

Operating Systems Design

14. Networking: Sockets

Paul Krzyzanowski
pxk@cs.rutgers.edu

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**
 - Developed at UC Berkeley

Sockets

Attempt at generalized IPC model

Goals:

- communication between processes should not depend on whether they are on the same machine
- efficiency
- compatibility
- support different protocols and naming conventions

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

Programming with sockets

Step 1

Create a socket

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

AF_INET

SOCK_STREAM
SOCK_DGRAM

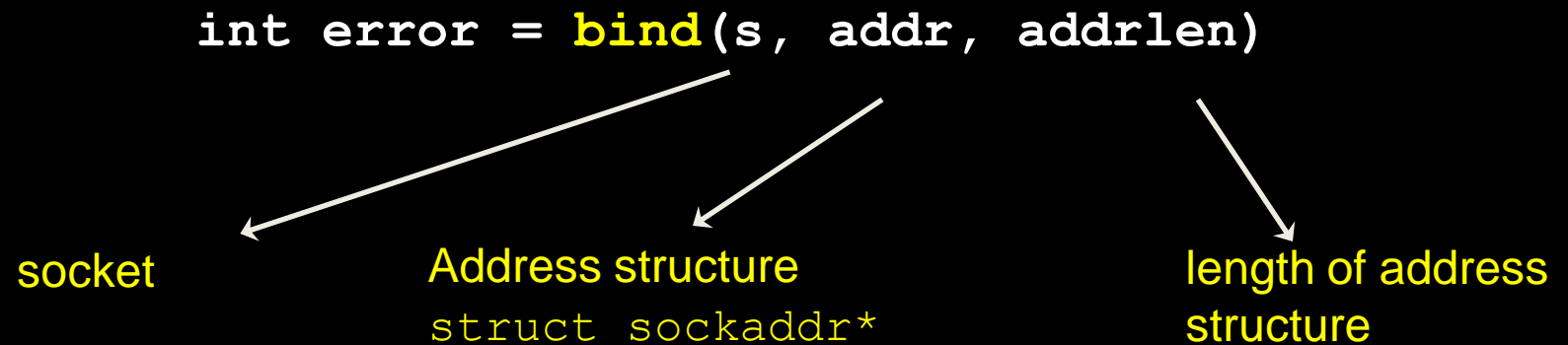
useful if some families
have more than one
protocol to support a
given service

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)



Step 3a (server)

Set socket to be able to accept connections

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

socket

queue length for pending
connections

Step 3b (server)

Wait for a connection from client

```
int snew = accept(s, cIntraddr, &cIntralen)
```

socket

pointer to address structure

length of address structure

new socket
for this session

s is only used for managing the queue of connection requests

Step 3 (client)

Connect to server

```
int error = connect(s, svraddr, svraddrlen)
```

socket

address structure
struct sockaddr*

length of address
structure

Step 4

Exchange data

Connection-oriented

read/write

recv/send (*extra flags*)

Connectionless

sendto/recvfrom

sendmsg/recvmsg

Step 5

Close connection

shutdown (*s*, *how*)

how:

0: can send but not receive

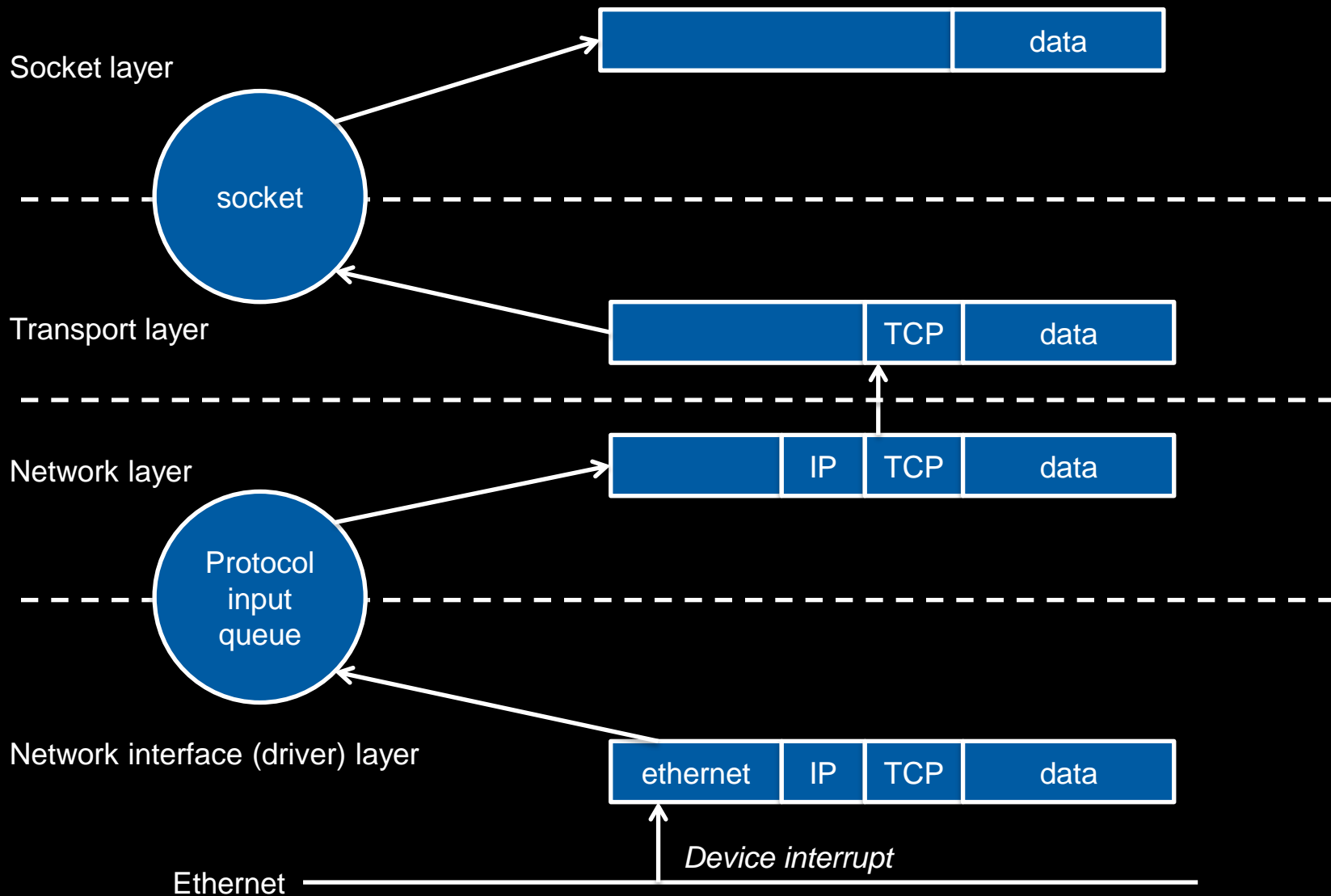
1: cannot send more data

2: cannot send or receive (=0+1)

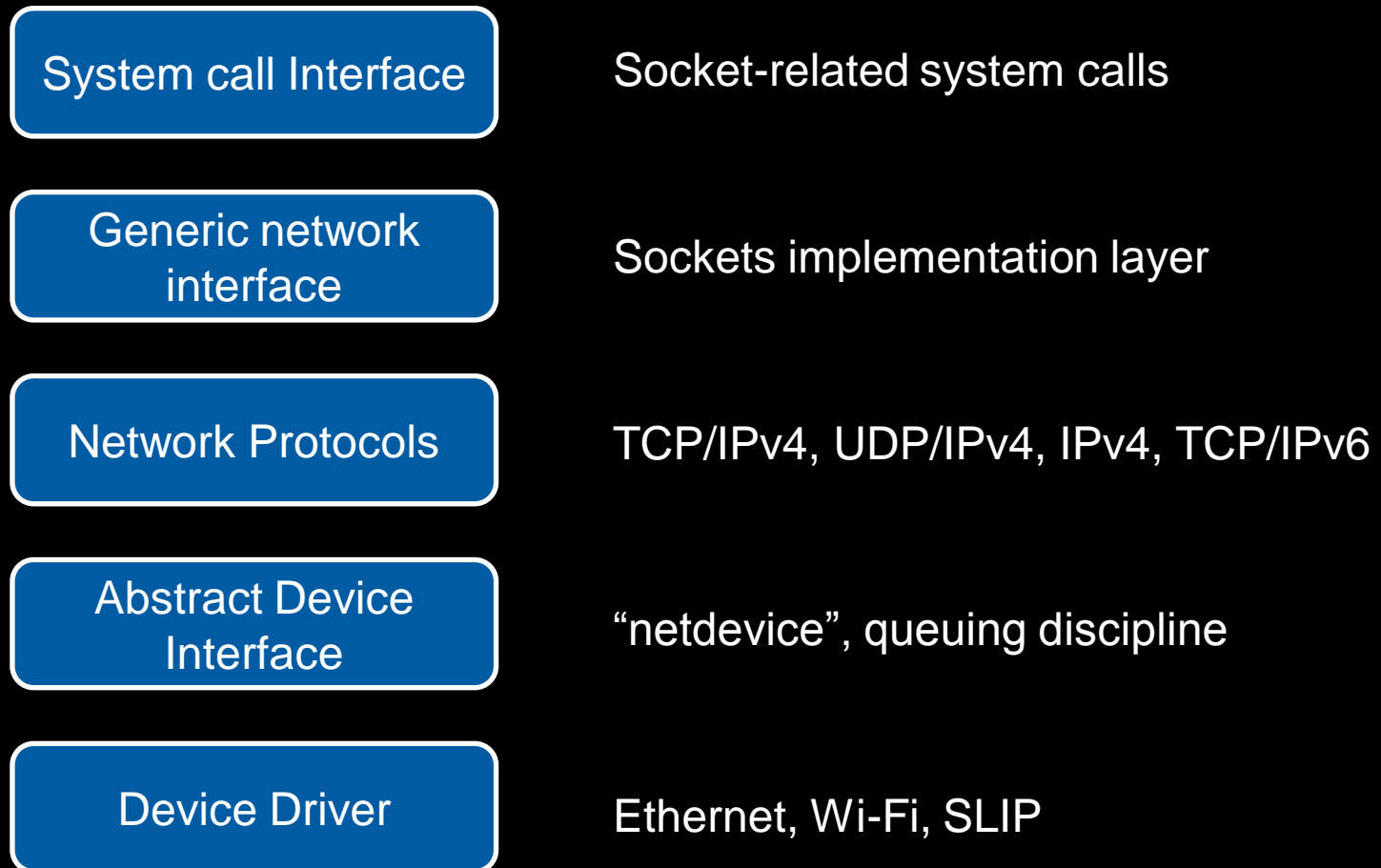
Socket Internals

Socket Internals

Logical View



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 descriptor call** (e.g., *read*, *write*, *close*)
 - **File descriptor \equiv 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 to avoid resource exhaustion

Sockets layer

- All network communication takes place via a socket
 - *socket* structure keeps all the state of a socket including the protocol and operations that can be performed on it
- Common functions to support a variety of protocols
 - TCP, UDP, IP, raw ethernet, other networks
- Each networking protocol has a structure called *proto*
 - Defines socket operations that can be performed from the sockets layer to the transport layer
 - *Create a socket, establish a connection with a socket, close a socket, ...*

Network protocols

- Define the specific protocols available (e.g., TCP, UDP)
- Modular: one module may define one or more protocols
- Initialized & registered at startup
 - *inet_init* function: registers the built-in IP family of protocols via *proto_register* function
 - The *register* function adds the protocol to the **active protocol list**
 - Optionally allocates caches
- Additional protocols can be added by calling *inet_register_proto_sw*

Network protocols: `sk_buff`: socket buffer

- Component for managing the data movement for sockets through the networking layers
 - Data stays in the buffer and does not move
 - Contains packet & state data for multiple layers of the protocol stack
- 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`

Abstract device interface

- Layer above 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, 802.11b/g/n, SLIP
- 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

Receiving a packet

- Device interrupt:
 - Allocate new `sk_buff`
 - Get data from the hardware buffer into the `sk_buff`
 - Call `netif_rx`, the generic network reception handler
 - This moves the `sk_buff` to protocol processing
 - When `netif_rx` returns, the service routine is finished
 - Or 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

- 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 handlers registered for the packet's protocol

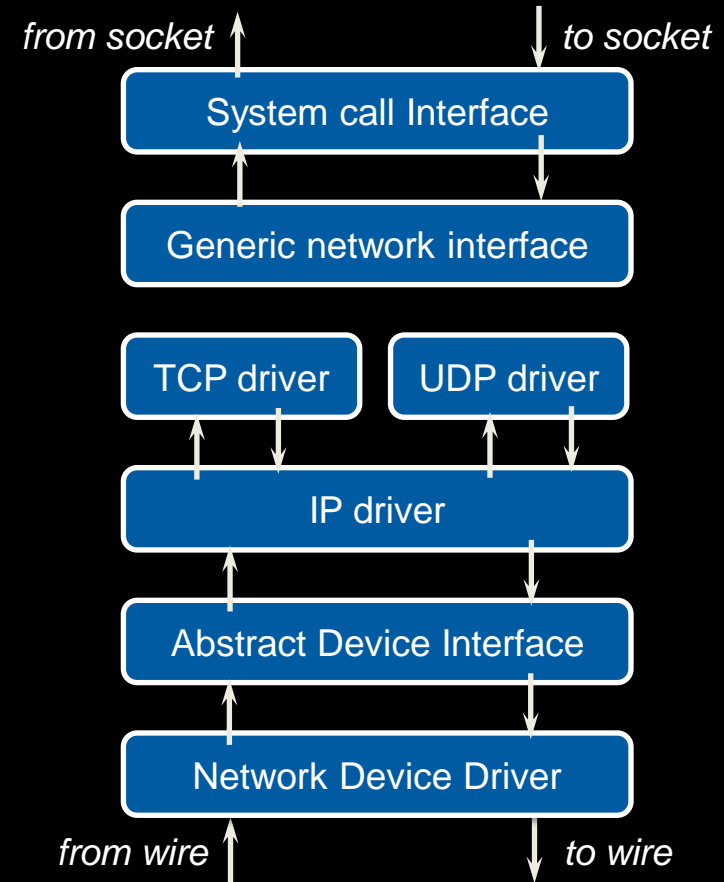
Receiving an IP packet

- IP is registered as a protocol handler for ETH_P_IP packets
 - IP handler will either route the packet, deliver locally, or discard
 - IP handler looks at protocol field inside the IP packet
 - Calls transport-level handlers (tcp_v4_rcv, udp_rcv, icmp_rcv, ...)
 - IP handler includes Netfilter hooks
- Next stage (usually): tcp_v4_rcv() or udp_rcv()
 - 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)
 - 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

Getting to the machine

IP is a logical network on top of multiple physical networks

OS support for IP: **IP driver**



IP driver responsibilities

- Get operating parameters from device driver
 - Maximum packet size (MTU)
 - Functions to initialize HW headers
 - Length of HW header
- Routing packets
 - From one physical network to another
- Fragmenting packets
- Send operations from higher-layers
- Receiving data from device driver
- Dropping bad/expired data

Device driver (e.g., ethernet) responsibilities

- Controls network interface card
 - Comparable to character driver
- Processes interrupts from network interface
 - Receive packets
 - Send them to IP driver
- Get packets from IP driver
 - Send them to hardware
 - Ensure packet goes out without collision
 - Follow the rules of Ethernet's CSMA/CD

Network device

- Network card examines packets on wire
 - Compares destination addresses
- Before a packet is sent, it must be **enveloped** for the physical network



Addressing the device

To send an IP packet, we need to know the *ethernet address* that corresponds to the *IP address* (or the ethernet address of the router that should get the packet)

Address Resolution Protocol (ARP)

Find the ethernet address for a given IP address:

1. Check local ARP cache
2. Send broadcast message requesting ethernet address of machine with certain IP address
3. Wait for response (with timeout)

Routing

Router

- Switching element that connects two or more transmission lines (e.g., Ethernet)
- Routes packets from one network to another (OSI layer 3 – Network Layer)
- Special-purpose hardware or a general-purpose computer with two or more network interfaces

Routing

- Packets take a series of **hops** to get to their destination
 - Figure out the path
- Generate/receive packet at machine
 - check destination
 - If destination = local address, deliver locally
 - else
 - Increment hop count (discard if hop # = TTL)
 - Use destination address to search **routing table**
 - Each entry has address and netmask. Match returns interface
 - Transmit to destination interface
- **Static routing**

Dynamic Routing

- Class of protocols by which machines can **adjust routing tables** to benefit from load changes and failures
- Route cost:
 - Hop count (# routers in the path)
 - Time: Tic count – time in 1/18 second intervals

The End