table have the following general form:

standard-MIB-prefix.x.tcp.TcpConnTable.tcpConnEntry.remainder

where remainder consists of five fields:

field, localIP, localport, remoteIP, remoteport

As expected, the object identifier contains the standard MIB prefix, specifies that the item lies in the TCP subhierarchy, and then specifies the connection table as well as the connection table entry. The localIP, localport, remoteIP, and remoteport portions of the identifier give the IP addresses and protocol port numbers of the connection endpoints, while the field selects among the fields in a table entry. As with other tables, a matching function verifies that an identifier correctly matches the connection table, and extracts the value of the field selector. Procedure stc_match performs the matching.

```
/* stc_match.c - stc_match */

#include <conf.h>
#include <kernel.h>
```



```
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*-----
 * stc_match - check if a variable exists in the TCP connections table
 *-----
 */
int stc_match(bindl, field, tcbn)
struct snbentry      *bindl;
int      *field, *tcbn;
{
    int oidi = STC_OIDLEN;
    IPAddr   lip, rip;
    int lport, rport;

    if ((*field = bindl->sb_oid.id[oidi++]) > SNUMF_TCTAB)
        return SYSERR;
    so2ipcpy(lip, &bindl->sb_oid.id[oidi]);
    oidi += IP_ALEN;
    lport = bindl->sb_oid.id[oidi++];
    so2ipcpy(rip, &bindl->sb_oid.id[oidi]);
    oidi += IP_ALEN;
    rport = bindl->sb_oid.id[oidi++];

    for (*tcbn = 0; *tcbn < Ntcp; ++(*tcbn)) {
        if (tcbtab[*tcbn].tcb_state == TCPS_FREE)
            continue;
        if (lport == tcbtab[*tcbn].tcb_lport &&
            rport == tcbtab[*tcbn].tcb_rport &&
            blkequ(rip, tcbtab[*tcbn].tcb_rip, IP_ALEN) &&
            blkequ(lip, tcbtab[*tcbn].tcb_lip, IP_ALEN))
            break;
    }
    if (*tcbn >= Ntcp || oidi != bindl->sb_oid.len)
        return SYSERR;
    return OK;
}
```

After extracting the field selector, stc_match searches the TCB table to find a valid



entry that matches the connection endpoints specified in the object identifier. It compares both the local and remote address and port pairs because TCP uses both endpoints to identify a connection. If stc_match finds a connection that matches the object identifier, it assigns argument tcbn the index in array tcbtab at which information about the connection can be found, and returns OK. Otherwise, it returns the error code SYSERR.

22.8.1 Get Operation For The TCP Connection Table

Each entry in the conceptual MIB connection table has five fields that correspond to the connection state, the local IP address, the local TCP port number, the remote IP address, and the remote TCP port number. Procedure stc_get implements the get operation for the connection table by using the field selector to choose among the five items.

```
/* stc_get.c - stc_get */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

-----
 * stc_get - perform a get on a variable in the TCP connections table
 *
 */
int stc_get(bindl)
struct snbentry      *bindl;
{
    int field, tcbn;

    if (stc_match(bindl, &field, &tcbn) == SYSERR)
        return SERR_NO_SUCH;
    switch (field) {
        case 1:      /* tcpConnState */
            SVTYPE(bindl) = ASN1_INT;
            SVINT(bindl) = tcbtab[tcbn].tcb_state;
            break;
        case 2:      /* tcpConnLocalAddress */
            SVTYPE(bindl) = ASN1_IPADDR;
            blkcopy(SVIPADDR(bindl), tcbtab[tcbn].tcb_lip, IP_ALEN);
    }
}
```



```
        break;

    case 3:      /* tcpConnLocalPort */
        SVTYPE(bindl) = ASN1_INT;
        SVINT(bindl) = tcbtab[tcbn].tcb_lport;
        break;

    case 4:      /* tcpConnRemAddress */
        SVTYPE(bindl) = ASN1_IPADDR;
        blkcopy(SVIPADDR(bindl), tcbtab[tcbn].tcb_rip, IP_ALEN);
        break;

    case 5:      /* tcpConnRemPort */
        SVTYPE(bindl) = ASN1_INT;
        SVINT(bindl) = tcbtab[tcbn].tcb_rport;
        break;

    default:
        return SERR_NO_SUCH;
    }

    return SNMP_OK;
}
```

22.8.2 Get-First Operation For The TCP Connection Table

Procedure stc_getf provides the get-first operation for the TCP connection table. Stc_getf calls stc_findnext to search array tcbtab until it finds the allocated TCB that has the lexically least identifier. If no TCB has been allocated, the MIB connection table is defined to be empty, so stc_getf applies the get-first operation to the next variable in the lexical order, and returns the result. If stc_findnext finds, a valid connection, stc_getf constructs an object identifier for the connection, applies the get operation to it, and returns the result.

```
/* stc_getf.c - stc_getf */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 * stc_getf - do a getfirst on a variable in the TCP connection table
 */
int stc_getf(bindl, mip, numifaces)
```



```
struct snbentry *bindl;
struct mib_info *mip;
int      numifaces;
{
    int oidi, tcbn;

    /* find first connection, if any */
    tcbn = stc_findnext(-1);
    if (tcbn < 0) {
        if (mip->mi_next)
            return((*mip->mi_next->mi_func)
                   (bindl, mip->mi_next, SOP_GETF));
        return SERR_NO_SUCH;
    }
    blkcopy(bindl->sb_oid.id, mip->mi_objid.id, mip->mi_objid.len*2);
    oidi = mip->mi_objid.len;

    bindl->sb_oid.id[oidi++] = (u_short) 1; /* field */
    sip2ocpy(&bindl->sb_oid.id[oidi], tcbtab[tcbn].tcb_lip);
    oidi += IP_ALEN;
    bindl->sb_oid.id[oidi++] = (u_short) tcbtab[tcbn].tcb_lport;
    sip2ocpy(&bindl->sb_oid.id[oidi], tcbtab[tcbn].tcb_rip);
    oidi += IP_ALEN;
    bindl->sb_oid.id[oidi++] = (u_short) tcbtab[tcbn].tcb_rport;
    bindl->sb_oid.len = oidi;

    return stc_get(bindl);
}
```

22.8.3 Get-Next Operation For The TCP Connection Table

Procedure stc_getn implements the get-next operation for the connection table.

```
/* stc_getn.c - stc_getn */
```

```
#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*-----
```



```
* stc_getn - do a getnext on a variable in the TCP connection table
*-----
*/
stc_getn(bindl, mip)
struct snbentry      *bindl;
struct mib_info      *mip;
{
    int oidi, field, tcbn, ttcbn;

    if (stc_match(bindl,&field,&tcbn) == SYSERR)
        return SERR_NO_SUCH;
    /* search for next connection */
    if ((tcbn = stc_findnext(tcbn)) < 0) {
        tcbn = stc_findnext(-1);
        if (++field > SNUMF_TCTAB)
            return((*mip->mi_next->mi_func)
                   (bindl, mip->mi_next, SOP_GETF));
    }
    oidi = STC_OIDLEN;      /* 6.13.1 */

    bindl->sb_oid.id[oidi++] = (u_short) field;
    sip2ocpy(&bindl->sb_oid.id[oidi], tcbtab[tcbn].tcb_lip);
    oidi += IP_ALEN;
    bindl->sb_oid.id[oidi++] = (u_short) tcbtab[tcbn].tcb_lport;
    sip2ocpy(&bindl->sb_oid.id[oidi], tcbtab[tcbn].tcb_rip);
    oidi += IP_ALEN;
    bindl->sb_oid.id[oidi++] = (u_short) tcbtab[tcbn].tcb_rport;
    bindl->sb_oid.len = oidi;

    return stc_get(bindl);
}
```

Stc_getn uses stc_match to find the TCB that matches the specified identifier. It then calls procedure stc_findnext to search for the next valid connection. Once stc_findnext finds a valid entry in tcbtab, it returns the index. If stc_findnext finds a valid connection, stc_getn constructs an object identifier for the entry, applies the get operation, and returns the result.

If no more valid entries exist, stc_getn increments the field specification, moves back to the start of array tcbtab, and continues the search. Finally, after exhausting all fields of all valid connections, stc_getn applies the get-first operation to the lexically



next MIB variable, and returns the result.

22.8.4 Incremental Search In The TCP Connection Table

Procedure stc_findnext searches the table of TCBs for the TCB that lexically follows the one specified by argument tcbn.

```
/* stc_findn.c - stc_findnext, sntpcmp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * stc_findnext - search the TCP connection table for the next valid entry
 */
int stc_findnext(tcbn)
int tcbn;
{
    int i, next;

    for (i = 0, next = -1; i < Ntcp; ++i) {
        if (tcbtab[i].tcb_state == TCPS_FREE ||
            (tcbn >= 0 && sntpcmp(i, tcbn) <= 0))
            continue;
        if (next < 0 || sntpcmp(i, next) < 0)
            next = i;
    }
    return next;
}

/*
 * sntpcmp - compare two TCP connections in SNMP lexical ordering
 */
int sntpcmp(tcb1, tcb2)
int tcb1, tcb2;
{
    int rv;

    if (rv=blkcmp(tcbtab[tcb1].tcb_lip,tcbtab[tcb2].tcb_lip,IP_ALEN))
        return rv;
```



```
    if (rv = (tcbtab[tcb1].tcp_lport - tcbtab[tcb2].tcp_lport))
        return rv;
    if (rv=blkcmp(tcbtab[tcb1].tcp_rip,tcbtab[tcb2].tcp_rip,IP_ALEN))
        return rv;
    if (rv = (tcbtab[tcb1].tcp_rport - tcbtab[tcb2].tcp_rport))
        return rv;
    return 0;
}
```

To find the lexical order imposed by the MIB for a pair of connections, the software must evaluate all four components of the connection endpoints. The local IP address is the most significant field, the local protocol port and the remote IP address are the next significant fields, and the remote protocol port number is the least significant field. Function sntcpcmp compares two endpoints according to the lexicographic order. It returns zero if they are equal, a value less than zero if the first is lexically less than the second, and a value greater than zero if the first is lexically greater than the second.

22.8.5 Set Operation For The TCP Connection Table

The MIB defines all values in the TCP connection table to be read-only, so a server must return an error in response to a set request. Procedure stc_set returns the appropriate error value.

```
/* stc_set.c - stc_set */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*
 * stc_set - return error: the TCP Connections Table is read-only
 */
stc_set(bindl, mip)
struct snbentry      *bindl;
struct mib_info      *mip;
{
    return SERR_NO_SUCH;
```



}

22.9 Summary

Table access functions differ from access functions for simple variables because the server must interpret part of the object identifier as an index to a specific table entry. Get and set operations require the client to supply the full name of the table entry. The get-next operation allows the client to walk tables without knowing the exact names for all items.

We reviewed the implementation of the MIB address entry table, network interface table, address translation table, IP routing table, and TCP connection table. Each table requires routines that provide get, get-next, and set operations. In addition, our implementation includes a match function for each table that matches object identifiers against available table entries, a findnext function that finds the next entry in the lexicographic order, and a get-first function that handles a get-next request for empty tables.

22.10 FOR FURTHER STUDY

More details on the names of MIB variables can be found in McCloghrie and Rose [RFC 1156] and [RFC 1155]. McCloghrie and Kastenholz [RFC 1573] discusses the interfaces group in MIB-II. McCloghrie and Rose [RFC 12131] specifies the names of variables used here. Many RFCs have proposed MIB variables for specific hardware interface devices; consult the RFC index for current examples.

22.11 EXERCISES

1. Should a manager be allowed to set the hardware address in an address translation table entry? Why or why not?
2. What is the exact form of an ASN.1 object identifier for the ipRouteNextHop field in an ipRoutingTable entry for IP address 128.10.2.3?
3. Can SNMP access all fields in all IP routing table entries? Why or why not?
4. What will the server return if a client issues an SNMP get-next request with an object identifier that specifies the last entry in the address translation table?
5. Can SNMP be used to create a new entry in any of the tables shown here? Explain.
6. The implementation shown does not provide a table of EGP information. Read the MIB specification, and make a list of all fields required for the EGP table.



23 Implementation In Retrospect

23.1 Introduction

This brief chapter takes a retrospective look at the implementation presented in previous chapters. It analyzes the code, points to strengths and weaknesses in the protocols from an implementation perspective, and draws conclusions about the design of protocol software.

23.2 Statistical Analysis Of The Code

Analyzing a system as large as the one covered in this text is difficult because no single measure provides an accurate assessment. Furthermore, it is impossible to measure "difficulty" or "effort" because the person-months of effort depend on the skill and background of the programmers involved. Thus, we have chosen to avoid such evaluations, and look instead at objective measures of the resulting code.

Two measurements of the software verify our intuitive assessments, and provide sufficient data to support a few conclusions. The first measurement counts the number of functions or procedures used to implement a given protocol; the second counts the number of lines of code. While we understand that the division of software into procedures depends on the programmer, and the number of lines of code depends on the coding style, these measures say much about the relative complexity of implementing the major protocols.

23.3 Lines Of Code For Each Protocol

The first measure to evaluate the software considers the lines of code required to implement each protocol. The code evaluated includes all pieces of the software discussed in the text, as well as several machine-dependent support procedures not



shown. It has been divided into eleven groups^①: TCP, SNMP, OSPF, IP (including all routines that handle routing, fragmentation and reassembly, and broadcast), NET (the network device driver and network interface routines), OTHER, ARP, IGMP, RIP, ICMP, and UDP. The OTHER group, includes utilities, such as the checksum procedures and initialization routines. The evaluation did not include other operating system functions or the code for other protocols (e.g., the code for rwho was omitted from consideration).

The code considered includes approximately 15,000 lines. Figure 23.1 shows the lines of code in each group as a percentage of the total code considered. As the figure shows, TCP requires the most code. Of course, TCP provides the most functionality of all protocols considered, and handles the most types of errors. By contrast, IP requires slightly more than half as many lines of code because it does not need sophisticated retransmission or acknowledgement. Even IP, ICMP, and UDP together do not account for as much code as TCP.

The size of the network interface code may seem excessive, but readers should recall that it includes the device driver code as well as the interface between IP and the device. Device drivers are inherently hardware-dependent. They usually require many lines of code to handle the details of DMA memory, hardware interfaces, and hardware interrupt processing. Thus, the code needed for the interface is not unusual.

The amount of code required for SNMP and OSPF provide the only real surprises. SNMP accounts for nearly as much code as TCP even though it performs a significantly less sophisticated service. The reason is simple: unlike TCP, the SNMP design uses variable-length encodings for every field (including the length fields themselves!). Thus, extracting even a simple integer field from an SNMP message requires much computation, even though current computers cannot handle more than 32-bit integers. In fact, the SNMP code contains an ad hoc parser for a cumbersome and tedious language. To summarize:

The variable-length field encoding used by SNMP makes the code much larger (and slower) than the other TCP/IP protocols.

^① The groups are listed here in decreasing order by size.

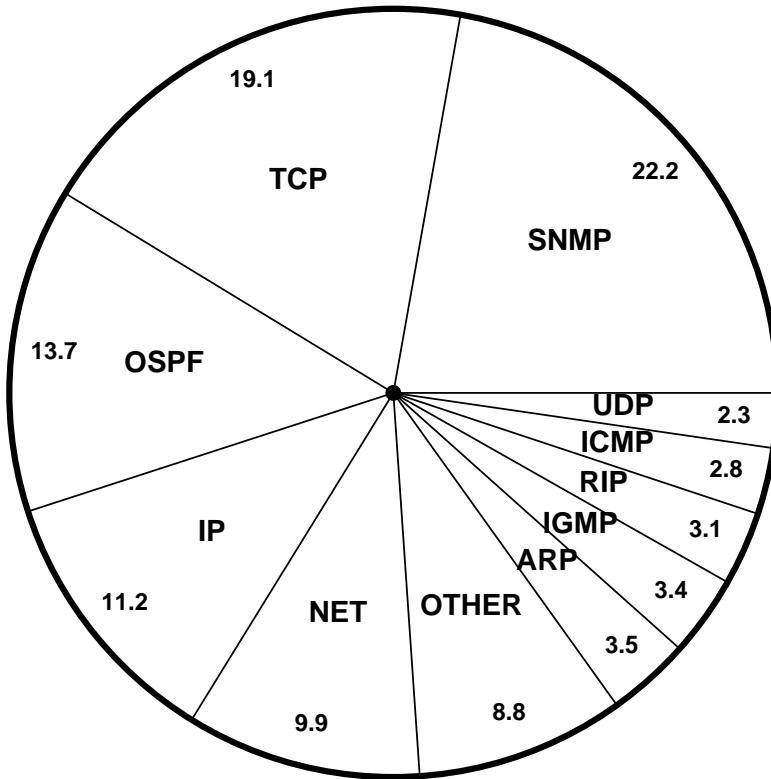


Figure 23.1 Lines of code used to implement each major protocol as a percentage of the total. Category OTHER includes miscellaneous utility functions, such as the one used to compute checksums, as well as initialization code.

The size of OSPF code may also seem surprising — it contains three and one-half times the code required for RIP. Because the protocol attempts to accommodate a variety of internet topologies, it must handle many alternatives. Production code for OSPF may be significantly larger than our minimal version because it must allow a network administrator to partition an autonomous system into areas and import external routes.

23.4 Functions And Procedures For Each Protocol

Figure 23.2 shows another measure of the code. It reports the number of procedures and functions used to implement each protocol as a percentage of the approximately 360 routines used for the entire system. The counts exclude operating system procedures and general-purpose library routines that are not part of the TCP/IP software.

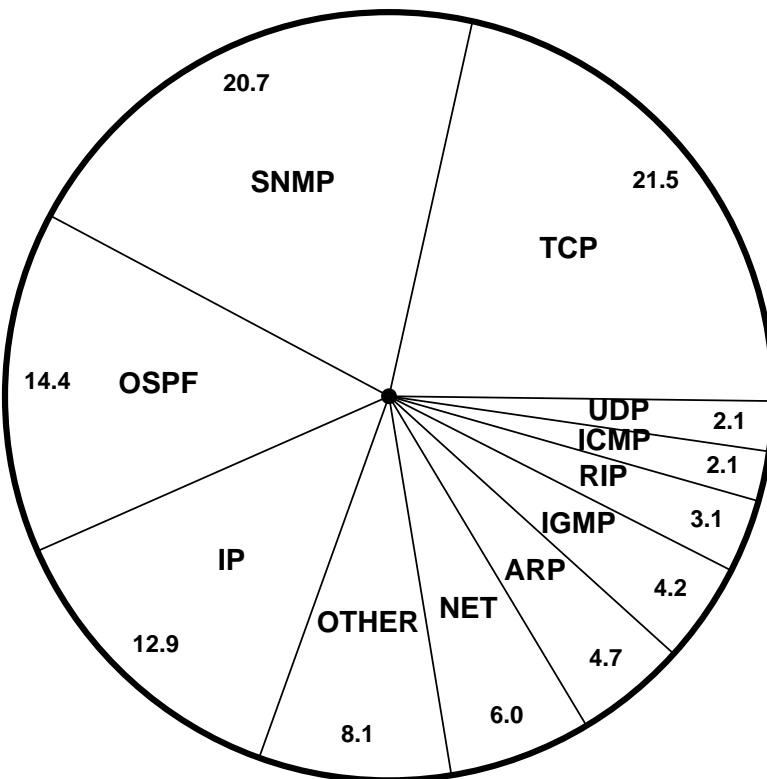


Figure 23.2 Number of functions and procedures used to implement each protocol as a percentage of the total. The chart follows the order Figure 23.1 uses.

Of course, individual coding style can influence the number of procedures used. Although several programmers have contributed to the code considered here, they all started by studying the Xinu system and adopting its style. One person wrote most of the code, and all code has been reviewed to make it conform to the desired style. Thus, it seems reasonable to assume that the coding style is relatively uniform, and that variations among individual programming talents do not account for significant differences.

23.5 Summary

An analysis of the code provides few surprises. As expected, TCP requires the most code because it provides the most services and handles the most problems. In fact, TCP accounts for slightly more code than IP, ICMP, UDP, and ARP combined. The network interface accounts for a large portion of the code because it includes device drivers.



23.6 EXERCISES

1. The analysis reported here came from a version of the code available in 1994. Obtain a machine-readable copy of the software and compare values reported here to those for later versions of the software.
2. Should ASN.1 procedures be included when counting SNMP code? Explain.
3. The example code provides gateway functions. What percentage of the code can be eliminated if it runs on a host that has only one network connection?
4. Obtain a machine-readable copy of the software described in the text. What percentage of OTHER code accounts for domain name system software? Explain the
5. Estimate the total memory space required for each protocol by estimating the size of the data structures each uses.
6. To verify that the commenting style does not differ among protocols, obtain a machine-readable copy of the software, build a program that removes comments and unnecessary white space from the code, and apply the program to all procedures. Does the ratio of sizes of compressed code to uncompressed code differ for different protocols?
7. Compare our implementation of TCP/IP to another one (e.g., the one distributed with BSD UNIX). Are there significant differences in lines of code or number of functions and procedures?
8. Extend our implementation of OSPF to permit a system administrator to configure areas and interface costs. How much additional code is needed?



24 Appendix 1 Cross Reference Of Procedure Calls

24.1 Introduction

Because any software system as large as TCP consists of hundreds of procedures stored in hundreds of files, studying the code can be difficult. Reading code from a large system is especially frustrating simply because the reader cannot easily find procedures and variables. In particular, it is most difficult to find references to a given procedure, because they can appear anywhere.

Computer scientists formalize the relationship among procedure calls into a call graph. The call graph for a set of procedures is a directed graph in which each node corresponds to a single procedure, and an edge from node A to node B means procedure A calls procedure B. For simple programs, a pictorial representation of the call graph suffices. For large systems, however, a pictorial representation becomes too large and complex to make it useful; textual representations become necessary.

This appendix provides a textual representation of the call graph for all procedures used in the text. It expresses the relationship is tailed by as well as the usual calls relationship. The information is organized into an alphabetical list of all procedures and in-line macros used in the example TCP code^①. Each entry in the alphabetical list gives the name of a procedure or macro, the name of the file in which the definition appears, the page number in the text on which the file appears (if it does), the names of all files that contain references, and the names of all procedures referenced in the defining file. The cross-reference uses files because the page on which a file occurs can be computed easily. Files that contain multiple procedures have all references listed under each procedure.

a1read in t inalrwint.cpg. 468 called in: alnvval.c

^① To make names easier to find, they have been sorted with uppercase and lowercase letters treated identically, and with underscores ignored. Thus, one finds TCP_HLEN immediately before tcpshowmuch.



a 1 read ten In alrwlen.cTp£. 466 called in: STia2b.c snparse.c
alreadyoid in alrwoid.c.,pg.470 called in: alnvval.c sna2b.c
alreadyval in alrwval.cpg. 472 called in: sna2b.c
calls: alreadyint alreadyoid alwriteint alwritelen alwriteoid
A1_SIGNED macro in asn\Xpg.464 called in: alnvinUc
alwriteint in alrwint.c, pg.468 called in: alnvval.c
alwritelen in alrwlen.c, pg. 466 called in: alnvval.c inksnmp.c snb2a.c
alwriteoid in alrwoid.c, pg. 470 called in: alnvval.c snb2a,c
alwriteval in. alrwval.c, pg. 472 called in: snb2a.c
calls: alreadyim alreadyoid alwriteint alwritelen alwriteoid
addarg in addarg.c called in: sheil_c calls: isbadpid
arpadd in arpadd-c, p%. 49 called in: arp_in,c calls: arpa^loc
arpaUoc in arpalloc.c, pg. 54 called in: arpadd.c hgarpadd.c
net write .c calls: arpdq

Tntemciworking With TCP/IP Vol II

arpdq inarpdq.c,^. 57 called in: arpalloc.c arptimer.c calls: icmp
arpfind in arpfind.c, pg. 44 called in: arp_jn.c hgarpdel.c netwriee.c
arpin in arp.in.c, pg. 51 called in: ni_in.c
calls: arpadd arpfind arpqsend hs2net net2hs
arpinit in arpinit.c, pg J£ called in: netstart.c
arpprlnt in arppriru.c called in: x_aip_c
arpqsend in aipqsend.c, pg. 50 called in: arp_in.c calls: netwrite
arpsend in aipsend.cpg. 46 called in: arptimer.c netwrite.c calls: hs2net
arptimer in arptimer.c,^, 56 called in: slowtimer.c calls: arpdq arpsend
ascdate in ascdare.c called in: x_iate,c x^who.c calls: isleap
BTOP macro in dma.h
BYTE macro in nelwork.h, pg. 580 called in: fclient.c fingetd.c K_dumper.c
x_fmgr.c x_ns.c



■ 1

|B i»Y>_ -.

Procedures And In-Lme Macros

555

ddifull inddiJull.c,j^.43J caiiedin: ospf_ddin.c caiis: nb mismatch
cksum \n cksixm.cr pg., 72 called in: dd_xrmi.c icmpx icmp_inx igmp.c igjnp^in.c
ipfsend.c ipproc.c ippuptx lsa_send,c l&r_xmiLc ospfcbeckx ospf_hsend.c
DBHASH macro in 0sptf_db.f1 called in: db_lookupx db_update.c
dbjtiit in db_init.c called in: ospfinitx
dbjookup in dbjookup x caiiedin: db_update.c lsack_in.c lsr_inx lsr_queue.c
lsu_in.c
db_new in db_new.c called in: db_update,c
dbnlink indb_n]inkx called in: db_resync,c cat Is: db_update
db_resync in db_resync.c called in: if_elecx calls: db_nlink
dbrlink in db_riinkx
dbupdate in db_updateex called fn: db__nlink.c calls: db_lookup db_new
ddiexchng rn ddLexchng c,/*#. 429 caiiedin: ospf_ddinx calls: lsr_queue
nb_mismatch
ddiexstart in ddi_e\5startx, pg. 428 caiiedin: ospf_ddinx calls: lsr_queue
ddqiJftue in dd_queue.c, pg. 421
IX caiiedin: ddi^exstart.c iib_mism-atch.c
.;<; nb_reform.c nb_switchx ospf_ddin,c
calis: Iss_build ospfddtmp1
ddxmit in dd_xmit_c, pg. 424 caiiedin: ddi_exchng.c ddi_full.c
dd_queue.c nb_rexit.c calls: cksum ipsend ospfddtmp1
DELAY macro in siu,h
dn_cat macro in domain.h caiiedin: ip2naroex name2ip.c
dnparse in dupars^c caiiedin: tcpbind.c calls: isdigit
dot2ip in dot2ip.c caiiedin: name2ip.c x_conf.-c x_igmp,c x_routex x_snmp.c



dsdirec macro in disk.h
dssyilc macro in disk.h
echod in echod.c
ECHOMAX macro in icsetdatax, pg- *42
echop in echod.c
efaceinil in initgate.c caiiedin: inilhostc vails: ethrncest nefnurn nadd setmask
egp inegpx
ethcntl in etncntlx calis: echsm

556

ethdemux in elhdemux.c called in: ^thiriter.c talis: net2hs ni_in
ethinit in ethinitc calls: eihstrt hiS low 16
ethint in ethint.c calls: ethinier
ethinter inethinter,c called in: ethint.c calls: ethdemux ethwstrt printcsrO
ethmcast in ethmcasLc, pg. 154 called in: inilgate.c
ethread in ethread.c
ethstrt in edistrtc called in: ethcnLc ethtnitc calls: hi3 lowl6 printcsnO
ethwrite in ethwrite ,c calls: ethwstrt hs2ner
ethwstrt in ethwstrt.c called in: ethinter.c ethwritcc oth write .c
EVENT macro in icpfsm-h, pg, 200 called in: tc pouLc tqdutnpx
fd_to_dd macro in io.h
fgetc macro in io.h
findfd macro in io.h
finger in fclicnt.c, pg. 326 calls: name2ip
flngerd in fingerd.c
called in: fserver.c calls: Ifing

Cross Reference Of Procedure Calls Appendix 1

firstid macro in q.h
firstkey macro in q.h
fopen macro in name.h
fpulc macro in io.h



freestk macro in menuh
getaddr in getaddr.c calls: rarpse
get_bit in ethctrl.c calls; cthsCrl
getchar macro in io.h
gethashbii in ethctrl.c
calls: ethstrt
getiaddr in geiaddr.c called in: getname.c getnetc
netstaitx calls: jarpsend
getiname in getname.c called in: netstartc calls: getiaddr ip2name
getinet in getnei.c
calls; getiaddr
getmib in snhash.c,pg, 459 called in: snprint.c snrslv.c calls: oidequ
getname in getname.c called in: login.c name2ip.c rwhod.c shell.c sninat.c
x_upptime.c K_who.c calls: getiaddr ip2name
getnet in getnetx calls: getiaddr

Procedures And In-line Macros

GETPHYS macro in dma.b
getsim in inithostc calls: efaceinit ofaceinit
getutim in gecutim,c called in: Login.c netstaii.c n*hodc
shell.c calls: net2hl nei2xt
gname in name2ip.c calls: dn_cat dot2ip getname hs2net nei2hs
ha shirt it in snhash.c, pg. 459 called in: sninit.c calls: oifrequ
hashoid in snftash.c, pg. 459 calls: oidequ
hgadd inhgadd^?^.5i called in: hgjouvc calls: hgarpadd
hgarpadd in hgarpadd.c, pg. J53 catted in: hgadd .c calls: aipalloc
hgarpdel inhgarpdel.c,/^ 156 called in: hgleave.c calls: arpfnd
hginit mhginn.c.pg. 167 called in: n-etstmc calls: tigj-oin rtadd
hgjoin in hgjoin.c, pg. 157 called m: hginit.c ospfifiniLc
x_igmp.c calls: hgadd hglookup igmp rtadd



hgleave in hgleave.c, pg. 166 called in: jjgmp.c calls: hgarpdel hglookup rtdel tmclear

hglookup in hglookup.c, /?#, 150 called in: hgjoin.c hgleave.c ignip_in.c udpsend.c

hgprint in hgprim.c called in: x_jiet.c calls: printone

rtgrand in hgrajid.c, /J#. 162 called in: igmp_settimers.c

hiS macro in network.h, j)g. 580 called in: ethiniLc ethstrt.c

hl2net macro in networkh, pg. 5ffl called in: ripadd.c riprepl-c rwhod.c tcph2nei.c

Eis2net macro in netwoik-h, pg. 58Q called in: arpsend.c arp_in.c ethwrite.c ip2namcc iph2net.c name2ip.c rarpsend.c ripadd.c tcpcsum.c icph2nel.c tcprmss.c udpcsum.c udph2n.e(.c

ibdisp macro in iblock.h ibt-odb macro in ibLock.h

ice IT ok in icerrok.c, pg. 139 called in: icjnp.c calls: isbrc

558

Cross Reference Of Procedure Calls Appendix 1

iemp in icmp.c, pg. 137 called in; arpdq.c icmp_in_c

ipftimer, c ipproc.g ippputp.c

ipredJrect.c locaJ_j*ut_c nets-fartc

udp_tti.c x,,pin.g.c culls: cks-um icen*ok icsetbuf

icsetriata icsetsrc ipsend

iemp in in tcmp_in.c, />#. 130 called in: locaLoutc calls; cksum iemp icreditrect icsetsrc ipsend netmask setmask

tcreditrect in icreditrectx, pg. 132 called in: icmp_in.c calls: neiciask rtadd rtdel rtfree rtget

icsetbuf in icsetbuf.c, pg. 140 called in; icmp.c

icsetdata in icsetdatax, pg. 142 called in: icmp.c calls: isodd

icsetsrc in icsetsrex, pg. 136 called in; icmp.c icmp_in.c calls: netmatch

ifelcct in if.elecLcp, 402 called in; ospf.c o-sptimer.c calls: db_resync if_electl nb_reform

ifelectl in ifjelectlx+pg. 400 called in: if_elect.c



igmp in igmp.c, pg. 159 called in: hgjoin.c igmp_updaie.c calls: cksum ipsend
igmpin in igmp_in.c, pg. 164 called in: local_out.c calls: cksum hglookup
igmp_seUimers tmclear

igmp settimers in igm.p_settimers.c, pgJ60 called in: igmp_in.c calls: hgrand imser
igmpupdale in igmp_update.c, pg. 163 calls: igmp

IGJTYP macro in igrnp.h, pg. 148 called in: igmp_in.c

IG_VER macro in jgmp.h, pg, 148

called in: igmp_in.c

initgate in initgate.c called in: netstart.c calls: ethmcast netnum nadd setimask

inlhosLc in inithosLc called in: netstan.c calls; cfaoeinit ofaceinit

ip2do(in ip2dot.c called m: ip2narne.c x_netc x_routes.c x_snmp.c

ip2naine in ip2name.c called in: getname.c s_conf.c

3c_routes*c calls: dn_cat bs2net ip2dcl net2hs

IP_CLASSA macro in ip.hT pg. 70 called in: netmask.c netnum.c rhashx

IP_CLASSB macro in ip.h, pg. 70 called in: netmask.c netnuiruc rhash.c

IP_CLASSC macro in ip.h. pg. 70 called in: netmask.c netnum.c rhash.c

Procedures. And In-Line Macros

559

IP^CLASSD macro in ip-Kpg- 10 called in: hgjoin.c hgleave.c icerrot.c netmatch.c
netwritex ripck.c rhash.c udpsend.c x_igmp.c

IP^CLASSE macro in ip.h, pg, 10 called in: ipproc-c ripo-k.c

ipdbc in ipdbc.c, pg. 73 called m: ipproc.c colls: ippputp isbrc rtfree rtget

ipdoopts in ipdoopis.c, pg. 104

ipdstopts in ipdftopis.c, />£, 104 called in: local_out.c

ipfadd in ipfadd.c,P£ 118 called in: ipreass.c

ipfcons in ipfcons.c, pg. 121 called in: ipfjoin.c

ipfhcopy ia ipfhcopy.c,pg.112 called in: ipfsend.c ippputp.c



ipfinit in ipfbmt.c.pg. 124 called in: netstart.c

ipfjoin in ipfjoin.c, pg. 119 called in: ipreass.c calls: ipfcons

tpfsend \rt ipfsend.c, pg. IJQ culled in: ippup.c calls: cksum ipfhcopy iph2na
IP_HLEN netwrite

iptimer in ipftimer.c, p£. 122 called in: stowntimer.c CQlte: icmp

iph2net in iph2net>c.pg. 76 called in: ipfsend.c ippoc.c ippup,c calls: hs2net

IP_HLEN macro in ip.h, pg. 70 called in: ddi_exchng.c ddi_exstart.c d-di_Jull.c
tcmpx icmp_in.c icsetdaia.c igmp.in.c ipdstopts.c ipfconsx ipfhcopy,c ipfjoin.c ipfsend.c
ippocx ippup,c ipsend.c lsack_in.c lsr,in,c lsu_in,c nb_s*ilch.c ospf.c Ospfcheck.c
ospf_ddin.c ospfJiin,c tcpackit£ tcpcksum.c tcpdata.c tcpok,c tcpopas.c tepresei.c
xjptng.c

ip_in in ip_in,c, pg. 80 called in: ni_in.c

ipnet2h in ipnei2h.c, pg. 76 called in: ippoc.c locaLout.c calls: net2hs

ippoc in ippoc.c, pg. 66 calls: cksum icmp ipdbc ipgetp iph2net ipnet2h ippup
ipredirecl rtfree riget

ippup in ippup.c, pg. 108 called in: ipdbc.c ippoc calls: cksum icmp ipfhcopy
ipfsend iph2net IF_HLEN net2hs netwrite

ipreass in ipreass.c, p%. 116 called in: local_out.c calls: ipfadd Ipjotn

ipredirect in ipredirect.c, pg. 144 called in: ippoc calls: icmp netmask rtfree rtgec

ipgetp in ipgetp.c,p^<W called in: ippoc.c

560

ipsend inipsend.c*/^ 78 called in: ddLxmitx icmp.C icmp_in,c igmp.c lsa_send.c
lsr_xmit.c ospf_hsend.c icpactit.c icpreset.c tcpsend.c udps&nd,c

isalnum macro in ctvpe.h

isalpha macro in ctype.h called in: x_snmp.c

isascii macro in ctype.h

isbaddev ma-croin io.h called in: A_mount,c

isbadpid macro in proc.h cailed in: addarg.c raipju.c udp_in.c

isbadport macro in ports.h

isbadsem macro in sem.h

isbrc in isbrc.c, pg. 75 cailed in: icerrok.c ipdbc.c nermaich.c net write, c



iscntrl macro mctype.h
isdigit macro in ccypeJi cailed in: dnparse.c *_snmp.c
isempty macro in q.h
isleap macro in date.h called in: ascdate.c
islower macro in ciype.h
isodd macro in kernel.h, pg. 582 cailed in: icsetdata,c
isprint macro in ctype.h called in: sprintf.c

Cross Reference Of Procelure Calls Appendi* I

isprshort macro in ctype.h
ispUIKt macro in ctype.h
i&space macro in ctype.h
ISUOPT macro in icpuopt.c, pg. 350
isupper macro in ciype.h
ISVAUD macro in dma.h
isxdigit macro in ctype.h
lastkey macro in q.h
level macro in io.h
lexan in lexan.c called in: shcll.c
Iflng in Ifing.c called in: fingerd.c x,,finger_c
1ocal_OUt in local_out.c cailed in: netwrite.c calls: iemp tcmp_in igmpjn ipdstc-pts
ipnei2h ipreas
login in login, c calls: geEname geiutini
lowl6 macro in network.h* pg. 580 called in: ethinit.c ethsrrt.c
Isaadd in lsa__add.c called in: lsr_in.c
Isackin in lsack_in.c called in: ospf.c calls: dfcjooup
Isasend in lsa_send.c called In: Lsr_in.c calls: cksum ipsend

Procedures And In-Linc Macros

Isaxmit in lsa_xmitc
)sr add inlsr_add.c called in: lsr_queue,c calls: ospflsrtmpl
lsr_check in lsr__check.c called in: lsu_in.c



lsr_in. in lsrjnx, pg.432 called in: ospf.c calls; dbjookup fsa_add nb_mismalch
ospflstmp

lsr_queue in lsr_queue.c called in: ddi_exchng.c ddi_exstart.c calls: dbjookup
isi_add isr_xmit

Isr^xmit in lsr_xmil.c called in: lsr_queue_c ospflimerc calls: cksum ipsend

LSS_ABORT macro in lss_bui!d-c,

pg. 434

Issjbuild iolss_buildc,j# 434 called in: dd__queue,c calls: tib^cleari ospfddtimpl

Isuin in)su_tn.c called in: ospf.c calls: dbjookup lsr_check

ltim2ut macro m date.h

major macro in sysrypes.h

makedev macro in sys-type&.h

marked macro in maikJi called in: rwhod.c x_who,c

max macro in kemehh. pg. 582 called in: tcpiwindow.c

561

min macro in kernel.b, p#. 582 called in: tcppersisLc tcpexmt.c tcpsms&.c
icpsndlen.c icpwinitc tcpxmit.c

minor macro in systypes.h

mkarp in rarpsend.c calls: h&2net rtadd

MKEVENT macro in tcp fsm Ji, p*. 200 called in: tcp kic Lc IcpkiUtimers.c
icppersist.c tcp exnuc tcprtt.c tcpswindowx tepwaice ccp xmi Lc

mksnmp in mksnmp-c, pg.492 catted in: snclient.-c snmptEx calls; alw rilelen snb2a

MOVC macro in dma.h

MOVL macro in dma*h

MOVSB macro in dma,h

MOVSL macro in dma.h

MOVSW macro in dma.h

name2ip in name2ip.c called in: fclient.c x_duniperx

x_firiger,c x_ns.c x_ping.c x_snmp,c calls: dn_cat dot2ip getnamc hs2net
net2hs

nb_add m nb_add-c, pg. 408 called in: ospfjddin.-c ospf_hin.c

N_BADMAG macro in a.out.h



nfr_aok mnb_aok,c, pg. 414 catled in: nb_reform,c nb_swiich.c ospflldin.c

562

nbcJearl in nb_dearLc, pg. 405 salted in: lss_build.c nb^reform.c nb^s witch, c nbmakel in nb_makel.c, pg. 406 catted in: nb_reform.-c nb_switch,c ospf_ddin.c - nb mismatch in nb_rnismatch.c,

pg* 433

called in: ddLexchng.c ddi_full,c Jsr__in.c

itb^reform in nb_reform,c, j?g. 404

caited in: if_electc

calls: rib_aofc nb_clearl nb_makel

nb_re\mt in nb_rexmt.c, pg. 414 catted in: ospftimeix

NBSSADDR macro in a.ouOi

nb_Switch in nb_switch.c, pg. 410 called in: ospfjiin.c calls: nb_aok nb_ctearl nb_makel

N_DATADDR macro in a.out.h

nellhl macro in netwoil.ri.pg. 580 called in: geiucinu^ riprecv.c riprepl,c rwhoincLc ccpnet2h-c tcpsmss.c

net2hs macro in network.h, pg. 580 called In: arp_in.c ethdemux-C ip2name,c ipnei2h.c ippuiip.c name2ip.c ospfnet2hx rarp_inx ■ riprecv.c ripreplx tcpnet2h.c lcsmss,c udpcksum.c udpnet2h.c x_rls,c

netlxt macro in date.h called in: getntim.c

Cross Reference Of Procedure Calls Appendix 1

netmask in Tkmask.cpg. 92 called in: icmp_inrc icredirect.c

ipredirecLc rarp_in,c riprecv.c

seimaskx calls: netnum

netmatch in netmatcrLc ./?£. 91 called in: icsetsr.c rtgetc calls: isbrc

netnum in netmimx, pg* 90 called in: inilgate.c netmask.c rarp_inx ripadclc

netstart in netstarc calls: arpinit getiaddr getiname getulim hginit icmp initgate inithost ipfinit rtadd rwho udpecho

netwrite in netwriie.c, pg. 47 called in: arpqsend.c ipfsend.c

ippotp.c calls: arpalloc aipfind arpsend isbrc



local_oirt

NIGET macro in netith, w 28 called in: ipgetp.c

iIMII in fli_irt.C, pg-35 called in: elhdemux.c calls: raip_ici
nonempty macro in q.h

N_PAGSIZ macro in a.ouLh

N_SEGSIZ macro in a.out.h

NjSTROFF macro in a.out.h

N_SYMOFF macro in aout.h

N TXTADDR macro in a-ouLh

net dump in netdump.c

N TXTOFF macro in a.out.h

Procedures And tn-Line Macros

563

ofaceirit in initgat&c called in: inithosLc calls: eihmcast netnum rtadd setmask
oidequ macro in &nmp.h, pg. 461 called in: snhash.c snrsW.c

ospf in ospf>c*pg. 416 calls: if_elect lsack_in Lsr_in Isujn ospfcheck ospfinit
ospfnet2h

ospfcheck in ospfcheckx, pg. 436 called in: ospf,c calls: cksum

OSpf_ddit1 in ospf__ddin.c* pg. 426 called in: ospf.c

calls: drJi_exchng ddr_exstari ddi_full dd_queue nb_add nb_aok nb_makel

ospfddtmp1 in ospfddtmp1.c, pg. 423 called in: dd_queue.c dd_xmit.c lss_build.c

ospfliello in ospfhello.c, pg. 398

ospf_hin in ospMiin.c, pg. 406 called in: ospf.c calls: nb_add nb_switch



ospfhsend in ospf_hsend.c, pg. 394 called in: ospfliello.c calls: cksum ipsend
ospfhtmpl

ospfhtmpl in ospfhtmpl.c, pg.^96 called in: ospf_bsend.c

ospflfinit in ospflfinil.c called in: ospfinit.c calls: hgjoin

ospfjn iROsyf_inx,pg.4I6 called in: loc&l_out.c

ospfinit in ospfirtiLc called in: ospf.c calls: dbjnit ospfifinit

ospflsrtmpl in ospflsrtmpLc called in: lsr_add.c

ospflstmpl in ospflslrapl.c called in: Isrjiuc

ospfnet2h in ospfnet2h.c,pg. 436 called in: ospf.c calls: net2bs

o-spftimer in ospftimer.c, pg. 412 called in: slowtimerc calls: if__elect lsr_xmit
nb_rexmt

ottlimit in othlnit.c

othwrite in oihwtite.c calls: ethwsiff

printcsrO in eihinii.c called in: elhinter.c ethstit.c colls: eths*rt hiS lowl6

print-one in arpprinzx called in: hgprins.c

PTOB macro in dma.h

putchar macro in io.h

rarp_in in rarp_in.c called in: ni_in.c calls: isbadpid net2bs netmask netnum
setmask

rarpSENT in raipsend.c called in: getaddr.c calls: h.s2net rtadd

564

resolve in name2ip,c calls: dn_cat dot2ip getname hslnet net2hs

rip in ripin.c., pg- 365 calls: ripcheck riprecv riprepl

ripadd in ripadd.c.,pg. 374 called in: ripsend.c

calls: hL2net hs2net netnum riprnetric ripstart

ripcheck in ripcheck.c.pg. 366 called in: ripin.c

ripifset in ripifset.c, pg. 373 called in: ripsend.c calls; rtfree rtget

riprnetric \n ripirittricc, pg, 376 called in: ripadd.c

ripok in ripok.c, pg. 370 called in: riprecv.c

ripout in ripout.c, pg. 378 calls: ripsrnd

riprecv in riprecv.c, pg. JtfS called in: ripin.c



calls: net2hJ net2hs netmask ripok rtadd rtfree rtgei
riprepl in riprepl.c, pg. 370 called in: ripin.c
calls: h!2net net2hl net2hs ripsend nfree nget udpsend
ripsend in ripsendc, j??. S72 called in: ripout.c riprepl.c calls: ripadd ripifset
udpsend
ripstart in ripstaitc. j>g. i77 called in: ripadd.c

Cross Reference Of Procedure Calls Appendix I
roundew macro in mem-h rourtdmb macro in mern.h
rtadd tnnadd.cpg.98
called in: hginit.c hgjoin.c icredirect.c initgate.c netstart.c rarpsend.c riprecv.c
setmask.c x_rouie.c
calls: rhas-h rtinit rtnew
rtdel inrtdel.c.pg. 102 called in: hleave.c icredirect.c
setmask.c srt_set.e x_n>ute.c calls: nhas-h
rtdump in rtdump.c calls: ninit
RTFREE macro in raMch, pg. 87 called in: rtadd.c rtdel.c rtfree.c rttimer.c
rtfree in rtfree.c, pg. 103 called in: icredircct.c ipdbc.c ippoc.c ipredict.c ripifset.c
riprecv.c riprepl.c tcpbind.c tcpcon.c tepwinite udpsend.c x_route*c
rtget inrtget.cpg. 94
called in: icredirecLc ipdbc.c ippoc.c ipredirecLc ripifset.c riprecv.c riprepl.c
tcpbind.c tcpcon.c tepwinite udpsend.c x_rouK.c
calls: netmatch rhash rtinit
rhash inrhash.^pg. 93 called in: nadd.c rtdel.c rtget.c
rtinit inrtink.c, pg. 96 called in: rtadd.c rtdump.c rtget.c
rtnew in rtnew.c, pg. 101 called in: nadd.c

Procedures And In-Line Macros

rttimer in mimtr.c, pg. 96 called in: slowtimer.c
rwho in rwho.c called in: netstarLc
rwhod in rwhod.c calls: getname geilim hl2net marked tidpsend
rwhoind in cwhoindx calls: ne(2hl)
saefindnext in saejlndn.c, pg. 510 called in: sae_ge(f.c sae_getn.c



sae_get in sae_get.c, pg. 500~ called in: sae_getf.c sae_getn.c
snmib.c calls: sae_maich
saejgetf insae_getf.c, pg. 508 called in: snmib.c calts: sae_fmdnext sae_get sip2ocpj
sae_getn w sac_gctnx, pg. 509 called in: snmib.c
calls: sae_findnex£ sae_get sae_jnatch sip2ocpy
sae^match in sae.niatch.c, pg. 505 called in: sae_get.c sae_getn.c
snmib.c calls: soipequ
sae_set in sae_set.c, pg. 5H called in: snmib.c
satcmp in sa(_fmdn>c, pg. 518
sat findnext in sat_findn.c, pg. 518 called in: sa*_geif.c sat_gem.c

565

sal_get in sat_gei.c, pg. 514 called in: sat^getf.c sat_getn.c snmib.c
sat_gelf in sai_getf.c, pg. 515 coiled in: snmib.c calls: sip2ocpy
satgetn in sat_getn.c, pg. 516 called in: snmib.c calls: sip2ocpy
sat_match in satjnatch.c, pg. 512 called in: sal_getc sat_getruc
sat_set.c snmib.c calls: soipequ
sat_set in sat_sei.c, pg. 520 called in: snmib.c
SECYEAR macro in clock.h
SEQCMP macro in tcp.h, pg. 196 called in: tcpacked.c tcpdata.t tcpgetdata.c
ccpsend.c tcpswindow.c
sel_bit in ethcntLc calls: eihstrt
seterr macro in snrslv.c, pg. 485
setmask in setmask-c, pg. 134 called in: icnip_in.c initgate.c
rarp_in.c calls: netmask itadd rtdel
SHA macro in arp.htpg. 42 called in: arpadd.c arpsend.c arp_in.c raipsend.c
shell in sheSL.c calls: addarg getname getutim lex an

*6G

sif_get in sif_get.c, pg. 522 calied in: sif_getf.c sif_getn.c snmib.c calis: sif_match
sif_getf in sif_getf.c. pg. 526 called in: snmib.c calls: sif_get
sif_getn in siflgetiuc, pg. 527 called in: snmib.c calis: sif_get sifjmatch



Sif_match in 5if_matcb.c, pg. 522 called in: sif_get.c sif_getn.c sif_set.c snmib.c
sifset in sif_set.ch pg. 528 colled in: snmib.c calls: aif_match
sipZocpy in snoip.c called in: sae_getf.c sae_getn.c sal^getf.c sai_getn.c srt_gerf.c
srt^gem.c stc_getf.c stc_getn.t

sizeof in snmib.c, pg. 452 calls: sae_j;et sae_getf sae_getn sae_match sae_set
sif_jget sif_gerf sif^gern sif_match sif_sei snleaf sntabLe srt_get &rt_getf sit_getn
srt_match stl_set stc_get stc_getf stc_gein s(c_match stc_set

slowtimer in sk>wtimer.c, pg. 81 caffs: arptimer ipftimer ospftimer rttimer

sna2b in sna2b.c, pg. 484 called in: snclient.c snerr.c Hnmpd_c calls: alreadlen
alreadoid alreadaval

Ctass Reference Df Procedure Calls Appendix I

snb2a in snb2a.c ,'/?#. 494 called in- mksnmp.c calls: alwriden alwriteoid
alwritevai

Snellen! in snclient.c, pg. 497 colled in: x_snmp.c calls: mksnmp sna2b snparse

&nerr in snerr.c called in:)t_snmp,c calls: sna2b snmpprint_objid

snfreebl in snfreebl.c, pg. 496 called in: snmpd.c x_snmp.c

sninit in sninit.c, pg. 499 called in: snmpd.c x_snmp.c calls: getname hashinit

snleaf in snleaf.cTp#, 488 called in: snmib.c

snmpd in snmpd.c, pg. 478 calls: mksnmp sna2b snfreebl sninit snparse snrslv

s n mppri n t in snpri nt. c called in: x_sntnp.c calls: getmib isprint

snmpprintobjid in snprint.c

called in: snenr.c calls: getmib isprint

snmpprint_objname in snprint.c calls: getmib isprint

snmpprinlval in snprint.c calls: getmib isprint

snparse in siiparsex, pg. 4H0 called in: snclient.c snmpd.c calls: alreadlen

Procedures And In-Line Macros



snrslv insnrsiv.cpg. 485 called in: snmpd.c calls: gelmib oidequ
sntable in sntable.c, pg. 490 called in: snmib.c
sntcpemp in stc__fuidn.c,pf. 544
so2ipcpy in snaip.c called in: stc_match.c
soipequ in snoip.c called in: sae_Tnatoh.c sat_match.c srt_mfltch_c
SPA macro in arp.h, pg. 42 called in: arpadd.c arpsend.c arpjn.c rarpsemdc
srt_findnext in srt_findn.c, pg. 535 called in: srt_getf.c sct_getn,c
srt^get insri^get.c, pg. 530 called in: snmib.c srt_getf.c srt_gecn.c calls: srt_match
srtjjetf in sTt^getf.c,/>g. 532 called in: snmib.c calls: sip2ocpy srt_findnext &rt_g«
srt_j;etii in srt_getn.c, pg. 534 called in: snmib.c calls: sipZocpy srt_findnext
srt_get srt_maich
srt^match in sit__maich.c, pg. 529 called in: snmib.c srt_get.c srt_getn.c
srt_set.c calls: soipequ

stc_fimUiext in stc_findn.c, pg. 544 coiled in: stc^getfx stc^getn.c
stc_get in stc_getc>pg, 540 called in: snmib.c ste_getf.c
stc_getn_c calls: stc.match
stc_getf instc_getf.c,/jg. 541 called in: snmib.c calls: sip2ocpy scc_findnexl slc_gei
stc_getn in stc_gecn,c, pg. 542 culled in: snmib.c
calls: sip2ocpy ste_fmdncxt stc_get stc_maich
stcmatch in stc_match.c,pg. 5JS called in: snmib.c stc_gel.-c slc_getn.c calls:
so2ipcpy
stc_se* in stc_set.c, pg. 545 called in: snmib.c
streku macro in snmp.h, pg. 46J called in: jL_snmp.c
SVINT macro in snmp.h, pg. 461
called in: alrwvaL.c sae_g-et,c sat_jetf.c sat_setx sif_get.c sif_setc snleafx srt_gei.c
srt_setc stc^getx x_snmp.c
SVID macro in snmp.h, pg. 461 called in: alrwvaLc sae_jfet.c sat__get.c
sat_set.c srt_getc srt_set.c stc_get.c x_snmp.c
SVOID macro in snmp.h, pg. 461 called in: snleaf.c x_snmp.c



srt__set insrt_set.c, pg.5J6 called in: snmib.c calls: ridel srl_match

SVOIDLEN macro in snmp.h, pg. 461 called in: snleaf.c x_snmp.c

563

SVSTR macro in sjimp.h* pg. 461 called in: alrwval.c sat^get.c sat_set.c s.if_get.c snieaf.c x_snmp.c

SVSTRLEN macro in snmp.h, pg. 461 called in: airwvaLc sat_getc sat_sei.c s.if_get.c sjtleaf.c x_snmp.c

SVTYPE macro in snmp.h, 461 called in: alrwvaLc sae_get.c sat^get.c sat_set.c sif_getc sif_set.c snieaf.c srt_gei.c srt_set.c stc_getx x_snmp.-c

TCB macro in tcp fsm.h, p^ 200 called in: tcpout.c tqduinp.c

tcballoc mccalloc.c,pfi,202 called in: tcplisten.c tcpmopen.c

IcbdealLoc in tcbdealloc.c, pg. 203 called in: tcpdose.c icpclosingx

tcpcon.c tcpout.c tcpsynrcvd.c

tcpLimewait.c calls: icpkiUtimers

tcpabort in tcpabort.c, pg. 236

called in: tcpclosewail .c tcepestablishet.i.c tcpfinLc tcpfm2,c tcplastack,c icpremu.c tcpsynrcvttc

calls: tcpkillitners lepwakeup

tcpacked in tcpacked hc, pg. 301 called in: tcpclosewait.c tcpclostng.c

tcepestablishe<Lc tcpfml.c tcpfin2.c

tcplastackx tcpsyn:rcv<Lc tcpsynsent.c

tcptimewail.c calls: tcpakil tcpostate icppreset

tcprrt

tcpackit in tcpackit.c, pg. 303 called in: tcpacked.c tcpinp.c calls: ipsetid tcpcksum tcph2net toprvwindow

LTOSS Reference Of Procedure Calls Appendix I

tepbind in icpbiad.c, pg. 332 called in: tcpmopen.c calls: dnparse rtfree rtget icpnntp

tcpcksum in tcpcksum.c, pg. 203 called in: tcpackit.c tcpinp.c tcppreset.c tcpsendx calls: hs2net

tcpclose in xepc\o\$e.cypg. 343 calls: tcbdealloc tepkick



tcpclosed in tcpclos&d,c,#£. 218 called in; tcpswitch.c calls; tcpriset

tcpcloscwait in tcpclosewait.c, pg. 226 called in: tcpswitch.c calls: tcpabort
tcpacked tcpriset repswindew

tcpclosing in tcpclosmg.c.pg. 221 called in: tcpswitch.c calls: tcbdealloc tcpristed
tcpriset lepwit

tcpcnfl in tecitt\c*pg. 345 calls: tcpLq lep&tat icpuopt lepwr

tepcon in tcpcon.c, pg. 335 called in: tcmopen.c calls: rtfree rtget tcbdealloc
tepkick

tepdata in icpdata.c, pg. 229 called in: tcpestabled.c tcpfinl.c

tcpfin2.c tcplisien.c tcpsynrcvd.c

tcpsynsent.c tcptimewaitx calls: tepdodat tepkick

tepdemux in tepdemux./jg, 210 called in: tcpnip.c

Procedures And In-Line Macros

569

lepdodat intepdodau, pg. 232 t called in: tepdafcte

calls: Icpwakeup tfcoalesce tfminsert

tcpestabled in tcpestabled. c,

pg. 228 called in: tcpswilch.c calls: tcpabort repacked tepdata

tcpriset tcpswindow

tcpfinl in tcpfm\c9pg. 224 called in: tcpswilch.c calls: icpabort tcpristed
tcpriset tcpswindow tepwait

tcpfinJ in tcpfin2.c.p^ 222 called in: tcpswilch.c calls: tcpabort tcpristed
tcpriset tcp-wait

lepgetc in tcpgetc.c, pg. 340 calls: tcpread

tepgetdata in tcpgetdata.c,p#, 3J5 called in: tcpread.c calls: tcpklck teprwiiidow

tcpget&pace in tcpgetspace.c, pg. 264 called in: tepwix calls: tepwakeup

tcpf2net in icph2net,c,pg. 263 called in: tcpackit.c tcpriset.c

tcpsend.c calls: M2net hs2net



TCP_HLEN macro in tcpJt, pg. 196

called in: tcpackit.c tcpdata.c tcopok.c tcpopts,c tcpreset-c tcprmss.c tcpsend.c
tcpflowmuch in ccphowmuch.c, pg. 260 called in: tcpxmit.c

tcpidLe in tcpidle,c>pg.252 called in: tcpswitzcuc calls: lepxmit

tepjn in Xcpjn.c* pg. 205 called in: local_out.c

tcpinit in tcpinit.c, pg. 351 calls: unmarked

tcpinp in tcpinp.c, pg. 206 calls: tcpackit tepksum tepdemux tcpnet2h tcopok
tcpopts tcpreset

tcpiss in tcpiss.c,pf. 26? called in: tcpsyncc

tcpkick in tcpkick.c, pj?r 280 caUed in: tcpclose.c tcpcon.c

tcpdata.c tcpgetdata.c tcpostate.c

tcpsw indo w.c tcp wr. c caHs: tmleft tmset

tcpkilltimers in tcpkillmire.c, pg. 276 called in: tcbdealloc.c tcpabort.c

tcpsynsent.c tepwait. c calls: tmclear

tcplastack in tcplastack.c, p#. 227 called in: tcp-switch.c calls: tcpabort tcpacked
tcpreset

tcplisten in tcplisten.c, pg. 243 called in: tcpswitch.c calls: tcballoc tepdata tcpreset
tcpsync tcpwinlt

tcplq in tcplq.c, pg. 346 called in: tcpcntlx

tcpmcntl intcpmcntLc, pg.336

Procedures Ami fti-Line Macros

571

tcpstat in tcpstai.c,/?£. 348 called in; tcpcntlx

tcpwindow in tcpwindow .c, pg* 290 called in: tcpclosewait.c

tcpe&tablished.c tcfinl.c calls: tcpkick tmclear

tcpsync intcpsync.cpg. 238 called in: tcplisten.c tcpmopen.c calls; tcpiss

tcpsynrcvd in tcpsynrcvdcw 240 called in: tcpswitch.c calls: icbdealloc tcpabtm



repacked tepdata tcpreset

tcpsjnseni in tcpsynsent.c, pg. 239 called in: tcpswitch.c calls: tcpacked tcpdaia
tcpkilltimers tcpreset

tcptimer in tcptimer.c, pg. 272

tcptimewait in tcptimewaitc, pg. 220 called in: tcpswitch.c calls; tcbdealloc
tcpacked tepdaca tcpreset tepwait

tcpuopt in xcpvopt.c, pg. 350 called in: tcpcnil.c

tepwait in tepwait.c, pg. 219 called in: tcpclosing.c tcpfinLc

tcpfin2.c tcpsend.c tcptimewail.c calls* tcpkilllimcrs imset

tepwakeu in tcpwakeu,c,/»g. 265 called in: tcpaborr.c tcpdodat.c icpg.etspace,c
tcpread.c tcpwr.c

tepwin in tcpwinU\c,/^. 244 called in: icplUten.c calls: min rtfree rtgeE

tepwr in tcpwr.cpg, 342 called in; tcpcnil.c tcpputc.c

icpwrite.c calls; tegetspace tcpkick icpwakeu

tepwrite in tcpwrite.c, pg. 340 calls: tepwr

tepxmit in tcpxmit.c, pg. 254 called in: tcpidle.c tcpswitch.c calls: min tcephowmuch
tcpexmt tcpsend unclear tmleft tmset

tfcoalesce in tfcoalesce.c, pg. 235

called in: tcpdodat.c

tfinsert in tfinsertc, p#. 234 called in: tcpdodaLc

THA macro in aip.h».pf. 42

called in; arpsend.c arp_in_c rarpsexd raip_in,c

tmclear in rmclear.c, pg. 274 called in: hgleave.c igmp_in.c tcpkilltimers.c leprtte
tcpwindow.c tcpxtnit.c tmset.c

tmleft intmleft.cw 276 called in: tcpkick.c tepxmite

tmset in tmsetc,_pf. 278 called in: igmp_5eitimers.c tcpkick.c

tcppersist.c tcpexmt.c tcpwait.c

tcpxmit.c calls: unclear

toascii macro in ctype.h tolower macro in ctype.h toupper macro in ctype.h



c&lkdin: arpsend.c aip_jn.c raipsend.c rarp_in,c
tqdump in tqdump,c
tqwrite in iqdump.c called in: x_timerq.c
truncew macro in mein,h
truncmb macro in mem.h
udpcksum in udpcksum.c, pg. 184 called in: udpsend.c udp_in.c calls: hs2nei
necihs
udpecho in udpecho.c called in: ncistart.c
udph2net in udph2net_c, pg. 182 called in: udpsend.c calls: hs2net
udpin in udp_in.c,pf- 752 called in: local_out.e calls: icmp isbadpid udpcksum
udpnet2h
udpnef2h in udpnet2h.c, pg. 18! called in: udp_in.c culls: net2hs

Cross fctference Of Procedure Calls Appendix 1

ul2ltim macro in date.h called in: x_who.c
Xt2fiet macro in date.h ZSTimeConst macro in zsreg.h

udpnxtp in udpnxip.c, pg. JS6
udpsend in udpsendx, pg. 187 called in: riprepl.c ripsend.c rwhod.c calls: hglookup
ipsend rtfree rtget udpcksum udph2net
unmarked macro in mark.h called in: tcpinit.c
upalloc in upalloc.c, pg. ISO



25 Appendix 2 Xinu Functions And Constants Used In The Code

25.1 Introduction

The code throughout this text uses constants, procedures, and functions provided by the Xinu operating system. Many of the functions correspond to Xinu system calls, while others correspond to library functions. Although it is possible to understand the TCP/IP protocol software without knowing the internal details of how these Xinu procedures operate, understanding the service each function provides is essential to a detailed understanding of how TCP/IP operates.

This chapter provides a brief description of all procedures and functions that are not shown in the text. It explains their purpose and the arguments they use. In addition, it lists include files kernel.h, conf.h, and network.h that many procedures include.

In general, Xinu system calls and library routines are "functions" in the sense that they always return a value. However, few system calls are functions in the mathematical sense because almost all have side-effects. Errors usually result in a return value of SYSERR (or, in some cases, specific error codes); procedures that operate without error return OK.

25.2 Alphabetical Listing

The following pages contain a listing, in alphabetical order, of names and arguments for all procedures and functions used by the code that are not otherwise shown in the text. Because the exact details of how these routines operate is unimportant, no distinction is drawn between library procedures and Xinu system calls. The brief explanations are intended to help readers understand the protocol software in the text, and do not describe possible problems or exceptions. Thus, programmers should obtain



more information about the arguments and calling conventions before attempting to write programs that call these routines.

atoi (string)

Extract an integer in ASCII format from string and return it as the function value.

blkcmp (ptr1, ptr2, nbytes)

Compare the nbytes bytes starting at address ptr1 to the nbytes bytes starting at address ptr2, returning zero if the bytes are identical. If the blocks are not equal, find the first byte that differs, and return a negative integer if that byte is less in the first block than the corresponding byte in the second block, and a positive integer otherwise.

blkcopy (toptr, fromptr, nbytes)

Copy nbytes bytes from address fromptr to address toptr.

blkequ (ptr1, ptr2, nbytes)

Compare nbytes bytes starting at locations ptr1 and ptr2, returning TRUE if they are equal, and FALSE otherwise.

chprio (pid, newprio)

Change CPU scheduling priority of process pid to newprio, and return the old priority as the function value.

close (dev)

Close device dev (for TCP, this deletes the connection).

control (dev, func, arg1, arg2)

Control device dev, applying function func with arguments arg1 and arg2. Control operations are device-dependent.

create (caddr, ssize, prio, pn, nargs, arg...)

Create a process to execute code at address caddr, with initial stack size ssize, CPU priority prio, process name pn, nargs arguments, and argument value(s) starting with arg.

deq (indx)

Remove first item from list with index indx, and return it.

disable (ps)

Save processor status word in ps, and disable CPU interrupts.



enq (item, indx, key)

Insert item on the ordered list with index indx, using integer key to choose a position for the item.

fprintf (dev, fmt, value...)

Format value(s) according to format fmt, and send results to device dev.

freebuf (bufptr)

Free buffer at address bufptr, and return to buffer pool.

freemem (memptr, nbytes)

Free nbytes bytes of memory at address memptr.

freeq (indx)

Delete the list with index indx, returning all memory to the free list.

getbuf (poolid)

Get a buffer from buffer pool poolid.

getc (dev)

Read one character from device dev; block until one arrives.

getdmem (nbytes)

Allocate nbytes bytes of memory that can be used for DMA I/O.

getidproni (but, size)

Get up to size bytes from the hardware ID prom, and store in buffer buf.

getmem (nbytes)

Allocate nbytes bytes of memory from the free list, and return a pointer to it.

getpid ()

Return the process id of the currently executing process.

getprio ()

Return the CPU priority of the currently executing process.

gettme (tptr)

Obtain the local time, expressed in seconds past the epoch date (January 1, 1970), and place in the long integer with address tptr.



index (str, ch)

Return the index of the first occurrence of character ch in string str, or zero if ch does not occur in str.

initq ()

Initialize the general-purpose list mechanism at system startup (must precede newq).

kill (pid)

Destroy the process with id pid.

kprintf (fmt, value...)

Convert value(s) to an ascii string according to format fmt and print on the console like printf, but bypass the interrupt system so it can be used to debug the operating system kernel.

mark (ptr)

Causes the kernel to remember that location ptr has been "initialized" and can be tested with function unmarked.

mkpool (bufsiz, numbufs)

Create a buffer pool containing numbufs buffers, each of which is bufsiz bytes long, and return the pool identifier.

mount (prefix, dev, replace)

Add new name prefix to the file namespace, and associate it with device dev and replacement string replace.

nammap (name, newname)

Map a name through the namespace, write the new name in newname, and return the device identifier as the function value.

newq (size, type)

Allocate a new list that can hold up to size nodes, use type to determine whether mutual exclusion for the list is controlled with a semaphore or by disabling interrupts, and return the list index.

open (dev, name, mode)

Open file or object with name using device dev and access mode mode, returning the device descriptor of the new device used to access the object.



panic (message)

Write string message on the console, and halt (abort) the operating system as well as all applications immediately.

pcount (portid)

Return the number of messages currently waiting at port portid, or negative n if n processes are blocked waiting for messages to arrive.

pcreate (count)

Create a new port with space for up to count messages, and return its identifier.

pdelete (portid, dispose)

Delete port with identifier portid, calling procedure dispose to dispose of each message that is waiting.

precede (portid)

Extract the next message from port portid, blocking until one arrives if the port is empty.

printf (fmt, value..)

Convert value(s) to an ASCII string according to format fmt, and write on the console device.

psend (portid, message)

Deposit integer message on port with id portid, blocking until space becomes available in the port.

putc (dev, ch)

Write the single character (byte) ch to device dev.

qsort (darray, n, isize, cmp)

Quicksort n values, each of isize bytes, in array darray, using function cmp to make comparisons between items.

read (dev, buf, len)

Read up to ten bytes of data from device dev and place in buffer buf, returning the number of bytes read. The exact semantics of read depend on the device, but most devices block the caller until data arrives.

receive ()

Block the calling process until a message arrives for that process, and then return



the message,

recvclr ()

Without blocking, return a message if one has arrived for the calling process, or OK otherwise.

recvtim (maxdelay)

Block the calling process until a message arrives or maxdelay tenths of seconds elapse, returning the message or TIMEOUT.

remove (fname, key)

Remove file with name fname, using key as protection key.

rename (file1, file2)

Change the name of filefile1 to file2.

restore (ps)

Restore CPU interrupts to the status saved in ps by disable.

resume (pid)

Resume a previously suspended process with id pid,

rindex (str, ch)

Return index of last occurrence of character ch in string str, or zero if ch does not appear in str.

scount (sid)

Return the current count of semaphore with id sid; counts of negative n mean n processes are blocked on the semaphore.

screate (icount)

Create a new semaphore with initial count icount, and return its id.

sdelete (sid)

Delete the semaphore with id sid, and unblock any processes that may be blocked on it.

seek (dev, pos)

Seek to position pos on device dev; the exact semantics are device dependent.

seqeq (indx)



Search through list with index indx one item at a time, without removing the items; the list only remembers one search position at any instant.

send (pid, msg)

Send message msg to process with identifier pid, but discard the message if the process already has an unread message waiting.

send (pid, msg)

Send message msg to process with identifier pid, overwriting any previously unread message.

set_evec (vec, func)

Assign hardware exception vector vec a pointer to exception handling function func.

setdev (pid, idev, odev)

Set the standard input and output devices for process with id pid to idev and odev, respectively.

setnok (nok, pid)

Set the "next-of-kin" for process with id pid to nok, allowing nok to be notified if process pid terminates.

signal (sid)

Signal semaphore with id sid, allowing a process to continue if any are blocked on the semaphore.

sleep (sdelay)

Delay the calling process rdelay seconds before returning.

sleep10 (tsdelay)

Delay the calling process tsdelay tenths of seconds before returning.

sprintf (str, fmt, value...)

Convert value(s) to an ASCII string according to format fmt, and place results in string str.

streat (tostr, fromstr)

Concatenate a copy of null-terminated string fromstr to the end of null-terminated string tostr.

strcmp (str1, str2)



Compare null-terminated strings str1 and str2, and return an integer less than zero, equal to zero, or greater than zero, to indicate that str1 is lexically less than, equal to, or greater than str2.

strcpy (tostr, fromstr)

Copy the contents of null-terminated string fromstr to null-terminated string tostr.

strlen (str)

Return the length of null-terminated string str measured in bytes, not including the null terminating character.

strncat (tostr, fromstr, maxlen)

Concatenate a copy of null-terminated string fnmstr to the end of null-terminated string tostr, but do not exceed maximum length maxlen bytes.

strncmp (str1, str2, maxlen)

Compare up to maxlen bytes from null-terminated strings str1 and str2, and return an integer less than zero, equal to zero, or greater than zero, to indicate that str1 is lexically less than, equal to, or greater than str2.

strncpy (tostr, fromstr, maxlen)

Copy null-terminated string fromstr to null-terminated string tostr, but do not exceed maximum length maxlen bytes.

suspend (pid)

Suspend (block) the process with id pid.

umount (prefix)

Remove the namespace mapping that has a name prefix equal to prefix.

wall (sid)

Decrement the count of semaphore with id sid, and block the calling process on that semaphore if the resulting count is negative; a process blocked on a semaphore can only continue after another process calls signal for the semaphore.

write (dev, buf, len)

Write len bytes from buffer buf to device dev. The exact semantics of write depend on the device, but most devices block the caller until all bytes can be written.



25.3 Xinu System Include Files

The code throughout this text includes three files from the Xinu system that define symbolic constants, type names, and macros: network.h, kernel.h, and conf.h. File network.h includes all .h files related to networking code. In addition, it defines macros like net2hs that convert values between network byte order and the local host's byte order.

```
/* network.h */

/* All includes needed for the network */

#include <lereg.h>
#include <ip.h>
#include <ether.h>
#include <ipreass.h>
#include <icmp.h>
#include <udp.h>
#include <tcp.h>
#include <tcpfsm.h>
#include <tcpstat.h>
#include <tcb.h>
#include <net.h>
#include <dgram.h>
#include <arp.h>
#include <fserver.h>
#include <rfile.h>
#include <domain.h>
#include <dma.h>
#include <netif.h>
#include <route.h>
#include <rip.h>
#include <daemon.h>
#include <snmpvars.h>

/* Declarations data conversion and checksum routines */

extern short cksum(); /* ls comp of 16-bit 2s comp sum*/

#if BYTE_ORDER == LITTLE_ENDIAN
#define hs2net(x) ((unsigned) ((x)>>8) + (unsigned)((x)<<8))
#define net2hs(x) ((unsigned) ((x)>>8) + (unsigned)((x)<<8))

```



```
#endif

#if BYTE_ORDER == BIG_ENDIAN
#define hs2net(x) (x)
#define net2hs(x) (x)
#define hl2net(x) (x)
#define net2hl(x) (x)
#endif

/* network macros */
#define hi8(x) (unsigned char) (((long) (x) >> 16) & 0x00ff)
#define low16(x) (unsigned short) ((long) (x) & 0xffff)

#define BYTE(x, y) ((x)[(y)]&0xff) /* get byte "y" from ptr "x" */
```

File kernel.h contains declarations used by all operating system functions. It defines values for the return codes SYSERR and OK as well as other constants that appear in the code. It also defines the labels PROCESS and LOCAL used to declare procedures.

```
/* kernel.h - disable, enable, halt, restore, isodd, min, max */

/* Symbolic constants used throughout Xinu */

typedef char Bool; /* Boolean type */
#define FALSE 0 /* Boolean constants */
#define TRUE 1
#define EMPTY (-1) /* an illegal gpg */
#define NULL 0 /* Null pointer for linked lists*/
#define NULLCH '\0' /* The null character */
#define NULLSTR ""
#define SYSCALL WORD /* System call declaration */
#define LOCAL static /* Local procedure declaration */
#define COMMAND int /* Shell command declaration */
#define BUILTIN int /* Shell builtin " " */
#define INTPROC WORD /* Interrupt procedure " */
#define PROCESS WORD /* Process declaration */
#define RESCHYES 1 /* tell ready to reschedule */
#define RESCHNO 0 /* tell ready not to resch. */
#define MININT 0x8000
#define MAXINT 0xffff
#define LOWBYTE 0377 /* mask for low-order 8 bits */
#define HIBYTE 0177400 /* mask for high 8 of 16 bits */
```



```
#define LOW16          01777777      /* mask for low-order 16 bits      */
#define MINSTK          8192        /* minimum process stack size      */
#define NULLSTK          0x1000      /* process 0 stack size          */
#define DISABLE          0x2700
#define MAGIC            0125252     /* unusual value for top of stk  */

/* Universal return constants */

#define OK               1           /* system call ok              */
#define SYSERR           -1          /* system call failed         */
#define EOF              -2          /* End-of-file (usu. from read) */
#define TIMEOUT          -3          /* time out (usu. recvtime)   */
#define INTRMSG          -4          /* keyboard "intr" key pressed */
                           /* (usu. defined as ^B)       */
#define BLOCKERR         -5          /* non-blocking op would block */

/* Initialization constants */

#define INITSTK          0x4000      /* initial process stack size   */
#define INITPRIORITY     20          /* initial process priority    */
#define INITNAME          "main"       /* initial process name        */
#define INITARGS          1,0          /* initial count/arguments   */
#define INITRET           userret      /* processes return address   */
#define INITPS            0x2000      /* initial process PS          */
#define INITREG           0           /* initial register contents   */
#define QUANTUM           10          /* clock ticks until preemption */

/* Machine size definitions */

typedef char CHAR;           /* sizeof the unit the holds a character*/
typedef long WORD;           /* maximum of (int, char *)      */
typedef char *PTR;           /* sizeof a char. or fcnt. pointer */
typedef int INT;             /* sizeof compiler integer      */
typedef long REG;            /* sizeof machine register      */

/* Machine dependent type definitions */

#define SAVEDPS          INT
```



```
#define HINTLEV      8          /* highest autovector level, LL */
#define MAXSER       6          /* max. device per autovector LL*/

/* Machine dependent constants */

#define SSP        15          /* Supervisor stack pointer */
#define PS         16
#define PC         17

#define MAXLONG    0xffffffff
#define MINLONG   0x80000000

typedef short STATWORD[1]; /* saved machine status for disable/restore */

/* by declaring it to be an array, the      */
/* name provides an address so forgotten*/
/* &'s don't become a problem           */

/* Miscellaneous utility inline functions */
#define isodd(x) (01&(WORD)(x))
#define min(a,b) ( (a) < (b) ? (a) : (b) )
#define max(a,b) ( (a) > (b) ? (a) : (b) )

extern int rdyhead, rdtytail;

extern int preempt;

extern int noint;
```

File conf.h defines constants used for a particular Xinu configuration. For example, it defines the size of buffers that TCP uses to send and receive data (TCPSBS and TCPRBS). In addition, conf.h contains extern declarations for the individual functions that comprise device drivers.

Conf.h is unusual because a program generates it automatically from a configuration specification. For example, it defines Ntcp to be 16 because the original input specification defines 16 table slots for TCP connections. The code in the text does not contain constants that define sizes — it refers to the symbolic constants like Ntcp defined in conf.h. Xinu generates a new version of conf.h automatically whenever the original input specification changes. Thus, the support software produces a new version of the system automatically whenever a programmer changes a parameter such as a buffer size.



26 *Bibliography*

ABRAMSQN, Nr [1970], The ALOHA System - Another Alternative for Complex Communications, Proceedings of the fait Jofnr Computer Conference.

ABRAMSON, N. and F. K"JO <EDS.) [1973], Computer Communication Networks, FT&ntiee HalL Englewood Cliffs, NewJer&ey.

ANDREWS. D. W., and G. D. SHULTZ [1982], A Token-Ring Architecture for Local Area Neiworks: An Update. Proceedings of Fatt 82 COMPCON, IEEE,

BALL, J. E., E. J. BURKE,]. CERTNER, K. A. LANTi, and R. F. RASHID [1979), Perspectives on Mess age-Based Disiributcd Computing, IEEE Computing Networking Symposium, 46-5 k

BBN [1981], A Hisiory of the ARPANET: The First Decade, Technical Report Bolt, BerdJiek. and Newman, Inc.

BBN [DecenYber 1981), Specification for the Interconnection of a Host and an IMP (revised), Technical Report 1822. Boll, Bcranek, and Newman, inc.

BERTSEKAS D, and R. GALLAGER J19&7], Dura Nernwks* Prentice-Hall, Englwood Cliffs, New Jersey.

BlftRELU A., and B. NELSON [February 1984], Implementing Remote Procedure Calls, ACM Transactions on Computer Systems, 2(1), 39-59.

BOGGS, D., J. SHOCH, E. TAFT, and R. METCALFE [April 19801, Pup- An Internetwork Archilecture, tEEE Transactions on Commun nations.

BORMAN, D., {April 19-B9-J, Implementing TCP/IP on a Cray Computer, Cumpuwr Communication Review, 19(2). 11-15.

BROWN. M, K. KOLLING, and E. TAFT [November 1985], The Alpine File System, Transactions on Computer Systems. 3(4), 261-293.

BROWNBRIDGE, JD.. L. MARSHALL, and B. RANDLLL [December L982J. The New-castle Connections or UNIXes of the World Unite!, Safin-are - Practice and Expt nence, 12(12). 1147-1162.

CERF. V., and E. CAIN [October 19831- The DOD Internet Architecture Model.



Computer Netwwks.

5S9

Bib-iiograph^

59)

DENNING P. J., {September-October 1989}, The Science of Computing: Wvrdnet, in American Scientist, 432-434.

DENNING P. J., J November-December 1989], The Science of Computing: The ARPANET After Twenty Years, in American Scientist, 530-534.

DIGITAL EQUIPMENT CORPORATION. INTEL CORPORATION, and XEROX CORPORATION JSept-ember 19801, The Ethernet: A Local Area Network Data Link Layer and Physical Layer Specification.

DION, J. [Oct. I9fc0]r The Cambridge File Server. Operating Systems Review, 14(4), 26-35.

DRIVER. K. H. HOPEWELL, and J. JAQUINTO [September 1979]. How the Gateway Regulates Information Con urn I, Data Communications.

EDGE. S. W. f 1979], Comparison of The Hop-by-Hop and Endpoiim Approaches to Network Interconnection, in flow Control in Computer Networks, J-L. GRANGE and M. GIEN (EDS.). North-Hoi land, Amsterdam, 359-373.

EDGE, S. [1983], An Adaptive Tirneout Algorithm for Retransmission Across a Packet Switching Network, Proceedings of ACM SIGCOMM '83.

ENSLOW, P. [January 19781, What isa 'Distributed' Data Processing System? Computer, 13-2]

FALK. G. [1983], The Structure and Function of Network Protocols, in Computer Communications, Volume I: Principles, CHOU, W. (ED.)T Prentice-Halt, En.glcwoo<] Cliffs. New Jersey.

FARMER, W. O.. and E. E. NEWHALL JI969J, An Experiment Distributed Switching System to Handle Bursty Computer Trafficr Proceedings of the ACM Symposium on Probabilistic Optimization of Data Communication Systems. 1-33

FCCSET [November 1987]. A Research and Development Strategy for High Performance Computing, Report pom the Executive Office of the President and Office of Science and Technology Policy,

FEDOft. M. [June 19SS], GATED; A Multi-Routing Protocol Daemon for UNIX. Proceeding* of the 19HH Summer USENIX inference, San Francisco. California.



Ff1NLE&. J,f CI. J. JACOBSEW, and M. STAHL [December 1985], DON Protocol Handbook Volume Two, &ARPA internet Protocol*, DDN Network Information Center, SRI International. 333 Ravenswood Avenue. Room EJ291, Menlo Park., California.

FRANK. H., and W. CI30U [1971], Routing in Computer Networks, Networks, 1(0,99-112.

FRANK, H.. and J, FRI3CH [1971], Communication, Transmission, and Transportation Networks. Addison-Wesley. Reading. Massachusetts.

FHANTA, W. R., and I. CHLAMTAC [19K1J, Local Networks, Lexington Books, Lexington. Massachusetts.

FRJCC [May 1989]. Program Pfon for the National Research and Education Network, Federal Research Internet Coordinating Committee, US Department of Energy. Office of Scientific Computing report ER-7.

FHIDRJCH. M.. and W. OLDER [December 1981], The Felix File Server. Proceedings af Jhe Eighth Symposium on Operating Systems Principles, 37-16.

592

BibFiography

FULTZ, G. L.,, and L. KLEINHOCK, (June 14-16, 1971], Adaptive Routing Techmqu.es for Store-and-Forward Computer Communication Networks, presented at IEEE International Conference Qtt Communications, Montreal. Canada

OERLA, M., and L. KLEINHOCK [April 1980J, How Control: A Comparative Survey, IEEE Transactions on Commonjcation.

GOSIP (April 19S9], US. Government Open Systems inierconnection Profile (GOSIP) version 2.0, GOSIP Advanced Requirements Group, National Institute of Standards and Technology (NIST).

GRANGE, J-L., and M, GIEN (EDS.> [1979], Flow Control in Computer Networks, North-Holland, Amsterdam,

GREEN, P.-E. (ED.) [1982], Computer Network Architectures and Protocols, Plenum fVess, New York.

KINUEN, R, J. HAVEftTY, and A. SHELTZER [September 1983J, The DARPA Internet: Interconnecting Heterogeneous Computer Networks with Gateways. Computer.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (June 1986aJ, Information processing systems — Open Systems Interconnection — Transport Service



Definition. International Standard number 8072, ISO, Switzerland.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION [July 1986b], Information processing systems -- Open Systems Interconnection — Connection Oriented Transport Protocol Specification. International Standard number 8073, ISO, Switzerland,

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION [May 1987 a], Information processing systems — Open Systems Interconnection — Specification of Basil. Specification of Abstract Syntax Notation One (ASN.I) International Standard number 8824, ISO, Switzerland.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION [May 1987b], Information processing systems — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.I). International Standard number 8825, ISO, Switzerland.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION [May 1988a], Information processing systems — Open Systems Interconnection — Management information Service Definition. Part 2: Common Management Information Service, Draft International Standard number 9595-2, ISO, Switzerland.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (May 1988a), Information processing systems — Open Systems Interconnection — Management Information Protocol Definition, Part 2: Common Management information Protocol, Draft International Standard number 9596-2.

JACOBSON, V. [August 1988], Congestion Avoidance and Control Proceedings ACM Sigcomtn '88.

JAIN, R. [January 19851, On Caching Out-of-Order Packets in Window Row Controlled Networks, Technical Report DEC-TR-342, Digital Equipment Corporation.

JAIN, R. [March 1986^ Divergence of Timeout Algorithms for Packet Retransmissions, Proceedings Fifth Annual International Phoenix Conference on Computers and Communications. Scottsdale, AZ.

Bibliography

593

JAIN, R. [October 1986], A Timeout-Based Congestion Control Scheme for Window Row-Controlled Networks, IEEE Journal on Selected Areas in Communications. Vol. SAC-4, no. 7,

JAIN, R., K. RAMAKRISHNAN, and D-M. CHIU [August 1987], Congestion



Avoidance in Computer Networks With a Connectionless Network Layer. Technical Report, DEC-TR-506, Digital Equipment Corporation.

JENNINGS D, M., L H. LANDWEBER, and I. H. FUCHS [February 28, 1986], Computer Networking For Scientists and Engineers, Science vol 231, 941-950.

JUBIN, J. and J. TORNOW [January 1987], The DARPA Packet Radio Network Protocols, IEEE Proceedings.

KAHN, R. [November 1972], Resource-Sharing Computer Communications Network;, Proceeding* of the iEEt,6Q{\l}, 1397-1407.

KARN, P., R PRICE, and R. DiERSING [May *9B5], Packet Radio in the Amateur Service, IEEE Journal on Selected Areas in Communications,

KARN, P., and C. PARTRIDGE [August 19S7]. Improving Round-Trip Time Estimates in Reliable

Transport Protocols, Proceeding* of ACM SIGCOMM 'S7.

KENT, C, and J. MOGUL jAugust 19S7L Fragmentation Considered Harmful, Proceedings of \CM SIGCOMM '&?,

KUKE, C (August 1987], Supercomputers on the Internet: A Case Study, Proceedings of ACAf SIGCOMM 'R7.

KOCHAN, S. G., and P. H. WOODS [1989], UNIX Wetworkinx, Harden Books, Indianapolis, IN.

LABARRE. L, (ED.) [December 1989], OSI Internet Management: Management Information Base, Internet Draft <FETF.DRAFTS>DRAFT-JETF-SWMP-ATIB2-0I.TXTm DON Network Information Center, SRI International, Ravenswood, CA.

LAMPSON, B. W, M. PAUL, and H, J, SIEGERT (EDS.) [1981], Distributed Systems -Architecture and Implementation (An Advanced Course), Springer- Verlag, Berlin.

LANZILLO, A. U and C PARTRIDGE [January 1989], Implementation of Dial-up IP foT UNIX Systems, Proceedings 1989 Winter VSENIX Technical Conference. San Diego, CA,

LAQUEY, T. L., [July 1989]. User's Directory of Computer Networks. Digital Press, Bedford, MA.

LAZAR, A. (November 1983], Optimal Flow Control of & Class of Queuing Networks in Equilibrium. IEEE Transactions on Automatic Control* Vol. AC-2S:11.

LBFFLER, S., M, McKUSICK, M, KARELS. and J. QUARTERMAN [19891, The Design and Implementation of the 43BSD UNIX Operating System, Addison Wesley, 198-9.

LYNCH, D. C,, and O, J- JACOBSEN (PUBLISHER and EDITOR) [19S7-J,



Connexions, the Interoperability Repon, Fnttrop Incorporated,, 480 San Antonio Rd, Suite 100, Mountain View, California.

LYNCH, D. C. (PRESIDENT) [19S7-], The Annual Interop Conference Interop Incorporated, 480 San Antonio Rd, Suiie [00, Mountain Vie*, California,

MCNAMARA, J. [19&2]h Technical Aspects of Data Communications, Digiial Press, Digital Equipment Corporation, Bedford. Massachusetts.

5-94 Bibliography

MCQUILLAN, J. M., I. RICHER, and E. ROSEN [May 1930], The New Routing Algonthm for the ARPANET, IEEE Transactions on Communtcatums, (COM-28), 711-719.

MERIT [November 1987], Management ajtd Operation of the NSFNET Backbone Network: A Proposal Funded by the National Science Foundation and the State of Michigan, MERIT Incorporated, Ann Arbor h Michigan.

METCALFE, R. M, and D. R. BOGGS [July 1976], Ethernet: Distributed Packet Switching for Local Computer Networks, Communications of the ACM, 19(7), 395-404.

MILLER, C. K., and D. M. THOMPSON [March 1982J, Making a Case for Token Passing in Local Networks* Data Communications.

MILLS, D.p and H-W. BRAUN [August 19S7], The NSFNET Backbone Netwoik, Proceeding of ACM SIGCOMM '87.

MITCHELL, J. and J. DION [April 19&2], A Comparison of Two Network-Based File Servers, Communications of the ACM, 25(4), 233-245.

MORRIS, R- [1979], Fixing Timeout Intervals for Lost Packet Detection in Computer Communication Networks, Proceedings AFIPS National Computer Conference, AFIPS Press, Montvale, New Jersey-

NACiLE, J. [April 19871, On Packet Switches With Infinite Storage, IEEE Transactions on Communications, Vol. COM-35:4.

NARTEN, T tSept, 19B9], Interne! Routing, Proceedings ACM SfGCOMM '89.

NEEDHAM, R. KL [19791, System Aspects of the Cambridge Ring, Proceedings of the ACM Seventh Symposium on Operating System Principles, 82-85.

NELSON, J. [September 1983], 802; A Progress Report, Datamation.

OPPEN, D., and Y. DALAL [October 19811, The Clearinghouse: A Decentralized Agent for Locating Named Objects, Office Products Division, XEROX Corporation.

PARTRIDGE, C [June 1986], Mail Routing Using Domain Names.: An Informal Tour, Proceedings of the 1986 Summer USENIX Ctinfewnce. Atlanta, GA.

PARTRIDGE, C. (June 1987], Implementing the Reliable Daia Protocol (RDP),



Proceedings of the J987 Summer USEfflx Conference, Phoenix, Arizona.

PETERSON, L. [1985], Defining and Namrng the Fundamental Objects in a Distributed Message System, Ph,D, Dissertation, Purdue University, West Lafayette, Indiana.

PIERCE, J. R. [19721, Networks for Block Switching of Data, Beit System Technical Journal, 51.

POSTEL, J. B. [April 1980], Internetwork Protocol Approaches, IEEE Transactions on Communications. COM-28. 604-611.

POSTEL, J. B., C. A. SUNSHINE, and D. CHEN [1981], The ARPA Internet Protocol, Computer Networks.

QUARTERMAN, J. S. [1990], The Matrix: Computer Networks and Conferencing Systems Worldwide, Digital Press, Digital Equipment Corporation, Maynard, MA,

QUARTERMAN, J. S., and J. C. HOSKINS [October 19861. Notable Computer Networks, Communications of the ACM, 29(10).

Bibliography

59\$

REYNOLDS, J., i POSTEL, A. R. KATZh G. G. FINN, and A. L DESCHON [October 1985], The

DARPA Experimental Multimedia Mall System, IEEE Computer.

RITCHIE, D. M. (October 19841, A Stream Input-Output System, AT&T Bell Laboratories Technical Journal 63(8), I987-1911X

RITCHIE, D. M., and X. THOMPSON [July 1974], The UNIX Time-Sharing System, Communications of the ACM, 17(7), 365-375; revised and reprinted in Beit System Technical Journal, 57(6). [Juty^Attgust 1973], 1905-1929,

ROSE, M, (ED,) [October 1989], Management Information Base for Network Management of TCPyiP-based Internets, internet Draft <IETF.DRAFTS>DRAFT-IETF-OIM-Mf82-00.TXT1 DDN Network Information Center, SRI International, Ravenswood, CA.

ROSENTHAL, R. (ED.) [November 1982], The Selection of Local Area Computes Networks, National Bureau of Standards Special Publication 500-96,

SALTZER, J, [J978], Naming and Binding of Objects, Operating Systems, An Advanced Course, Springer-Verlag, 99-208.

SELTZER, J. [April 1982], Naming and. Binding of Neiwork Destinations,



International Symposium on Local Computer Networks. [FHVT.C.6* 311-317.

SALTZER. J., O. REED, and D. CLARK [November 1984], End-to-End Arguments in System Design, ACM Transactions on Computer System?, 2(4), 277-285

SCHWARTZ, M., and T. STERN JApnL 1980S. /fff Transactions on Communications, COM-28(4), 539-552.

SHOCH, J. P. [19781, Internetwork Naming, Addressing, and Pouring, Proeedings j>f COFFIPCON.

SROCU, J. F. y. DALALH and D. REJ>ELL [August 1982], Evolution of the Ethernet Local Computer Network, Computer.

SNA [19751, /SW System Network Architecture - General Information, IBM System Development Division, Publications Center, Department #01, p.Q. Box 12195, Research Triangle Pari, North Carolina, 27709,

SOLOMON. M, L, LANDWEBER, and D. NEUHEGEN [1982J. The CSNET Name Server, Computer Networks {6). 161-172,

STALLINGS, W. [19851, Locpt Networks: An Introduction, Macmillan Publishing Company, New York.

STALLINGS, W. [19851, Data and Computer Communications. Matmitian Publishing Company, New York,

STEVENS, Wh R. {1990], UNIX Network Programming, Prentice-Hall, Englewood Cliffs, New Jersey.

SWINEHART. D, G. MCDAN1EL, and D. R. BOGGS [December 19791, WFS: A Simple Shared File System for a Distributed Environment, Proceedings of the Seventh Symposium on Operating System Principles. 9-17

TANENBAUM, A, [1981], Computer Networks: Toward Distributed Processing Systems. Prentice-Hall, Englewood Cliffs, New Jeriey.

59t

Bibliography

TICHY, W., and Z, RUAN [June 1984], Towards a Distributed File System, Proceedings of

Sarrowr 64 USENIX Conference, Salt Lake City, Utah, S7-97.

TOMUNSON. R. S. (1975], Selecting Sequence Numbers, Proceedings ACM SIGOPS/SIGCOMM Interprocess Communication Workshop, 11-23, 1975.

WARD, A-A. [IQSQJ.TRIX: A Network-Oriented Operating System, Proceedings



of COMPCON, 344-349.

WATSON, R. [1981]. Timer-Basftd Mechanisms in Reliable Transport Protocol Connection Management, Computer Networks, North-Holland Publishing Company,

WEINBERGER, P. J. [1985], The UNIX Eighth Edition Network File System, Proceedings 19SS ACM Computer Science Conference. 299-301.

WELCH, B.# and J, OSTERHAUT [May 1986], Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System, Proceedings IEEE Sixth International Conference on Distributed Computing Systems, 1845-189.

WILKES, M. V., and D, J. WHEELER [May 1979], The Cambridge Digital Communication Ring,

Proceedings Local Area Computer Network Symposium.

XEROX [1981], Internet Transport Protocols. Report XSIS 023112. Xerox Corporation, Office Products Division, Network Systems Administration Office, 3333 Coyote Hill Road, Palo Alto* California.

ZHANG, L. [August 1986], Why TCP Timeis Don't Work Well, Proceedings of ACM SIGCQMM-•86.



27 Index

arpinil 554
arpLnii in arpinit.c 5B
arpprint 554
arpqsend 554
arpqsend in arpqsend.c 50
arpsend 554
arpsend in arpsend.c 46
arplimer 554
arptimer in arptimer.c 56
arp-'f- 554
arp_in in arpjn.c 51
ascdate 554
ASN.I 441, 474, 477, 491
conversion 500
converting integers 467
converting object ids 469
converting to internal form 494
converting values 472
lengths 465
lexical order 445
numeric representation 444
object identifier 44]
object identifier conversion 484



representations 464 asnl.h 464 a*oi 574

B

Berkeley broadcast 74

Berkeley urgent 313

bfkcmp 574

blkcopy 374

blkequ 574

blocked &

BSDURG 313

BTOP 554

bucket hash 451

bucket hashing 86

buffer management 32

BYTE 554

BYTE in network.*! 5%\$

byte 14

byte order conversion 76, 181,207

C

call graph 553

cascade of updates 360

checksum 208

chprio 574

cksirm 554

cksum in cksum.c 72

client

SNMP 44L close 324, 574

CLOSE-WAIT state 218,225,242,344 CLOSED state 203, 117, 240 CLOSING

state 221 community 483 conf.h 584

congestion collapse 295 congestion window 295 connection

endpoint 209 connectionless 361 consumer 11 control 574 count to infinity 357

counting semaphore 9 create 8, 574 cumulative acknowledgement 284

D



daia mark 312
DBHASH 555
db_init 555
dbjookup 555
db_ne* 555
db.nlink 555
db_resync 555
db_rlink 555
db_update 555
ddi_exchng 555
ddi_e*chng in ddiexchng.c 429
ddLeAstart 555
ddi_exstart in ddi_exstan.c 428
ddi_full 555
ddi_fullindfiji*tf.r 431

Index

599

dd_queue 555
dd_queueinddqueue.c 421
dd_xmit 555
dd_xmii in ddxmit.c 424
default rouCe S8
DELAY 555
DELETE message 249
deletion event 219
delta list 269
demultiplexing 209



Department of Defense AAA

dcq 575

designated router 399

destination unreachable 58, 184

device 14

TCP master 325

TCP slave 325 device abstraction 323 device driver 14, 27 device paradigm 323 dgram.h 178 direct memory access 34 disable 575 disable in kerneLh 582 DMA 34 dnparse 555 dn_cat 555 do not fragment 125 dol2ip 555 dsdirec 555 dssync 555

E

echo reply 132 echo request 132 echod 555 ECHOMAX 555 echop 555 efacein.it 555

egp 555

end of options 294

endpcint 209

enq 575

entity agent 446

error index 480 error type code 480 ESTABLISHED state 200, 237, 2425

242, 248, 344 ethcrttl 555+ ethdemux 555 ethinit 556 ctbint 556 ethinter 556 ethmcast 556

ethmcast'm ethmcast.c 154 ethread 556 ethstn 556 ethwrite 556 ethwstrt 556 EVENT 556

F

fclientc 326

fd_to_dd 556

fgetc 556

FIN 218, 221, 223, 223, 225, 225, 2275

227, 229, 233, 236, 242, 302, 344 FIN-WAIT-1 state 228, 223, 344 FINAVAIT-2 state 218.222 findfd 556 finger 326, 556 finger \n fclient .c 326 finger server 327 fingerd 556 fingerd in fserver.c 328 finite state machine

implementation 204 firstid 556 firstkey 556 foper 556 fprintf 575 fputc 556 fragment 233 freebuf 575 freemem 575 freq 575 frEestk 556 fserver.c 328



GOO

Index

gather-write 34
gel 480
get-fir&t 487
get-nest 4S0 *
get-next-request 446,447,475
get-request 446
geiaddr 556
getbuf 575
getc 575
getcbar 556
geidmem 575
geliaddr 556
getidpram 575
getiname 556
getinei 556
getmern 575
getmib 556
getmib in snhash.c 459
getname 556
get net 556
GETPHYS 556
getpid 575
getprio 575
getsim 557
gettme 575
getutim 557



get_bit 556
getjiashbit 556
gname 557
gratuitous acknowledgement 289
gratuitous response 36/
H
bashing 86, 179, 451
hashcnir 557
hashinit in snhash.c 459
bashoid 557
hashoid in snhash.c 459
Hello interval 397
Hello protocol 392
hgadd 557
hgadd in hgadd.c 151

hgarpadd 557
hgarpadd in hgarpadd.c 153
bgaipdel 557
hgaipdel in hgarpdeLc 156
hginit 557
hginit in hginit.c 167
hgjoin 557
hgjoin in hgjoin.c 157
hg leave 557
bgleave in hgleave.c 166
hglookup 557
hgJoo-kup in hglookup.c 150
hgprint 557
hgrand 557
hgrand in hgtand.c 162
hi8 557



hi8 in netw&rk-h 580

hi2net 557

h!2net in network.h 580

hole 3L2

host group 147

host redirect 145

hs2net 76, 557

hs2net in tietwork.H 580

I

ibdisp 557

ibtodb 557

icerrok 557

icen-ok in icerrok.c 339

ICMP

destination unreachable 58, 66, 68

host redirect 69

network redirect 69

redirect 66, 69

time exceeded 69

time-to 4 ive 69 icmp 68, 557 icmp i.n icntp.c 137 ICMP echo reply 143 ICMP echo request (43 ICMP host redirect 145 JCMP netwoik redirect 145 ICMP redirect 144

Ind**

ICMP time exceeded 123

icmp.c 137

icmp.h J 28

icmp_in 558

icmp_Jn in icmpjn.c 130

icredirect 558

icredirect in icredirectc 132

icsstbuf 558



icse_ibuf in icsethuf.c 140
icsetdaa 558
icsetdata in iesetdata.c 142
icsetsrc 55ft
icsetsrc in icsetsrc.c 136
IDLE state 252
idle scate 248T 304, 305
if_elect 55&
if_elect in ifetea.c 402
if_electl 558
if_electf in ifelect J c 4O0
IGMP J58
igmp 558
igmp in igtnp.c 159
igmp.h 148
ignvpJn 558
igmp^in migmpjn.c 164
igmp_se (timers 55S
igmp_settimers in igmp settimers.c
igmp_update 55R
igmp_update in igtnp update.c 163
IGJTYP 558
IG_TYP in igmp.c 159
IG_TYP in igmp.it]48
IG_VER 558
IG_VER in igmp.v 159
1GJVER in igmp.h 148
index 576
infinity 357
initgate 558
initbos* 558
initq 576



Input 175
interface state 399
interface structure 28
Internet Control Message Protocol 127

Internet Group Management Protocol J 47, 158 internetwork

see internet interrupt 14

interruption 309

IP 61

PROTO field 61 options 104 software design 62

ip.h 70

iP2dot 55\$

rp2name 558

ipAdEnfAddr 504

ipAdEntBcastAddr 504

ipAdEntIfIndex 504

ipAdEntMask 504

ipdbc 559

ipdbc in ipdbc.c 73

ipdoopts 559

ipdoopts in ipdoopl.c 104

ipdstopts 559

ipdstopts in ipdstopts.c 104

ipfadd 559

ipFadd in ipjadd.c 11-8

ipfcons 559

ipfcons in ipfcorts^c 121

ipfheopy 559

jpfheopy in ipfhcapy.c i 12

ipfinil 559

zpfinit m ipfmit.c 124

ipfjoin 559



ipfjoin in ipfjoin.c 119
ipfsend 559
ipfsend in tpfsend-v HO
ipftiraer 559
ipftimer in ipftimer.c 122
ipgetp 559
ipgetpin ipgelp.c 64
ipti2nei 559
iph2nei in iph2ner.c 76
ipnet2li 559
ipnet2h in ipnetlh.c 76
ipproc 559

602

Index

ipproc in ipproc.c 66
ipputp 559
ipputp in ippuiip.c 10S
ipreass 559
ipreass in ipreass.c 116
ipreass.h 114
ipredirect 559
ipredirect in ipredirectx
ipsend 559
ipsend in ipsend.c 78
IP^CLASSA 558
IP_CLAS5A in ip.h 70
IP_CLAS5B 55S



IP_CLASSB in rp.h 70
IP_CLASSC 558
IP_CLASSC in ip.h 70
1P_CLASSD 558
IP_CLASSD in iph 10
IP_CLASSE 559
1P_CLASSE in ip.h 70
IP_HLEN 559
IP_HLEN in ip.h 70
ip_Ln 559
ip_in in iptH.c 80
isalnum 560
is-alpha 560
isascii 560
Lsbaddev 560
isbadpid 560
isbadport 560
isbadserci 550
isbrc 560
isbrc in isbrc.c 15
iscntrl 560
isdigit 560
isemply 560
isleap 560
i slower 560
ISO 444
isodd 560
isodd in kernel.h 582
i sprint 560
isprshort 560
ispunct 560



144

isspace 560 I5UOPT 560

is upper 560 1SVALID 560 isxdigit 560

K

k-out-of-n 361 Karn's algorithm 284 KEEPALIVE message 249 kernel .h 582 kill 8, 576 kprintf 576

large buffers 33

LAST-ACK state 21S, 227, 344

lasAey 560

lajer

network interface 27 length encoding 465 level 560 lexan 560 lexical order 445 lexicographic ordering 445 lfiig 560 listen queue 242 LISTEN state 200, 209, 237, 243, 344,

346 local ring broadcast 60 local_out 560 login 560 long 76 fowl6 560

lowl 6 in neiwork.h 580 lsack_in 560 lsa_add 560 1sa_send 560 Isajtmit 561 lsr_a<id 561 lsr_criek 561 lsr_in 561 Isr in in Isr in.c 432

Indei

603

isr__qcue 561

Isr^xmit 561

LSS_ABORT 561

LssJmiJtl 561

lss_bui]d in hsbuuid.c 434

lsu_in 561

Jtim2ut 561

M

macroscopic state 200



major 561
makedev 561
Management Information Ba&e 441
mark 576
marked 561
master 394
master device 325
max 561
max sn kemel.h 5\$2
maximum datagram size 33
maximum segment lifetime 219
maximum segment size 245, 291,291
maximum transfer unit 30
mbuf 33
message passing 9
message-driven 248
MIB 37, 441,443,474,477
hash tabJe 450, 452
hierarchy 447
name prefix 445
names for tables 446
numeric name 44+
object identifier 441
simple variables 443
tables 443
threaded tree 447
variabJe names 444
variable simulation 443 MIB variables 30 MIB-H 546 mib.h 448
microscopic state 200 min 561 min in kernei.h 582

minor 561
mkaip 561



MKEVENT 276, 561
MKEVENT in tcpfsm.h 200
mkpool 576
mksnmp 561
Tnksnmp.c 492
mode 373
modulus 162
more fragments 107, 108, 110
mount 576
MOVC 561
MOVL 561
MOVSB 561
MOVSL 561
MOVSW 561
MSS 245, 291
MSS option 291
MTU 34
multiaccess network 383
multiplicative decrease 295
must be zero 366
mutual exclusion 10
N
name2ip 561
naming authority 444
nam map 576
nb_add 561
nb__add in nhadd.c 408
nb_aok 561
nb_aok in nbjiok.c 414
nb_clearl 561
nb_clearl in nh_ckpt.c 405
nb_makel 562



nb_makel in nbtnakei.c 406
nb_mismatch 562
nb_misrnaTcti in rtb_mismaich.c 438
nb,, reform 562
nb_reform in nb_reform.c 404
nb_rexmt 562
nb_rexrnt in nb_rexmi.c 414
nb___ switch 562
nb switching switchx 410

vl

f] ipi ■»!

MM
net2hl 562
net2hl in network.h 580
net2h& 76, 562
net2h& in network.h 580
net2xt 562
netdump 562
netif 30
netif-h 28
netmask 562
netmask \wneimask.c 92
netmaich 562
netmatch in netmatchx 91
nefcnum 562
netnum in netnum.c 90
netstan 562



network byte order ^6, 181,207,259
network interface abstraction 28
network interface layer 27
network interface table 521
network redirect 145
network.h 580
net write 562
netwrite in newite.c 47
newq 576
next-hop address 86
nif 28
N1GET 30, 562
N1GET in netifh 28
ni_in 562
ni_in in nijn.c 35
no-operation 294
non-se Preferential 604
nonempty 562
NULL 483, 487
N_BADMAG 561
N_BSSADDR 562
N_DATADDR 562
N_PAGS1Z 562
N_SEGSIZ 562
N_STROFF 562
N_SYMOFF 562
N_TXTADDR 562
N_TXTOFF 562

Index

O

object identifier 44/



Octet 14

Ofaccinit 563

oidequ 563

oidequ in snmp.h 461

open 237, 242,323,576

open-close semaphore 266

open-read-write-close 323

operating system 7

options

IP 104 OSPF 3S1 ospf 563 ospf in ospf.c 416 ospf process in ospfinc 416 ospf.c 416 ospf.h 389 ospfcheck 563 ospfcheck in ospfchccck.c 436 ospfddtmp 563 ospfddtmp in ospfddtmp.c 423 ospfheLlo 563 ospfheELo in ospjheilo.c 398 ospfhtmpl 563 ospfhtmpl in ospjhtmpi.c 396 ospftfinit 563 ospfinii 563 ospflsrtmpl 563 ospfistmpl 563 o\$pfnet2b 563 ospfnet2h in ospfnetlh.c 436 ospfiimer 563 ospftimer ill ospftimer.c 412 ospf_ddin 563 ospfddin in ospfddin.c 426 ospf_hin 563 ospf_hin in ospf_hin.c 406 ospf_hsend 563 ospf_hsend in ospfjstndc 394 cspfjf.h 388 cspf_in 563 ospf_nyc 416 ospfjs.h 418

Indc*

ospf_pkth 386 othinit 563 othwrite 563 out-of-band data 311 out-of-band notification 309

P

page alignment 34

panic 576

passive 313

passive mode RIP 356

passive open 200, 205,217, 237

pcurrnt 12, 577

pcreate 12, 577

pdelece 577

PERSIST message 249

PERSIST state 252

ping 37



poison reverse 359
port 9, 12
preceive J2, 577
pritiicsrO 563
primf 577
priruone 563
priority 9
procedure cross-reference 573
Procedure-Driven 204 procedure-driven 204
process 8
TCP input 207
process identifier 8
Processing
producer 11
program 8
program abort 309
program interruption 309
PROTQ field 61
protocol
ARP 39 protocol port number 171 psend 12, 577 pseudo-header 171 pseudo-network interface 31 ptcpumode 339

PTOB 563
public 483 push 2
push bit 233, 318 push request 318 putc 577 putchar 563
Q
qsort 577 query
resolution 480
R
rarpsend 563
rarp_in 563
read 313, 324,577



read semaphore 339
READERS 265
receive 13, 577
receive urgent pointer 231
receiver-side silly window 287
record 443
recvclr 577
recvtim 13, 577
redirect message 132
relative time 269
remove 578
rename 57-8
request
 RIP 361 request id 494 RESET 261, 302,304 resolution of query 480 resolve 563
response
 RIP 361 restore 578 restore in kerneLh 582 resume S, 578 retransmission
 maximum co-um 286 RETRANSMIT 305

id

/-•>•***- — -

606

Index

RETRANSMIT message 249 RETRANSMIT state 300 retransmit state 286T 304, 305 rindex 578 RIP 355

active mode 356



message size 361
passive mode 356
poison 359
reliability 361
request 363
response 361
update cascade 360
well-known port 362 rip 564
RIP metric 376 Tip.h 363 ripadd 564
ripadcl in ripaddir 374 ripcheck 564 ripcheck in ripcheckx 366 ripifset 564
ripifset in ripifsetx 373 ripin in ripinx 365 ripmetric 564
ripmemc in hpawtiicv 376 ripok 564
ripok in ripokx 370 ripoui 564
ripom in ripoutx 378 riprecv 564
riprecv in riprecvx 368 riprep 564
riprep] in riprepix 370 npsend 564 ripsend in ripsendx 372 ripstan 564
ripstart in ripstan.c 377 round-robin 64 roundsw 564 roundnib 564 route metric 376
route propagation 355, 381 route .h 87

routing 85
Routing Information Protocol 355
routing loop 357
routing protocol 355,381
rtadd 564
nadd in rtaddir 98
rtdel 564
rtdel in rtdelex 102
rtdump 564
RTFREE 564
rtfree 564
rtfree in rtfreex 103
rtget 564



rtget in rtgeSx 94
rehash 564
rthash in rthash.c 93
rtinit 564
rtinit in rtimtx 96
rtnew 564
rtnew in rtnew x 101
rtimer 564
mimer in rttimerx: 96
rwho 565
rwhod 565
rwhoind 565
S
sae_findn,c 510
sae_findneKt 565
sae_findnest tn saejindnx 510
sae_get 565
sae_get.c 506
sae_getf 565
sae_getf.c 508
sae_getn 565
sae_getn,c 509
sae_match 565
sae_match.c 505
sae_sel 565
sae_set.c 511
sat cm p 565
satjldn.c 518
sat_Findne>u 565
sat_gei 565

Inde*



sat_get.c 514 sat_getf 565 sat_getf.c 515 sat_gein 565 sat_getn,c 516 sat_ma<ch 565 5at_match,c 512 sa*_se* 565 sat^set.c 52C scatter-read 34 scount 578 screate 9, 578 sdelete 578 SECYEAR 565 seek 578 seqq 578

self-identifying frame 15 semaphore

read 329 send 13, 578 SEND message 249 send window 255

sender-side silly window avoidance 305 sendf 578 SEQCMP 565 sequence 483 sequence numbers §98 sequence space 198

hole 312 sequence values 198 server

SNMP 441 set 48a set-request 446 setdev 579 seterr 565 seimask 565 setmask in setmask.t; 134 seinok 579 set_bil 565 set_evec 578 SHA 565 SHA in arp.Ji 42 shell 565

short 76 s_if_get 565 sif_getc 522 sif_getf 566 sjf_getf.c 526 sif_getn 566 sif_fecn,c 527 sif_maich 566 sif^match.c 522 sifjet 566 sif_set.c 528 signal 10, 579 silly window avoidance 287 silly window syndrome 287 Simple Network Management Protocol 441 sip2ocpy 566 sizeof 566 slave device 325 sleep 579 sleep 10 579 slow-start 296 slowtimer 566 slowtimer in stowtimerx 81 SMI 475 sna2b 566 sna2b_c 484 snb2a 566 snb2a.c 494 sndient 566 sndieni.c 497 snerr 566 snfreeb] 566 snfreeM.c 496 snhash.c 459 snhash.h 452 sninit 5-66 Sninit iri sninit.c 499 snleaf 566 snleaf.c 48B snmib-c 452 SNMP 37 MIB 441 NULL 4S7

60S

Index

client 441, 477, 496

community field 483

entity agent 446

error index 480

error type code 480

functions 487, 490

get-first 4S7

get-nett 487



get-next-request 446, 447
gel-request 446
hash table 450, 452
identifier conversion 484
indirect calls 487, 490
initialization 498
interpreting get-nexi 48/
lexical order 445
message parsing 480
name hierarchy 447
name prefix 443 object identifier 441 object size 465 operations 446, 450, 480, 487
query resolution 480, 485 reply generation 491 request descriptor 483 request id 494
retransmission 498 server 441, 477 server implementation 47\$ server organization 477
set-request 446 software organization 441 tables 490 threaded tree 447 timeout 49S trap
483
version number 483 SNMP tables 443 SNMP variables 443 snmp.h 461 snmpd 566
snmpd in snmpti.t\ 478 snmpprint 566 snmppri nt_o bj id 566

snmpprint__objname 566 snmppri nt_v.al 566 snmpvars.h 458 snparse 566
snparsex 480 snrslv 566 snrslv.c 485 sntabte 567 sntable.c 490 sntcpemp 567 so2ipcpy
567 software interrupt 14 soipequ 567 SPA 567 SPA in arp.h 42 split horizon 358
spoofing 60 sprimf 579 srt_fmdn.c 535 srt_findnext 567 srt_get 567 srt_getx 530
srt_getf 567 srt_gelf.c 532 srt_getn 567 srt_gein.c 534 srtmatch 567 srt_match,c 529
srt_set 567 srt_set.c 536 standard error 324 standard input 324 standard output 324 state

CLOSE-WAIT 218,225,242,344 CLOSED 203, 217,240 CLOSING 221
ESTABLISHED 200, 237, 242, 242,

248, 344 FIN-WAIT-1 21S, 223, 344 FIN-WAIT-2 21ft, 222 IDLE 252

LAST-ACK 218.227,344 LISTEN 200, 209, 237. 243, 344. 346

Irtdex

609



PERSrST 252

RETRANSMIT 300

SYN-RECEIVED 2005 237, 240, 244, 302

SYN-SENT 200, 237,239

TtME^WAIT 219

TRANSMIT 253, 300

idle 248, 305

retransmit 286, 304, 305

transmit 248, 304, 305 stc_findn,c 344 stc_findneAi 567 stc_findnext in stcjtnfd/t,c
544 stc_get 567

stc_gel in stcj\$etx 540 stc_getf 567 stc_getf in stcgetfx 541 stc_getri 567

stc_gefn in stc_getn.c 542 stc_match 567 stc_match in stc_match.c 538 s(c_ser 567
stc_set in stcset.c 545 sweat 579 stremp 579 strcpy 579 stream paradigm 309
STREAMS 15 strequ 567 strequ in snmp.h 461 sCrien 579 stmcat 579 stmemp 579
semepy 5-80 structure 443 suspend 580 suspended 8 SVtNT 567 SVIPADDR 567
SVOID 567 SVOLDLEN 567 SVSTR 567 SVSTRLEN 568 SVTYPE 568

SYN 200, 210, 220, 222, 223, 225, 226, 227, 229, 231, 239, 240t 240, 242,
244,291,293,294,302, 302

SYN-RECEIVED state 200, 237, 240, 244,302

SYN-SENT state 2Q0t 237, 239

system call 573

table-driven 204 tabteb in snmib.c 452 task 3 TCB 192, 568

deletion 219 tcb.h 192 tcballoc 568 tcballoc in tcballocx 202 tcbdealloc 568

tcbdealloc in tcbdeaitoc.c 20J TCBFJtDONE 266 TCBFJtUPOK 266 TCP

CLOSE-WAIT state 218,225,242, 344

CLOSED state 203,217,240

CLOSING state 221

ESTABLISHED state 200, 237, 242, 248, 344

FIN WATT-1 state 218, 223, 344

F1N-WAIT-2 state 218, 222

IDLE state 252

LAST-ACK. state 218, 227, 344

LISTEN state 200, 209, 237, 243, 344,346



MSS option 291
PERSIST state 252
READERS 265
RETRANSMIT state 300
SYN-RECEIVED state 200, 237, 240, 244, 302
SYN-SENT stare 200, 237, 239
TIME-WAIT state 219
TRANSMIT state 253, 289, 300
WRITERS 265

610

kidea

checksum 208
fragment 233
idle state 248
input 205
macroscopic state 200
microscopic state 200
open-close semaphore 266
push bit 233
readers* semaphore 266
retransmit state 286
sequence space 198
state 200
transmit state 248
window 211
writers* semaphore 266 TCP master device 325 TCP options 294 TCP output message



DELETE 249
KEEPALIVE 249
PERSIST 249
RETRANSMIT 249
SEND 249 TCP output process 20 TCP slave device 325 TCP state
ESTABLISHED 242 TCP timer process 20 tcp.h 196 tcpabort 568
tcpabort in tcpahort.c 236 tcpacked 568 repacked in tcpacked.c 301 tcpackit 568
tcpackit in tcpackit.c 303 tepbind 568 tepbind in tepbind.c 332 tepeksu 568 icpksum
in tcpcsum,c 208 icpclose 568 rpcclose in tcpclose c 343 tpdosed 568 tpdosed in
tcpclosed.c 218 tcpclose wait 568

tcpclosewait in tcpclosewaiF.c 226 tcpclosing 568
tcpclosing in tcpciosing.c 221
tcpctl 568
icpcnil in tcpcmt.c 345
tepcon 568
tepcon in icpcon.c 335
tcpdaia 568
tcpdaia in tcpdata.c 229
tepdemux 56S
lepdemux in tcpdemux.c 210
lepdodat 568
tep dodat in tcpdodat.c 232
tcp established 569
Icp estab-lished in icpestabJished.c 228
TCPE_NORMALMODE 313, 3L4
TCPE_TIMEDOUT 286
TCPE_URGENTMODE 313, 314
tcpfir 569
tcpfinl in scpfint.c 224
tcpfin2 569
tcpfir2in tcpfin2.c 222
icpfsm-h 200



TCPF_URG 3!0
tcpgelc 569
tcpgelc in rcpgetc.c 340
tcpgeidata 569
tcpgeidata in tcpgetdata.c 315
tcpgeispace 569
tcpgeispace.c 264
icph2nei 569
tcpdh2net in tcpdh2net.c 263
tcpdhomuch 5S9
tcpdhomuch in tcpdhomuch.c 260
tcpidle 569
tcpidl-e in tcpidle.c 252
tcpinil 569
tcpinil in tcpiniLc 351
tcpinp 569
tcpinp in tcpinp.v 206
tcpiss 569
tcpiss.c 267
tepkick 316, 569
tepkick 'mtcpkick.c 280

Inde*

tcpkilltimers 569
tcpkiKtimeis in icpkilltim^rs.c- 276
tcplastack 569
tcplastack in tcplastazk.c 227
tcplisaen 569
tcplisien in tcpiisten.c 243
icplq 569
tcplq in tcplq.c 346



tepmcntl 569
tcpmcntl in Icpmcntl.c 336
tcpmopen 569
icpmopen in. icpmopen.c 330
tcpnei2h 570
tcpnei2h in ltpriet2h.t; 208
tcpnntp 570
tcpnxip in tcprutp.v 334
tcpok 570
tcpok in tcpok.c 212
icpopts 570
icpopts in tvpopss.c 293
tcpstate 570
tcpstale in icpvst&tex 304
tcpout 20, 570
tcpout in tcpout.c 250
tcppersist 5^0
tcppersist in icppersist.c 252
tcpputc 570
tcpputc in icpputcc 341
tcpread 570
tcpreadin tcpread.c 338
tcpresei 570
tcpreset.c 261
tcprexmt 570
icpreMnt in tcprexmtx 286
tcpromss 570
tcpromss in tcprmss.c 294
tcprrt 570
tcprrt in rvprt.t' 298
tcpwindow 316, 570
tcpwindow in tcpnvindow.c 288



tcpsend 570
tcpsend in tcpserrd.c 256
tcpserver 570
tcpserver in tcpserver.r 331

tcpsrnss 570
icpsmss in tcpsmss.c 292
tcp&ndlen 570
tcpsndten in tcpsndien.c 259
tcpstat 571
tcpstat in tcpstat.c 348
tepstar.h 349
tcpswinttow 571
tcpwindow in tcpwindow.c 290
tcpswitch.c 213
tcpsync 571
tcpsync in tcpsync.c 238
tcpjnrcvd 571
icpsvnrcvd in tcpsynrcvd.c 240
tcpvnsem 57!
tcpsynsenf in tcpsynsem.c 239
tcptimer 20, 571
tcptimer in tcptimerc 272
tcptimer.h 270
tcptimewail 571
tcptimewail in tcptimewait .c 220
[cpuopt 57J
icptfopl in tcpuopi.c 350
tcpwait 571
tcpwait in tcpwait.c 219
tcpwakeup 571
tcpwakeup in tcpwakeup.c 265



tcpwinit 571
tcpwinit in tcpwinit.c 244
tcpwr 571
tcpwr in tcpwr.c 342
tcpwrite 571
tcpwrite in tcpwrite.c 340
tcpsmit 571
tcpsmit in tcpxmti.v 254
TCP^HLEN 569
tcpjn 569
tcpjn in icpin.i 205
TCP^MAXRETRIES 286
TCP_MAXRXT 287
tfcoaiesce 571
tfcnalesce in tfcoalesce.c 235
tfminsert 571
tfinsert in tfinsertx 234

612

Index

THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250



tmclear 57]
tmclcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toasci i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444



unmarked 572
unmount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571



THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184



udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlace ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572



Index

THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483



triggered updates 359

truncew 572

truncmb 572

type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265



X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571

tmset in tmset.c 278

toasci i 571

tolower 571

topology graph 382

toupper 571

TPA 571



TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313



User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260

window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276



tmset 571
tmset in tmset.c 278
toascii i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253, 300
transmit state 248, 304, 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361
 udp in udp.h 176
 udpcksum 572
 udpcksum in udpcksum.c 184
udpecho 572
 udph2net 572
 udph2net in udphlmt.c 182
 udpneCh 572
 tidpnet2h in
 udpnetlh.v 181
 udpnntp 572
 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
 udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572



upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260

window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219



TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toasci i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187



udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313
User Datagram Protocol 171
ut21lim 572
V
vector-distance 356 virtual link 439
W
wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265
X
Xinu
cross-reference 573
system call 573 xt2net 572
Z
ZSTimeConst 572

612

Index



THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toasci i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127



U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572



Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571

tmset in tmset.c 278

toasci i 571

tolower 571

topology graph 382

toupper 571

TPA 571

TPA in arp.h 42

tqdump 572

tqwrite 572

transmission control block 192



TRANSMIT state 253, 300

transmit state 248, 304, 305

trap 483

triggered updates 359

truncew 572

truncmb 572

type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184

udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439



W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571

tmset in tmset.c 278

toascii i 571

tolower 571



topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310



urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260

window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]



tmelcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571

tmset in tmset.c 278

toasci i 571

tolower 571

topology graph 382

toupper 571

TPA 571

TPA in arp.h 42

tqdump 572

tqwrite 572

transmission control block 192

TRANSMIT state 253. 300

transmit state 248, 304. 305

trap 483

triggered updates 359

truncew 572

truncmb 572

type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184

udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572



unmount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313
User Datagram Protocol 171
ut21lim 572
V
vector-distance 356 virtual link 439
W
wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X
Xinu
cross-reference 573
system call 573 xt2net 572
Z
ZSTimeConst 572

612

Index

THA 571
THA in arp.h 42



thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmelcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186



udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313
User Datagram Protocol 171
ut21lim 572
V
vector-distance 356 virtual link 439
W
wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265
X
Xinu
cross-reference 573
system call 573 xt2net 572
Z
ZSTimeConst 572



Index

THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmcicar.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359



truncew 572

truncmb 572

type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X



Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571

tmset in tmset.c 278

toascii i 571

tolower 571

topology graph 382

toupper 571

TPA 571

TPA in arp.h 42



tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253, 300
transmit state 248, 304, 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpecksum 572 udpecksum in udpecksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313
User Datagram Protocol 171



ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571



tmset in tmset.c 278
toascii i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO



update cascade 360
urgentdata I, 230, 310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498



timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmcicar.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571

tmset in tmset.c 278

toasci i 571

tolower 571

topology graph 382

toupper 571

TPA 571

TPA in arp.h 42

tqdump 572

tqwrite 572

transmission control block 192

TRANSMIT state 253. 300

transmit state 248, 304. 305

trap 483

triggered updates 359

truncew 572

truncmb 572

type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcsum 572 udpcsum in udpcsum.c 184

udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in

udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572



udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260

window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index



THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmcicar.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U



UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z



ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571

tmset in tmset.c 278

toascii 571

tolower 571

topology graph 382

toupper 571

TPA 571

TPA in arp.h 42

tqdump 572

tqwrite 572

transmission control block 192

TRANSMIT state 253. 300



transmit state 248, 304, 305

trap 483

triggered updates 359

truncEW 572

truncMB 572

type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W



wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571

tmset in tmset.c 278

toasci i 571

tolower 571

topology graph 382



toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253, 300
transmit state 248, 304, 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310



urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260

window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclear in tmclear.c 274



tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toasci i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580



unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260

window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8



time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toasci i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186



udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313
User Datagram Protocol 171
ut21lim 572
V
vector-distance 356 virtual link 439
W
wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265
X
Xinu
cross-reference 573
system call 573 xt2net 572
Z
ZSTimeConst 572



Index

THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmcicar.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toasci i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572



truncmb 572

type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpxntp 572 udpxntp in udpxn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu



cross-reference 573
system call 573 xt2net 572
Z
ZSTimeConst 572

612

Index

THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmcicar.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572



tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483

triggered updates 359

truncew 572

truncmb 572

type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313
User Datagram Protocol 171
ut21lim 572



V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfei.c 276

tmset 571

tmset in tmset.c 278



toascii i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360



urgentdata I, 230, 310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260

window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191



timer event 250
tmclear 57]
tmelcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182



unique names 444

unmarked 572

unmount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260

window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index



THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclear in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP



use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpxntp 572 udpxntp in udpxn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572



Index

THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305



trap 483

triggered updates 359

truncEW 572

truncMB 572

type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184

udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260



window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571

tmset in tmset.c 278

toascii 571

tolower 571

topology graph 382

toupper 571



TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253, 300
transmit state 248, 304, 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330



urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260

window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571



tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253, 300
transmit state 248, 304, 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
unmount 580
unsynchronized states 213



upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313
User Datagram Protocol 171
ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123



TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toasci i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572



udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
umount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313
User Datagram Protocol 171
ut21lim 572
V
vector-distance 356 virtual link 439
W
wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index



THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmcicar.c 274
tmieft 571
tmleft in tmfeci.c 276
tmset 571
tmset in tmset.c 278
toasci i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572



type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573



system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571

tmset in tmset.c 278

toascii 571

tolower 571

topology graph 382

toupper 571

TPA 571

TPA in arp.h 42

tqdump 572

tqwrite 572



transmission control block 192

TRANSMIT state 253. 300

transmit state 248, 304. 305

trap 483

triggered updates 359

truncew 572

truncmb 572

type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184

udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in

udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V



vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250

tmclear 57]

tmclcar in tmclear.c 274

tmieft 571

tmleft in tmfefi.c 276

tmset 571

tmset in tmset.c 278

toascii i 571



tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcsum 572 udpcsum in udpcsum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444
unmarked 572
unmount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310



urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571

THA in arp.h 42

thread of control 8

time-to-live 49, 89, 114, 123

TIME-WAIT state 219

TIMEOUT 13, 498

timeout with retransmission 191

timer event 250



tmclear 57]
tmclcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toasci i 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184
udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572
udpsend in udpsend.c 187
udp_in 572
udp_in in udpjn.c 182
unique names 444



unmarked 572
unmount 580
unsynchronized states 213
upalloc 572
upalloc Lr> upatlacc ISO
update cascade 360
urgentdata I, 230,310
urgent data bit 310
urgent data pointer 310
urgent mode 310, 330
urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572

612

Index

THA 571



THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483
triggered updates 359
truncew 572
truncmb 572
type-dependent format 127
U
UDP
use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184



udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlace ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260
window advertisement 287 write 324, 580 WRITERS 265

X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572



Index

THA 571
THA in arp.h 42
thread of control 8
time-to-live 49, 89, 114, 123
TIME-WAIT state 219
TIMEOUT 13, 498
timeout with retransmission 191
timer event 250
tmclear 57]
tmclcar in tmclear.c 274
tmieft 571
tmleft in tmfefi.c 276
tmset 571
tmset in tmset.c 278
toascii 571
tolower 571
topology graph 382
toupper 571
TPA 571
TPA in arp.h 42
tqdump 572
tqwrite 572
transmission control block 192
TRANSMIT state 253. 300
transmit state 248, 304. 305
trap 483



triggered updates 359

truncew 572

truncmb 572

type-dependent format 127

U

UDP

use by RIP 361 udp in udp.h 176 udpcksum 572 udpcksum in udpcksum.c 184

udpecho 572 udph2net 572 udph2net in udphlmt.c 182 udpneCh 572 tidpnet2h in
udpnetlh.v 181 udpnntp 572 udpnntp in udpn.Up.r 186

udpsend 572

udpsend in udpsend.c 187

udp_in 572

udp_in in udpjn.c 182

unique names 444

unmarked 572

umount 580

unsynchronized states 213

upalloc 572

upalloc Lr> upatlacc ISO

update cascade 360

urgentdata I, 230,310

urgent data bit 310

urgent data pointer 310

urgent mode 310, 330

urgent pointer 313

User Datagram Protocol 171

ut21lim 572

V

vector-distance 356 virtual link 439

W

wait 10, 580 walking the table 446 weigh! 384 wildcard 175 window 211, 260

window advertisement 287 write 324, 580 WRITERS 265



X

Xinu

cross-reference 573

system call 573 xt2net 572

Z

ZSTimeConst 572