Network Protocols

- The IPv4 protocol stack was the first set of protocols implemented within the network architecture of 4.2BSD.
- The protocols themselves were originally implemented at BBN (Bolt, Beranek and Newman) as part of a DARPA project.
- The IPv4 protocol stack is still the most important protocol stack in FreeBSD.
- This protocol implementation is the standard on which the current Internet is built.
- The description in chapter 13 is concentrated on the IPv4 stack although IPv6 and IPSec are also covered but not included in this course.

IPv4 Network Protocols

- The layering of the IPv4 protocols is illustrated in Fig.13.1.
- All the protocols use the services of IP.
- The transport level protocols, TCP and UDP, add a port identifier to IP's host identifier so that sockets (and processes) can be identified.
- The *Internet Control Message Protocol* (ICMP) is used for error reporting and simple network management tasks.
- Raw access to IP and ICMP is possible through *raw sockets*.
- The Internet protocols was designed for a heterogeneous environment.
  - → Not even the *byte* is of the same size on all systems.
  - → The network protocols specify the data unit to be an *octet* - an 8-bit byte.
  - → All fields in the Internet protocols larger than an octet are expressed in *network byte order*, with the most significant octet first.
  - → The network implementation use macros to convert 16-bit and 32-bit fields between host and network byte order.

IPv4 Addresses

- An IPv4 address is a 32-bit number that identifies the network at which a host resides and also uniquely identifies a network interface on that host.
- Network addresses are assigned in blocks by *Regional Internet Registries* (RIRs) to *Internet Service Providers* (ISPs), which then give out addresses to companies or individual users.
- Historically IPv4 addresses were divided into three classes (A, B and C) with a different number of network address bits in each class.
- The current IPv4 address scheme is called *Classless Inter-Domain Routing* (CIDR).
- In the CIDR scheme each organization is given a contiguous group of addresses described by a single value and a network mask.
- It is because of this addressing scheme that routing entries store arbitrary netmasks with routes.

IPv4 Addresses cont.

- Each Internet address is maintained in an *in_ifaddr* struct (Fig. 13.2).
- The network mask for an interface is recorded in the *ia_subnetmask* field.
- The address of the subnet connected to an interface is stored in the *ia_subnet* field.
- If the address in a received packet masked with *ia_subnetmask* equals *ia_subnet*, the packet is for a local subnet.

Broadcast Addresses

- Originally 4.2 BSD used the address with a host part of zero for broadcasts.
- Later the broadcast address was defined to be the address with a host part of all 1s.
- On input, FreeBSD recognizes broadcast addresses with host parts of all 0s or all 1s and also the address with 32 bits of 1.

Internet Multicast

- IP multicasts are sent using destination addresses with the high-order bits set to 1110.
- Multicast addresses do not contain network and host parts, instead the entire address names a group using a particular service.
- IP multicast addresses map directly to physical multicast addresses at networks such as Ethernet.
- For a socket to use multicast it must join a multicast group using the *setsockopt* system call.
- The *setsockopt* call informs the link layer that it should receive multicasts for the corresponding link-layer address.

Internet Ports and Associations

- At the IP level, packets are addressed to hosts.
- However, each packet contains an 8-bit protocol number that identifies the protocol that is one level up in the stack.
- The transport protocols TCP and UDP use 16-bit port numbers to designate the connection or communication ports on a host.
- Each transport protocol maintains its own mapping between port numbers and processes or descriptors.
- An association is specified by a tuple $<$source address, destination address, protocol number, source port, destination port$>$.
- Connection oriented protocols must enforce the uniqueness of associations - other protocols usually do so as well.

Protocol Control Blocks

- For each TCP- or UDP-based socket, an *Internet protocol control block* (*inpcb* struct) is created (Fig. 13.3).
- The *inpcb* struct hold Internet addresses, port numbers and pointers to auxiliary data structures.
- TCP in addition creates a *TCP control block* (*tcpcb* struct) to hold the protocol state information needed for its implementation.
- Internet control blocks for use with TCP are held on a doubly linked list private to the TCP module.
- Internet control blocks for use with UDP are held on a similar list private to the UDP module.
- Two separate lists are needed because each transport protocol has a distinct space of port identifiers.
- There are a set of common routines to maintain the *inpcb* lists.
- IP demultipexes incoming packets based on the protocol identifier.
- Each higher level protocol is then responsible for checking its list of *inpcb* blocks to direct a message to the correct socket.

User Datagram Protocol (UDP)

UDP is a simple unreliable datagram protocol.

- The implementation of the Internet protocols are tightly coupled.
  - → Transport protocols send and receive packets including an IP pseudo_header containing source and destination addresses, protocol identifier and packet length.
- UDP protocol headers are extremely simple, containing only source and destination port numbers, datagram length and data checksum.
- The *udp_output* routine also adds an IP pseudo header to the packet.

UDP

Initialization

- When a new datagram socket is created, the socket layer locates the protocol-switch for UDP and calls *udp_attach()* with the socket as parameter.
- *udp_attach()* calls *in_pcballoc()* to create a new protocol control block (*inpcb*) on its list of current sockets.
- It also sets the default limits to the sockets send and receive buffers.
  - → This value is used as an upper limit for the datagram size.
- The UDP protocol-switch entry contains the flag PR_ATOMIC, requiring all data in a datagram to be sent at the same time.
- An application program may use the *bind* system call to assign a port number to the socket.
- For UDP this call results in a call to *udp_bind()*.
  - → *udp_bind()* calls *in_pcbbind()* which verifies that the port number is not in use and records the local part of the association in *inpcb*.
  - → If the address was not specified it is left unspecified.
    - ✯ In this case any local address will match at input and on output an appropriate address will be chosen.

UDP - Output

- Typically the system calls *sendto* or *sendmsg* are used to send datagrams.
- These system calls will call *udp_send()* with a mbuf chain as parameter.
- The output routine for UDP is *udp_output()* that takes the following parameters:

```
int udp_output (
    struct inpcb *inp,   /* Protocol control block */
    struct mbuf *msg,    /* The data to send */
    struct mbuf *addr,   /* Optional dest. address */
    struct mbuf *control, /*Optional control info */
    struct thread *td);  /* Pointer to process */
```

- To send datagrams the system must also know the remote part of the association.
- A program can specify this address and port with each send operation or they can be set in advance using the *connect* system call.
- In either case *udp_output()* will call *in_pcbconnect()* to record the destination address and port in *inpcb*.
  - → If the local address is not bound and if a route for the destination is found, the address to the outgoing interface is used as local address.
  - → If no local port number was bound, one is chosen at this time.

UDP - Output Cont.

- Finally *udp_output()* prepends space for the UDP and IP headers to the *msg* mbuf and fills in the headers.
- A checksum is also calculated and added.
- The packet is passed to the IP level by calling *ip_output()*.

UDP - Input

- Each Internet protocol layered directly on top of IP is called from IP using the following call:

```
(void) (*pr_input) (
         struct mbuf *m, int off);
```

- The first parameter is an mbuf chain including the IP header and the second parameter is an offset giving the size of the IP header.
- For UDP the input routine is named *udp_input()*.

Udp_input() Implementation:

- Check if the length of the first mbuf is shorter than the IP plus UDP headers.
  - → If shorter, call *m_pullup()* to make the headers contiguous.
- Check that the packet is of correct length and calculate the checksum.
  - → If any test fails - drop the packet.
- Check if multicast message and do special handling if it is.
- Call *in_pcblookup()* with the *address* and *port* numbers in the packet as parameters, to locate the protocol control block for the receiving socket.

UDP - Input Cont.

- There may be several control blocks with the same local port number but different addresses.
- In this case the *inpcb* with the best match is selected.

In_pcblookup():

- If an exact association is found that *inpcb* is selected.
- Check for a wildcard match.
  - → Any *inpcb* with correct local port but unspecified local address, remote port or remote address will match.

If an *inpcb* is located, the packet is queued in the receive buffer of the matching socket and a wakeup is made for a waiting process.

If no *inpcb* was located, an ICMP *port unreachable* error message is sent.

In most cases no one will receive the error message since associations for UDP usually are temporary.

Internet Protocol (IP)

- IP is the protocol responsible for host-to-host addressing, routing, packet forwarding and packet fragmentation and reassembly.
- IP does not always operate for a socket at the local host.
- IP may also:
  - → Forward packets.
  - → Receive packets for which there is no local socket.
  - → Generate error packets in response to these situations.

IP

- The IP protocol header is shown in Fig. 13.4.
- The IP header includes source and destination addresses and the destination protocol.
- The fragment field is used if a packet must be broken into smaller fragments for transmission.

IP Output

The IP output routine that is called from the transport level output routines takes the following parameters:

```
int ip_output (
    struct mbuf *msg,    /* The data to send */
    struct mbuf *opt,    /* Optional IP options mbuf*/
    struct route *ro,    /* Normally set to NULL */
    int flags,
    struct ip_moptions *imo,  /* Multicast options */
    struct inpcb *inp);  /* Pointer to inpcb */
```

- For normal operation only the *msg* parameter and possibly *opt* and *flags* are used.
- Since cached routes was removed from the *inpcb* struct in FreeBSD5.2 the *ro* parameter is usually set to NULL and routing lookup is performed by *ip_output()*.
- The *msg* mbuf already contains a partially filled in IP header allocated by the transport protocol.

IP Output

The *ip_output()* routine works as follows:

- Fill in IP options if present.
- Fill in the remaining header fields including IP version, header length.
- Determine the route.
- Check for multicast or broadcast and do special handling in for these cases.
- Do possible IPSec manipulations.
- Do possible packet filtering.
- If the packet size is no larger than the maximum packet size for the outgoing interface, compute the checksum and call the *if_output()* routine to send the packet.
- If the packet size is larger than the maximum packet size for the outgoing interface, break the packet in fragments and send them in turn.

IP Output

Routing step in more detail:

- If the *ro* parameter is NULL, *ro* is set to point to a local route struct.
- The destination address in the *route* struct is set to the destination address in the IP header.
- *rtalloc()* is called to allocate a route.
- The returned *rtentry* struct includes a pointer to an *ifnet* struct and an *ifaddr* struct for the outgoing interface.
- If the RTF_GATEWAY flag is set for the route, the destination address passed to the link layer is set from the *rt_gateway* field.
- The interface output routine, i*f_output()*, is called via the pointer in the *ifnet* struct.
- The interface output routine will validate the destination address and place the packet at the output queue.
- An error is returned from the *if_output()* routine only if the packet could not be sent.

IP Input

- The Iv4 *ip_input()* routine is typically called from the *swi_net* network thread.
- The *ip_input()* routine is called with a mbuf chain that contains the received packet.
- A packet may be processed in four different ways:
  - → Passed as input to a higher level protocol.
  - → It encounters an error that is reported back to the source.
  - → It is dropped due to an error.
  - → It is forwarded to the next-hop destination.

IP Input

Ip_input() Implementation:

- Verify that the packet is at least as long as the IPv4 header and ensure that the header is contiguous in one mbuf.
- Checksum the packet header and drop the packet if an error is detected.
- Verify that the packet is at least as long as the header indicates and drop the packet if it is not.
- Do any filtering or IPSec functions.
- Process the options in the header.
- Check whether the packet is for this host.
  - → If it is not and acting as a router, forward the packet. If not router drop the packet.
  - → If for this host - continue processing.
- If the packet has been fragmented, keep it until all fragments has arrived or it is to old.
- Pass the packet to the next-higher level protocol.

IP Input

Locating next-higher level protocol input routine.

- Which *pr_input()* routine to call is determined by the 8-bit protocol field in the IP header.
- The 8-bit protocol identifier gives 256 possible protocols.
- All Internet protocols above IP level are called through a global array of *protosw* structs, called *inetsw*[].
- The *inetsw[]* array is statically initialized with one entry for each protocol that is implemented in the system.
- To map between the 8-bit protocol number and the correct entry in *inetsw[]*, the 256 element *ip_protox* array is used.
- All 256 entries in *ip_protox* are initially set to index the raw ip entry in the *inetsw[]* protocol switch by the *ip_init()* routine.
  - → The *ip_init()* routine will then cycle through i*netsw[]*, and for every protocol that has a kernel implementation it will modify the *ip_protox* entry to index the correct *inetsw[]* entry.
- The *ip_input()* routine will call the next-level protocol with the following call:

```
(*inetsw[ip_protox[ip->ip_p]].pr_input)(m, hlen);
```

IP Forwarding

Details of packet forwarding code:

- Check that forwarding is enabled. If not drop the packet.
- Check that the destination address is one that allows forwarding.
- Save the IP header and some data bytes in case an ICMP error message need to be generated.
- Determine the route to be used in forwarding the packet.
- If the outgoing interface is the same that the packet was received on, possibly send an ICMP redirect message to the originating host.
- Call *ip_output()* to send the packet.
- If an error is detected, possibly send an ICMP error message.

IP Forwarding

- Misconfigured routers may be a big problem for the Internet.
- For this reason the router functions in FreeBSD are disabled by default.
- They may be enabled at runtime with the *sysctl* system call.
- Hosts not configured as routers never attempt to forward packets or return error messages in response to misdirected packets.

Transmission Control Protocol (TCP)

TCP includes several features not found in the simpler protocols such as:

- Explicit connection initiation and termination.
- Reliable unduplicated delivery of data
- Flow control
- Out-of-band indication of urgent data.
- Congestion avoidance.

TCP Protocol

- A TCP connection is a bidirectional sequenced stream of data between two peers.
- The stream initiation and termination (SYN and FIN) are explicit events that occupy positions in the *sequence space* of the stream.
- Initiation and termination packets are acknowledged in the same way as data.
  - → Sequence numbers are 32 bits from a circular space.
- The sequence numbers for each direction starts with an arbitrary value (*initial sequence number*) sent in the initial packet for a connection.
  - → Arbitrary *initial sequence numbers* are used to prevent spoofing based on guessing *initial sequence numbers*.
- Each TCP packet contains:
  - → The starting sequence number of the data in that packet.
  - → An acknowledgment of all contiguous data received from the remote side.

TCP Protocol

- The *acknowledgment number* is the sequence number of the next packet the site expects to receive.
- Acknowledgments are cumulative and thus may acknowledge more than one packet.
- Flow control is done with a *sliding-window scheme*.
  - → Each packet with an acknowledgment contains a window, that is the number of octets of data that the receiver is prepared to accept.
- The header for TCP is shown in Fig. 13.6.
- The TCP header also includes some flags:
  - → SYN - “Begin connection by synchronizing sequence numbers”
  - → FIN - “Finishing of transmission”
  - → ACK - “Acknowledgment field is valid”
  - → RST - “Reset of connection”
  - → URG - “Urgent data present”
  - → PSH - “Push the arriving data up to the application level”

TCP Protocol

- The options are encoded in the same way as for IP:
- The *no-operation* and *end-of-options* are one octet.
- All other options contains a type and length.
- FreeBSD includes three options along with the SYN when initiating a connection:
  - → Maximum receive segment size.
  - → Window scaling - The number of bits to shift the *window* value.
  - → Timestamp option
- The options must be sent in both directions to take effect.

TCP Connection

The normal connection procedure is known as a three-way handshake:

1. Host A sends SYN
2. Host B sends SYN,ACK
3. Host A sends ACK

- The list of connection states for a TCP connection is shown in Fig. 13.1 and the state diagram in Fig. 13.7
- After a connection is established, each peer includes an acknowledgment and window information in each packet.
- If the sender do not receive an acknowledgment in a reasonable time, it resends the data.
- If data is received out of order, the receiver generally retains the data but it cannot be acknowledged until the missing segment is received.

TCP Connection termination

Each peer can terminate a connection at any time by sending a packet with the FIN bit:

1. The FIN is acknowledged advancing the sequence number by 1.
2. The acknowledgment of the FIN terminates the connection.
3. The peer that sends the last ACK of FIN enters the TIME_WAIT state.
   - The peer remains in TIME_WAIT state for 2MSL (two times Maximum Segment Lifetime).
   - The reason for this is the need to repeat the last ACK if it was lost and the FIN is resent.

TCP State Variables

- Each TCP connection maintains a large set of state variables in the TCP control block (tcpcb).
- A set of sequence variables are used to keep track of the send and receive windows (Fig. 13.8).
- The sequence variables are explained in Table 13.2.
- The sequence variables are used in the output module to decide if data can be sent, and in the input module to decide if received data can be accepted.
- If the timestamp option is used, the tests to see if received packets are acceptable are augmented with checks on the timestamp.

TCP Algorithms

TCP processing occurs in response to the following events:

1. A request from the user (such as reading or writing data, or opening or closing a connection).
2. The receipt of a packet for the connection.
3. The expiration of a timer.

- These events are handled by *tcp_usr_send()*, *tcp_input()* and a set of timer routines.
  - → These routines processes the event and makes any necessary state changes.
  - → For any transition that requires output, *tcp_output()* is called.

TCP Algorithms

- The criteria for sending a packet with data or control information is complicated in TCP.
- Any of the following may allow data to be sent that could not be sent previously:
  - → A user write operation.
  - → The receipt of a window update from the peer.
  - → The expiration of a retransmission timer.
  - → The expiration of a window-update timer.
- In addition control packets may be sent due to:
  - → A change in connection state (e.g. open or close)
  - → Receipt of data that must be acknowledged.
  - → A change in the receive window due to removal of data from the input queue.
  - → A send request with urgent data.
  - → A connection abort.

TCP Timers

- To prevent the protocol from hanging if packets are lost, each connection maintains a set of timers.
- The timers are set via the kernel *callout* service.
- The following timers are used by TCP (Table 13.3):

tcp_timer_rexmt Retransmit timer. Started whenever data is sent on a connection. Stopped when all outstanding data on a connection is acknowledged.

tcp_timer_persist Persist timer. Protects against loss of a window update. Started when data is ready to be sent but the send window is too small unless the retransmit timer is active.

tcp_timer_keep Keepalive timer. This timer have two different uses:

1. During connection establishment the timer limits the time for a tree-way-handshake to terminate.
2. A nonstandard timer that may be used to detect idle connections if the socket level SO_KEEPALIVE option is set.

TCP Timers Cont.

tcp_timer_2msl "Twice the maximum segment lifetime" timer. Started when a connection enters the TIME_WAIT state. In FreeBSD also started when a connection enters the FIN_WAIT_2 state because some TCP implementations fail to send a FIN on a receive-only connection.

tcp_timer_delack Delayed ack timer. Explained in section 13.6.

TCP Round-Trip Time

- The correct value for the retransmission timeout depends on the delays in the network.
- A measure on the network delays is the *round-trip time* (rtt) which is the time to receive an acknowledgment for a sent packet.
- The system stores a smoothed average of the round-trip time (srtt) and a smoothed variance (rttvar).
- The initial retransmission timeout is set to srtt + 4*rttvar.

TCP Connection Establishment

Two ways for a node to establish a connection:

1. An active connection is established via a *connect* system call.
2. A passive connection is created as the result of waiting on an *accept* system call.

Active Connection

- When a new TCP socket is created, the socket layer locates the protocol-switch for TCP and calls *tcp_attach()* with the socket as parameter.
- *tcp_attach()* calls *in_pcballoc()* to create a new protocol control block (*inpcb*) and then creates an additional TCP control block (*tcpcb*).
- If the process calls *bind* to assign an address or port number, this is identical to the UDP case.
- The *connect* system call calls *tcp_connect()*:
  - → Calls *in_pcbconnect()* to set up an association - identical to UDP case.
  - → Selects an initial sequence number.
  - → *soisconnecting()* is called to set the socket in the SS_ISCONNECTING state.
  - → The TCP state is set to SYN_SENT.
  - → The keepalive timer is started (75 sec.).
  - → *tcp_output()* is called to send the packet.

TCP Connection Establishment cont.

- The connect packet normally contains three options: maximum-segment-size option, window-scale option and timestamps option.
- The maximum-segment-size (MSS) option communicates the maximum segment size that TCP is willing to accept on the connection.
- The initial MSS value is taken from the MTU value (maximum transmission unit) in the route for the destination.
- The MTU values in the route table are usually initialized from the interface MTU value.
- TCP can use the *Path MTU Discovery* method to discover the real MTU for a connection.
  - → The network is probed by sending packets with the IP *don't fragment* flag set.
  - → If the packet is to big for some network segment an ICMP error message is returned which contains the maximum packet size that is accepted.
  - → This value is recorded as the new MTU value in the host cache and the packet is resent.

TCP Connection Establishment cont.

Host Cache

- The host cache keeps information on the measured TCP parameters of past TCP sessions.
- Whenever a new connection is opened, a call is made to *tcp_hc_get()* to find information about a past connection.
- Lookup in the cache is on the remote IP address.
- If a cached entry is found, it is used to get better initial start values for some connection parameters (Table 13.4).
- Can lead to significant speedups for new tcp connections after the first one.

TCP Connection Establishment

Passive Connection

- A socket is created in the same way as for an active connection.
- Process calls bind to assign an address.
- The process calls listen, which calls usr_tcp_listen() to set the socket in TCPS_LISTEN state.
- When a packet arrives for a TCP socket in TCPS_LISTEN state:
  - → Create a new socket with *sonewconn()*.
    - ✯ Calls *tcp_usr_attach()* via *pru_attach()* to create a new inpcb and a new tcpcb.
    - ✯ Put new socket in so_incomp queue at the original socket.
  - → If packet has SYN
    - ✯ Call *in_pcbconnect()* to record the remote part of the association in *inpcb*.
    - ✯ Change state to TCPS_SYN_RECEIVED.
    - ✯ Set keepalive timer and set TF_ACKNOW and call *tcp_output()*.
    - ✯ tcp_output() sends a SYN/ACK packet.

TCP Connection Establishment Cont.

Passive Connection Cont.

- When an ACK arrives for a TCP socket in TCPS_SYN_RECEIVED state:
  - → Call *soisconnected()* to move the socket from the so_incomp queue to the so_comp queue.
  - → Change to ESTABLISHED state.
  - → Wakeup any process sleeping on an accept system call for the socket.

TCP SYN Cache

- Unfortunately the normal TCP connection establishment previously described became the target for a *denial of service attack*.
- A system could be blocked by sending a flood of SYN packets to it.
- To combat this attack, the *syncache* was introduced.
- The syncache handles the three-way handshake in a more efficient way than the previous implementation.
- When a SYN packet is received for a socket in TCPS_LISTEN state, *syncache_add()* is called.
  - → All information about the arrived SYN packet is stored in the syncache.
  - → The state remains TCPS_LISTEN.
  - → A SYN/ACK packet is returned but no data is acknowledged.
- When an ACK is received for a socket in TCPS_LISTEN state *syncache_expand()* is called.
  - → If a syncache entry is found for the ACK packet, do all actions that should have been done when SYN was received and enter TCPS_SYN_RECEIVED state.
- tcp_input() finishes the handling of the ACK packet in the normal way.

TCP Connection Shutdown

- A TCP connection is symmetrical so either side may initiate disconnect independently.
- As long as one direction of a connection can carry data, the connection remains open.
- A process may indicate that it has completed sending data with the *shutdown()* system call.
  - → *tcp_usr_shutdown()* called to switch to TCP_FIN_WAIT1 state.
  - → tcp_output() called to send FIN packet.
- The receiving socket will advance to TCPS_CLOSE_WAIT state, but may continue to send data.
- If the process calls *close*, the socket will also enter TCP_FIN_WAIT1 state and FIN is sent.
  - → In this case the socket is changed to SS_ISDISCONNECTING state and no more data can be received.
  - → If data remain in the send buffer, TCP will try to deliver them.
- The socket is freed when the last reference to it disappears.

TCP Input Processing

- Tcp_input() is called in the same way as udp_input() and the first steps are very similar.
- Locate pcb for port# (if none, send RST).
- If socket is in LISTEN state, do connection establishment for passive connection.
- Process options.
- Accept data if rcv.wnd$>$0 and seq# space is within window (trim data before & after window).
  - → If rcv_wnd=0, accept packet with no data & seq#=rcv_nxt.
  - → If not acceptable, drop & send ACK.

Flags

- if RST, close connection & drop packet
- if ACK not set, drop
- if ACK seq# $>$ previous ack#,
  - → if in SYN_RECEIVED & ACK is for SYN, enter ESTABLISHED state.
  - → if ACK includes seq# for RTT measurement, average time sample into srtt.
  - → if all outstanding data are ack'ed, stop rexmt timer, else set rexmt to current value.
  - → drop ack'ed data from socket send queue.

TCP Input Processing Cont.

- Window update
  - → if packet seq# $>$ previous update, or same seq# but ack# is higher, or seq# & ack#'s are the same but window is larger, record a new window size.
- URG
  - → Process urgent data.
- If data begins at rcv_nxt, put data in socket recv buffer with *sbappendstream()*.
  - → Otherwise, put in per-connection queue until missing data arrives.
  - → Set TF_DELACK in pcb.
- If FIN, mark socket with *socantrcvmore()*.
  - → If sender is closed, mark with *soisdisconnected()*.
  - → Set TF_ACKNOW.
- If needed *tcp_output()* is called to send a response.

TCP Output Processing

- *Sosend()* accumulates data in mbuf chain and calls tcp_usr_send (pru_send).
- *tcp_usr_send()* calls *sbappendstream()* to place data in the sockets send buffer and calls tcp_output().
- Allocate mbuf for packet header and call *tcpip_fillheaders()* to fill in header.
  - → call m_copy() to get data from send buffer.
- Set seq# from *snd_nxt* & ack# from *rcv_nxt*.
  - → Get flags from connection state.
  - → Compute window advertisement.
  - → If out of band data is sent, set URG flag.
  - → Set PSH, if all data in send buffer is sent.
  - → Compute checksum.
- Call i*p_output()* to send data.
- Start *rexmt* timer and update *snd_nxt* & *snd_max*.