**Sockets**

- IP lets us send data between machines
- TCP & UDP are *transport layer* protocols
  - Contain *port number* to identify transport endpoint (application)
- The most popular abstraction for transport layer connectivity: **sockets**

**Socket**

- 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*
- Sockets is not just for the Internet Protocol family

**Connection-Oriented (TCP) socket operations**

- Create a socket
- Name the socket (assign local address, port)
- Set the socket for listening
- Wait for and accept a connection; get a socket for the connection
- read / write byte streams
- close the socket
- close the listening socket

**Connectionless (UDP) socket operations**

- Create a socket
- Name the socket (assign local address, port)
- Send a message
- Receive a message
- close the socket
Programming with sockets

Socket-related system calls

- Sockets are the interface the operating system provides for access to the network
- Next: a connection-oriented example

Step 1
Create a socket

```
int s = socket(domain, type, protocol)
```

- `AF_INET`: useful if some families have more than one protocol to support a given service
- Conceptually similar to open BUT
  - open creates a new reference to a possibly existing object
  - socket creates a new instance of an object

Step 2
Name the socket (assign address, port)

```
int error = bind(s, addr, addrlen)
```

- `struct sockaddr`: length of address structure

Step 3a (server)
Set socket to be able to accept connections

```
int error = listen(s, backlog)
```

- `queue length for pending connections`

Step 3b (server)
Wait for a connection from client

```
int snew = accept(s, clntaddr, &clntalen)
```

- `new socket for this session`
- `s is only used for managing the queue of connection requests`
Step 3 (client)

Connect to server

\[
\text{int error = connect(s, svraddr, svraddrlen)}
\]

socket address structure

length of address structure

Step 4

Exchange data

Connection-oriented I/O

read/write

recv/send (extra flags)

Connectionless I/O: no need for connect, listen, accept

sendto/receiver

sendmsg/receivmsg

Step 5

Close connection

\[
\text{shutdown(s, how)}
\]

how:
0: can send but not receive
1: cannot send more data
2: cannot send or receive (=0+1)

Socket Internals

Logical View

Data Path

Logical View

Socket layer

Transport layer

Network layer

Protocol stack queue

Network interface (driver) layer

Ethernet

Data Path

Application writes data to socket

Packet received by device

From device

Packet for host?

Forward packet?

Packet for network layer?

Forward packet?

Look up route

More packet to transport layer

More packet to socket

Data goes to application buffer

Transmit the packet to device

From app

To app
OS Network Stack

System call interface

- Two ways to communicate with the network:
  1. **Socket-specific call** (e.g., socket, bind, shutdown)
     - Directed to `sys_socketcall` (socket.c)
     - Goes to the target function
  2. **File call** (e.g., read, write, close)
     - File descriptor ≡ socket

- 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 to avoid resource exhaustion

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 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
    - `struct socket`: socket-related system calls
    - `struct inode`: points to in-memory inode associated with the socket
    - `struct sock`: protocol-specific (e.g., INET) 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.)
  - Identifies device structure (`net_device`)
    - `dev`: device that received the packet
    - `dev`: identifies network device on which the buffer operates
  - Each socket (connection stream) is associated with a linked list of `sk_buffs`

Keeping track of packet data

- Allocate new socket buffer data
  - `skb = alloc_skb(len, GFP_KERNEL)`
- No packet data: `head = data = tail`

Add or remove headers without reallocating memory

Allocate new socket buffer data

Allocate new socket buffer data
Keeping track of packet data

Make room for protocol headers:

- skb_reserve(skb, header_len)
- skb->sk->sk->proto->max_header
- Data size is still 0

Add user data

Add TCP header

Add IP header

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

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)
  - 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)
- 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
  - Moves packet to the device's xmit queue
- Network driver
  - Queues packet for transmission to the underlying driver

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 into 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

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 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**
- 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 (socket 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