Operating Systems

17. Sockets

Paul Krzyzanowski
Rutgers University
Spring 2015

Sockets

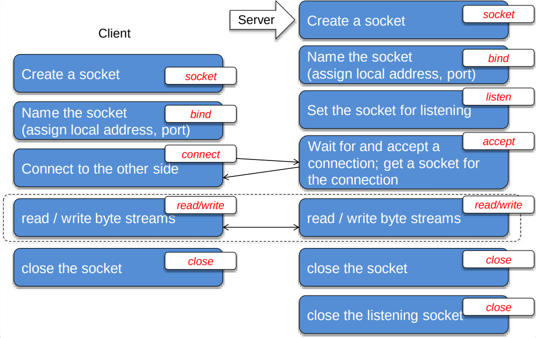
- Dominant API for transport layer connectivity
- Created at UC Berkeley for 4.2BSD Unix (1983)
- Design goals
  - Communication between processes should not depend on whether they are on the same machine
  - Communication should be efficient
  - Interface should be compatible with files
  - Support different protocols and naming conventions
    - *Sockets is not just for the Internet Protocol family*

Socket

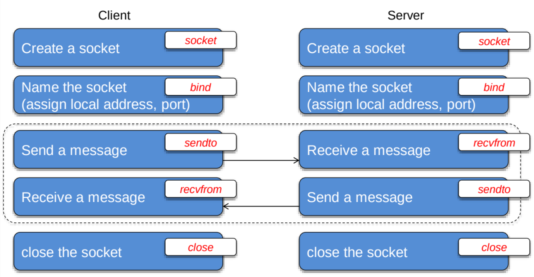
Socket = Abstract object from which messages are sent and received

- Looks like a file descriptor
- Application can select particular style of communication
  - Virtual circuit, datagram, message-based, in-order delivery
- Unrelated processes should be able to locate communication endpoints
  - Sockets can have a *name*
  - Name should be meaningful in the communications domain

Connection-Oriented (TCP) socket operations

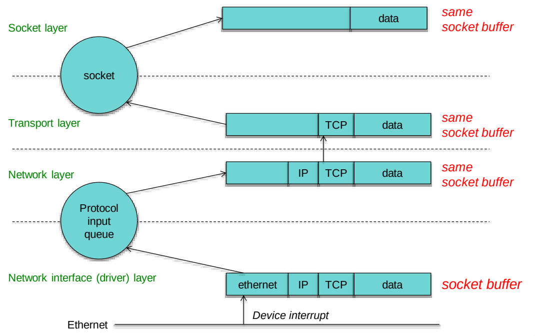


Connectionless (UDP) socket operations

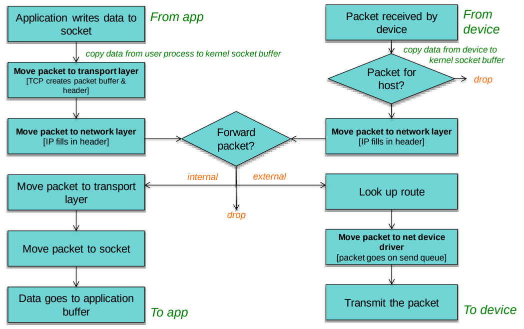


Socket Internals

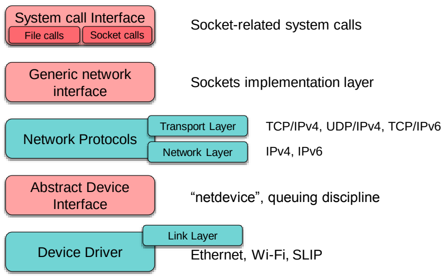
Logical View



Data Path



OS Network Stack



System call interface

Two ways to communicate with the network:

Socket-specific call
(e.g., *socket, bind, shutdown*)
- Directed to *sys_socketcall* (socket.c)
- Goes to the target function

File call
(e.g., *read, write, close*)
File descriptor ≡ socket
  - Sockets reside in the process's file table
- Direct parallel of the VFS structure
  - A socket's *f_ops* field points to a set of functions for socket operations

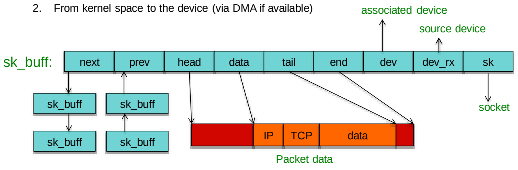
A `socket` structure acts as a queuing point for data being transmitted & received
- A socket has *send* and *receive* queues associated with it
  - High & low watermarks

Sockets layer

- All network communication takes place via a socket
- Two socket structures – one within another
  1. Generic sockets (*aka* BSD sockets) – `struct socket`
  2. Protocol-specific sockets (e.g., INET socket) – `struct sock`
- *`socket`* structure
  - Keeps all the state of a socket including the protocol and operations that can be performed on it
  - Some key members of the structure:
    - struct proto_ops *ops: protocol-specific functions that implement socket operations
      - Common functions to support a variety of protocols: TCP, UDP, IP, raw ethernet, other networks
      - Pointers to protocol functions: *bind, connect, accept, listen, sendmsg, shutdown, …*
    - struct inode *inode: points to in-memory inode associated with the socket
    - struct sock *sk: protocol-specific (e.g., INET) socket
      - E.g., this contains TCP/IP and UDP/IP specific data for an INET (Internet Address Domain) socket

Socket Buffer: `struct sk_buff`

- Component for managing the data movement for sockets through the networking layers
  - Contains packet & state data for multiple layers of the protocol stack
- Don't waste time copying parameters & packet data from layer to layer of the network stack
- Data sits in a socket buffer (struct sk_buff)
- As we move through layers, data is only copied twice:
  1. From user to kernel space
  2. From kernel space to the device (via DMA if available)

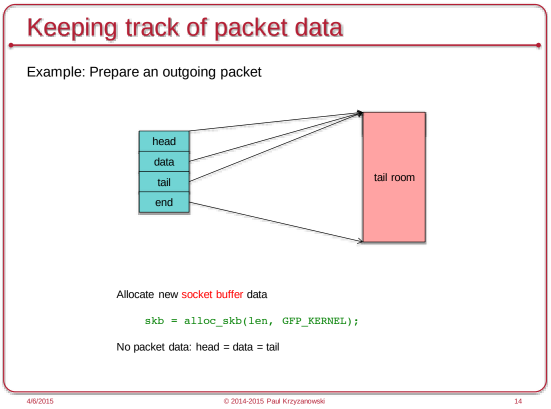


Socket Buffer: `struct sk_buff`

- Each sent or received packet is associated with an sk_buff:
  - Packet data in `data->`, `tail->`
  - Total packet buffer in `head->`, `end->`
  - Header pointers (MAC, IP, TCP header, etc.)

Add or remove headers without reallocating memory

- Identifies device structure (`net_device`)
  - `rx_dev`: points to the network device that received the packet
  - `dev`: identifies net device on which the buffer operates
    - If a routing decision has been made, this is the outbound interface
- Each socket (connection stream) is associated with a linked list of sk_buffs



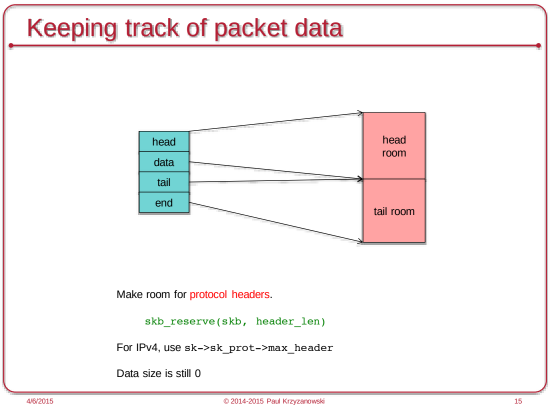
Keeping track of packet data

Example: Prepare an outgoing packet

Allocate new socket buffer data

```
skb = alloc_skb(len, GFP_KERNEL);
```

No packet data: head = data = tail



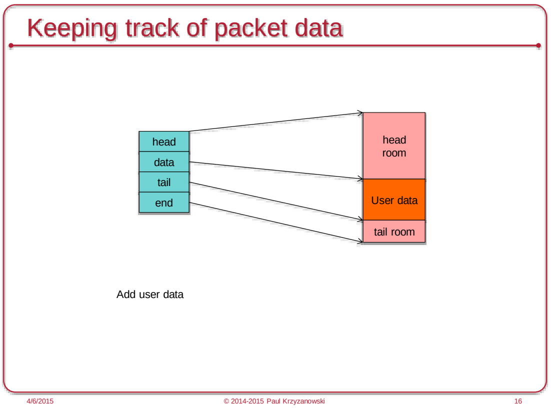
Keeping track of packet data

Make room for protocol headers.

```
skb_reserve(skb, header_len)
```

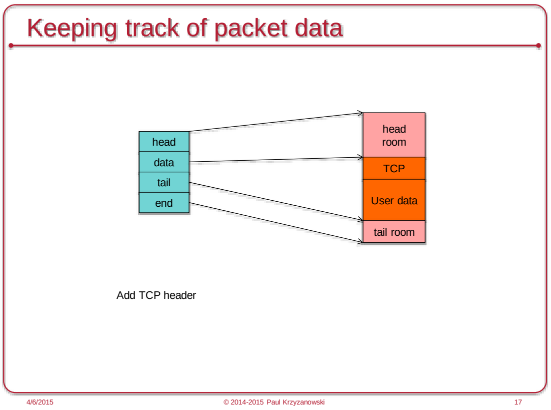
For IPv4, use `sk->sk_prot->max_header`

Data size is still 0



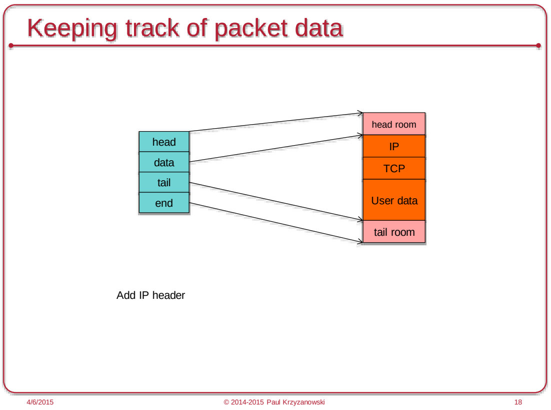
Keeping track of packet data

Add user data



Keeping track of packet data

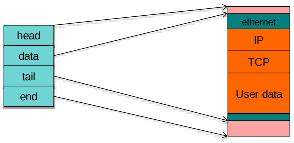
Add TCP header



Keeping track of packet data

Add IP header

Keeping track of packet data



Add ethernet header

The outbound packet is complete!

Network protocols

- Define the specific protocols available (e.g., TCP, UDP)
- Each networking protocol has a structure called *proto*
  - Associated with an "address family" (e.g., AF_INET)
  - Address family is specified by the programmer when creating the socket
  - Defines socket operations that can be performed from the sockets layer to the transport layer
    - *Close, connect, disconnect, accept, shutdown, sendmsg, recvmsg*, etc.
- Modular: one module may define one or more protocols
- Initialized & registered at startup
  - Initialization function: registers a family of protocols
  - The *register* function adds the protocol to the active protocol list

Abstract device interface

- Layer that interfaces with network device drivers
- Common set of functions for low-level network device drivers to operate with the higher-level protocol stack

Abstract device interface

- Send a packet to a device
  - Send sk_buff from the protocol layer to a device
    - dev_queue_xmit function
    - enqueues an sk_buff for transmission to the underlying driver
    - Device is defined in sk_buff
      - Device structure contains a method hard_start_xmit: driver function for actually transmitting the data in the sk_buff
- Receive a packet from a device & send to protocol stack
  - Receive an sk_buff from a device
    - Driver receives a packet and places it into an allocated sk_buff
    - sk_buff passed to the network layer with a call to netif_rx
    - Function enqueues the sk_buff to an upper-layer protocol's queue for processing through netif_rx_schedule

Device drivers

- Drivers to access the network device
  - Examples: ethernet, 802.11n, SLIP
- Modular, like other devices
  - Described by `struct net_device`
- Initialization
  - Driver allocates a `net_device` structure
  - Initializes it with its functions
    - `dev->hard_start_xmit`: defines how to transmit a packet
      - Typically the packet is moved to a hardware queue
    - Register interrupt service routine
  - Calls *`register_netdevice`* to make the device available to the network stack

Sending a message

- Write data to socket
- Socket calls appropriate *send* function (typically INET)
  - Send function verifies status of socket & protocol type
  - Sends data to transport layer routine (typically TCP or UDP)
- Transport layer
  - Creates a socket buffer (struct sk_buff)
  - Copies data from application layer; fills in header (port #, options, checksum)
  - Passes buffer to the network layer (typically IP)
- Network layer
  - Fills in buffer with its own headers (IP address, options, checksum)
  - Look up destination route
  - IP layer may fragment data into multiple packets
  - Passes buffer to link layer: to destination route's device output function
- Link layer: move packet to the device's xmit queue
- Network driver
  - Wait for scheduler to run the device driver's transmit code
  - Sends the link header
  - Transmit packet via DMA

Routing

IP Network layer

Two structures:

1. Forwarding Information Base (FIB)
   Keeps track of details for every known route
2. Cache for destinations in use (hash table)
   If not found here then check FIB.

Receiving a message – part 1

- Interrupt from network card: packet received
- Network driver – top half
  - Allocate new sk_buff
  - Move data from the hardware buffer into the sk_buff (DMA)
  - Call *netif_rx*, the generic network reception handler
    - This moves the sk_buff to protocol processing (it's a work queue)
    - When netif_rx returns, the service routine is finished
  - Repeat until no more packets in the device buffers
- If the packet queue is full, the packet is discarded
- `netif_rx` is called in the interrupt service routine
  - Must be quick. Main goal: queue the packet.

Receiving a packet – part 2

Bottom half

- Bottom half = "softIRQ" = work queues
  - Tuples containing < *operation, data* >
- Kernel schedules work to go through pending packet queue
- Call `net_rx_action()`
  - Dequeue first sk_buff (packet)
  - Go through list of protocol handlers
    - Each protocol handler registers itself
    - Identifies which protocol type they handle
    - Go through each generic handler first
    - Then go through the *receive* function registered for the packet's protocol

Receiving an IP packet – part 3

Network layer

"Ethernet Protocol: IP"

- IP is a registered as a protocol handler for ETH_P_IP packets
  - Packet header identifies next level protocol
    - E.g., Ethernet header states encapsulated protocol is IPv4
    - IPv4 header states encapsulated protocol is TCP
  - IP handler will either route the packet, deliver locally, or discard
    - Send either to an outgoing queue (if routing) or to the transport layer
  - Look at protocol field inside the IP packet
    - Calls transport-level handlers (*tcp_v4_rcv*, udp_rcv, *icmp_rcv*, …)
  - IP handler includes *Netfilter* hooks
    - Additional checks for packet filtering, port translation, and extensions

Receiving an IP packet – part 4

Transport layer

- Next stage (usually): tcp_v4_rcv() or udp_rcv()
  - Check for transport layer errors
  - Look for a socket that should receive this packet
    (match local & remote addresses and ports)
  - Call tcp_v4_do_rcv: passing it the sk_buff and socket (`sock` structure)
    - Adds sk_buff to the end of that socket's receive queue
    - The socket may have specific processing options defined
      - If so, apply them
- Wake up the process (ready state) if it was blocked on the socket

Lots of Interrupts!

- Assume:
  - Non-jumbo maximum payload size: 1500 bytes
  - TCP acknowledgement (no data): 40 bytes
  - Median packet size: 413 bytes
- Assume a steady flow of network traffic at:
  - 1 Gbps: ~300,000 packets/second
  - 100 Mbps: ~30,000 packets/second
- Even 9000-byte jumbo frames give us:
  - 1 Gbps: 14,000 packets per second → 14,000 interrupts/second

One interrupt per received packet

Network traffic can generate a LOT of interrupts!!

Interrupt Mitigation: Linux NAPI

- Linux NAPI: “New API” (c. 2009)
- Avoid getting thousands of interrupts per second
  - Disable network device interrupts during high traffic
  - Re-enable interrupts when there are no more packets
  - *Polling is better at high loads; interrupts are better at low loads*
- Throttle packets
  - If we get more packets than we can process, leave them in the network card’s buffer and let them get overwritten (same as dropping a packet)
    - Better to drop packets early than waste time processing them

The End