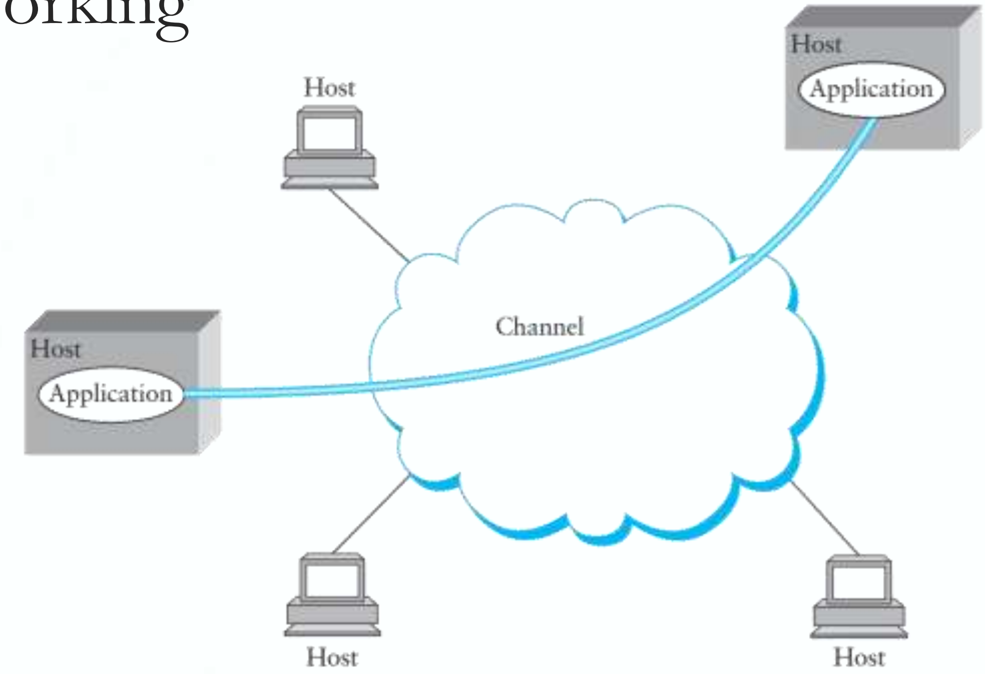
CS-446/646

Networking

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Networking Overview

Computer Networking



Goal

- Two Applications on different computers exchange Data
- Requires *Inter-Process* (not just *Inter-Node*) *Communication*

Networking Overview

The 7-Layer and 4-Layer Models

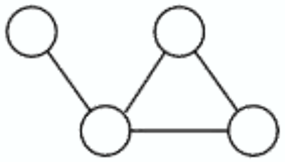
Open Systems Interconnection (OSI) Model	Transmission Control Protocol /Internet Protocol (TCP/IP) Model
Application	Applications (FTP, SMTP, HTTP, etc.)
Presentation	
Session	
Transport	TCP (Host-to-Host)
Network	IP
Data Link	Network access (usually Ethernet)
Physical	

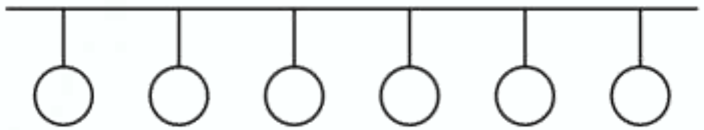
… also UDP, SCTP

Networking Overview

Physical Layer

- Computers send bits over Physical Links
  - e.g., Coax, Twisted-Pair, Fiber, Radio, …
  - Bits may be encoded as multiple lower-level "chips"
- Two categories of Physical Links
  - *Point-to-Point* Networks (e.g. Fiber, Twisted-Pair):



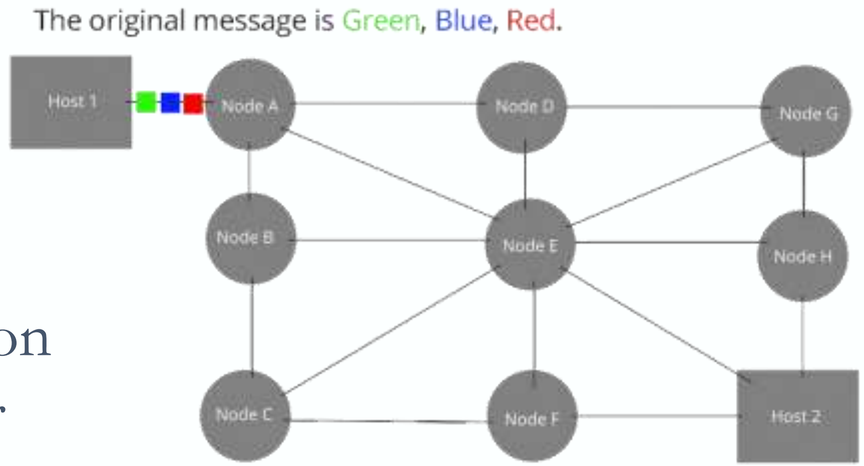
  - *Shared Transmission Medium* Networks (e.g. Coax, Radio):
    - 
    - Any message can be seen by all Nodes
    - Allows broadcast/multicast, but introduces contention
- One important constraint: Speed-of-Light
  - ~ 300, 000 km/sec in a vacuum, slower in Fiber
  - SF → NYC : ~15ms (Moore's law does not apply)

Networking Overview

Data Link Layer & Indirect Connectivity

- When no Direct Physical Connection to Destination exists
- Can Hop through multiple Devices

  - Allows Links and Devices to be shared for multiple purposes
  - Must determine which bits are part of which Messages intended for which Destinations

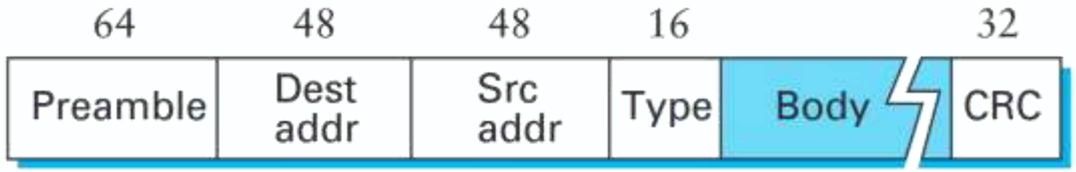


- *Packet-Switched* Networks
  - Pack Bytes together intended for a Destination
  - Append a *Packet Header* with intended Destination
  - Due to *Routing*, delivery can happen *Out-of-Order*

Networking Overview

Data Link Layer : *Ethernet*

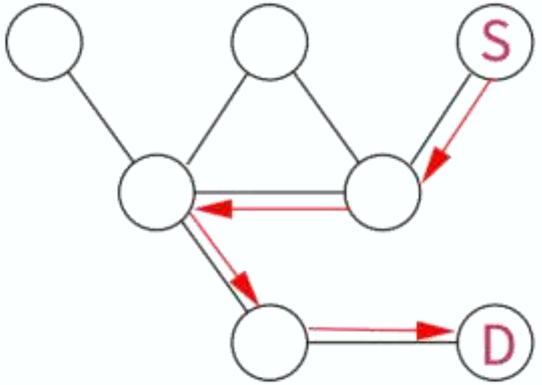
- Originally designed for shared medium (Coax), now generally not shared (Switched)
- Vendors give each Device a unique 48-bit MAC Address
  - Helps specify which card (Device) should receive a *Packet*
- *Ethernet* Switches can scale to switch *Local Area Networks (LAN)* (thousands of *Hosts*), but not much larger
- *Packet* format:



  - *Preamble* helps Device recognize start of *Packet*
  - CRC allows receiving card to ignore corrupted *Packets*
  - Body up to 1,500 Bytes for same Destination
  - All other fields must be set by sender's OS
    (NIC cards tell the OS what the card's MAC Address is,
    Special Addresses used for Broadcast/Multicast)

Network Layer: *Internet Protocol (IP)* Suite

- IP used to connect multiple Networks
  - Runs over a variety of Physical Networks
  - Hence can connect Ethernet, DSL, Mobile Networks, etc.
  - Most computers today speak IP
- Every host has a unique 4-Byte IP address (16-Bytes for IPv6)
  - (Or at least thinks it has, when there is Address shortage)
  - e.g. www.ietf.org → 104.20.0.85
- *Packets* are *Routed* based on Destination IP Address
  - Address space is structured to make *Routing* practical at global scale
    - e.g. `134.197.78.*` goes to UNR
  - Therefore *Packets* need IP Addresses as well as MAC Addresses



Transport Layer : *UDP* and *TCP*

- UDP and TCP most popular Transport-Layer Protocols on IP
  - Both use 16-bit Port Number as well as 32-bit IP Address
  - Applications *Bind* a Port & receive traffic to that Port

UDP – User Datagram Protocol

  - Exposes *Packet-Switched* nature of Internet
  - Sent *Packets* may be dropped, reordered, even duplicated (but generally not corrupted)

TCP – Transmission Control Protocol

- Provides illusion of a reliable "pipe" between two *Processes* on two different Machines
- Masks lost & reordered *Packets* so Apps don't have to worry
- Handles Congestion & Flow Control

Principles: ***Packet-Switching* & *Layering***

Packet-Switching

- A *Packet* is a self-contained unit of Data which contains information necessary for it to reach its intended Destination
- Independently, for each arriving *Packet*, compute its Outgoing Link
- Makes forwarding simple (depends only on information within *Packet*)

Layering

- Break up system functionality into a hierarchy of *Layers*
- Each *Layer* uses only the service of the *Layer* below it in the hierarchy
- *Layers* communicate sequentially with the *Layers* above or below them

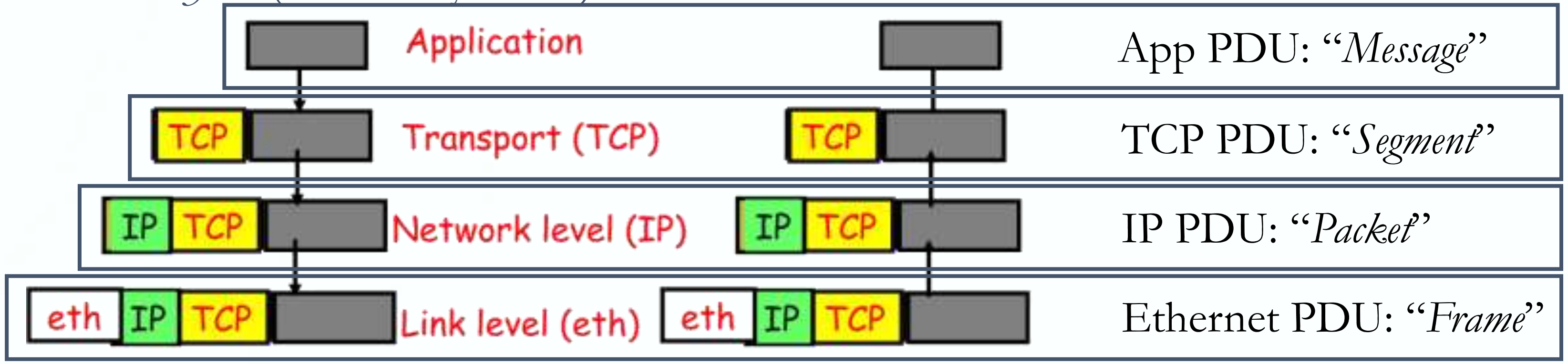
Networking Overview

Principle: *Encapsulation*

- Stick *Packets* inside *Packets*
- How *Packet-Switching* and *Layering* are realized in a system
  - e.g. an Ethernet *Packet* may encapsulate an IP *Packet*
  - An IP *Router* forwards a *Packet* from one Ethernet to another, creating a new Ethernet *Packet* containing the same IP *Packet*
  - In principle, an inner *Layer* should not depend on outer *Layers* (not always true)

Note:
Term "*Packet*" is somewhat loosely used sometimes.
More formal term:

- *Protocol Data Unit (PDU)*:
  A single unit of information transmitted among peer entities of a Network; composed of:
  a) **Protocol**-specific Control **Data**
  b) **User Data**



Unreliability of IP

- Network does not deliver *Packets* reliably
  - May drop *Packets*, reorder *Packets*, delay *Packets*
  - May even corrupt *Packets*, or duplicate them
- How to implement reliable TCP on top of IP Network?

Note: This is entirely the job of the OS that runs at the End-Nodes

Naïve Approach: Wait for *Ack* for each *Packet*

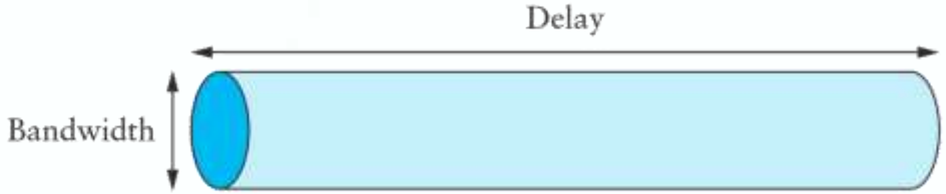
- Send a packet, wait for *Acknowledgment*, send next Packet
- If no *Ack*, Timeout and try again

Problems?

- Low performance over high-delay Network
  (Bandwidth is one *Packet* per *Round-Trip Time*)
- Possible "Congestive Collapse" of Network
  (e.g. everyone keeps retransmitting when Network is overloaded)

Performance: *Bandwidth-Delay*

- Network delay over *Wide Area Network (WAN)* will never improve much
- But *Throughput* (bits/s) is constantly improving
- Can view Network as a "pipe" :



  - For full Utilization want **# Bytes in flight** ≥ Bandwidth × Delay (But don't want to overload the Network, either)
- What if Protocol doesn't involve Bulk Transfer?
  - e.g. a "ping-pong" Protocol will have poor *Throughput*
- Another implication:
  - *Concurrency* & *Response Time* critical for good Network utilization

A little about TCP

- Want to save Network from "Congestive Collapse"
  - *Packet* loss usually means *Congestion*, so have to back off exponentially
- Also want multiple outstanding *Packets* at a time (improve Utilization)
  - Achieve a Transmit Rate up to n-*Packet* window per *Round-Trip*
- Must figure out appropriate value of n for Network
  - Slowly increase Transmission by one *Packet* per *Ack*-ed window
  - When a *Packet* is lost, cut window size in half
- Connection Setup and Teardown complicated
  - Sender never knows when last *Packet* might be lost
  - Must keep state around for a while after Connection `close()`
- Lots more hacks required for good performance
  - Initially ramp up value of n faster (but not too fast; caused collapse in 1986 [Jacobson], so TCP had to be changed)
  - Fast Retransmit when single *Packet* lost

Note:
Term "TCP *Packet*" is both informal and formal usage. In more precise terminology "*Segment*" refers to the TCP *Protocol Data Unit (PDU)*, "*Packet/Datagram*" to the IP PDU, and "*Frame*" to the *Data Link Layer* PDU. UDP uses *"Datagrams"* as its PDU for connectionless communication.

Lots of OS issues for TCP

- Have to track Un-*Ack(nowledg)*-ed Data
  - Keep a copy around until recipient *Acknowledges* it
  - Keep a Timer around to retransmit if no *Ack*
  - Receiver must keep out-of-order *Segment* & reassemble
- When to wakeup *Process* receiving Data?
  - e.g. sender calls **`write(fd, message, 8000);`**
  - First TCP *Segment* arrives, but is only 512 Bytes
  - Could wake receiving *Process*, but useless without full Message
  - TCP sets "push" bit at end of 8000 Byte write data
- When to immediately send short *Segment* –vs– Wait for more Data
  - Usually send only one Un-*Ack*-ed short *Segment*
  - But bad for some Apps, so provide **`NODELAY`** option
- Must *Ack* received *Segments* very quickly
  - Otherwise, effectively increases *Round Trip Time*, decreasing Bandwidth

OS Networking Facilities

Sockets

- Abstraction for Communication between Machines

1 – *Datagram Sockets* : Unreliable Message delivery

- With IP, gives us UDP
  - Send atomic messages, which may be reordered or lost
  - Special *System Calls* to read/write:
    `send*()` / `recv*()`

2 – *Stream Sockets* : Bi-Directional Pipes

- With IP, gives us TCP
  - Bytes written on one end read on the other
  - Reads may not return full amount requested → Must re-read

OS Networking Facilities

Socket Naming

- TCP & UDP name Communication Endpoints by using:
  - 32-bit **IPv4 Address** specifies Machine (128-bits for **IPv6 Address**)
  - 16-bit **TCP/UDP Port** number demultiplexes within Host
- A "*Connection*" is thus defined by a "*5-Tuple*":
  - *1)* Protocol (TCP/UDP), *2)* Local IP, *3)* Local Port, *4)* Remote IP, *5)* Remote Port
    - TCP requires *Connected Sockets*, UDP does not (but *Connected Mode* UDP exists)
- OS keeps *Connection* State in *Protocol Control Block* (PCB) structure
  - Keep all PCBs in a Hash Table
  - When *Packet* arrives (if Destination IP Address belongs to Host), use its *5-Tuple* to find corresponding PCB and determine what to do with *Packet*

System Calls for using TCP

Client

Create a *Socket*

```
sockfd = socket(domain, type, protocol);
```

Assign it an Address *

```
bind(sockfd, sockaddr, addrlen);
```

Connect to listening *Socket*

```
connect(sockfd, sockaddr, addrlen);
```

* This call to `bind()` is optional;
`connect()` can bind to all *Interfaces* and pick some high-numbered *Port*

Server

Create a *Socket*

```
sockfd = socket(domain, type, protocol);
```

Assign it an Address

```
bind(sockfd, sockaddr, addrlen);
```

Mark *Socket* as passive – i.e. listening for Clients

```
listen(sockfd, backlog);
```

Accept Connection (can *Block* if none pending on *Queue*)

```
accept(sockfd, sockaddr, addrlen);
```

Using UDP

- Use `socket()` *System Call* with `type = SOCK_DGRAM`; then `bind()` as before

New *System Calls* for sending individual Packets:

- `numbytes = sendto(sockfd, msgbuf, len, flags, sockaddr_dest, addrlen);`
- `numbytes = recvfrom(sockfd, msgbuf, len, flags, sockaddr_from, addrlen);`

Note: Must send/get Peer Address with each *Packet*

- Can also use UDP in *Connected Mode*
  - `connect()` assigns remote Address
  - Use `send()` / `recv()` *System Calls*
    - like `sendto()` /`recvfrom()` without last 2 arguments

Uses of *Connected Mode* UDP *Sockets*

- Kernel demultiplexes *Packets* based on *Port*
  - Allows different *Processes* to get *Packets* from different Peers
  - For security, *Ports* [0, 1023] are considered *Privileged* (e.g. *Port* 80 – for *HTTP*)
    - Usually can't be bound by non-**`root`** *Processes*
      - Unless e.g. the *Process* is granted the **`CAP_NET_BIND_SERVICE`** *Capability*
    - So, services have to be **initiated** by **`root`**, but can then safely **inherit** *Privileged* UDP *Port* that is already *Connected* (*Connected Mode*) to a particular Peer

- Feedback based on *Internet Control Message Protocol* (ICMP) Messages
  - Assuming no *Process* has successfully bound the UDP *Port* you sent *Packet* to:
    - With **`sendto()`**, you might think Network is dropping *Packets*
    - Server sends back ICMP "*Port Unreachable*" Message, but can only detect it when using *Connected Mode* UDP *Sockets*

Implementing Networking in the Kernel

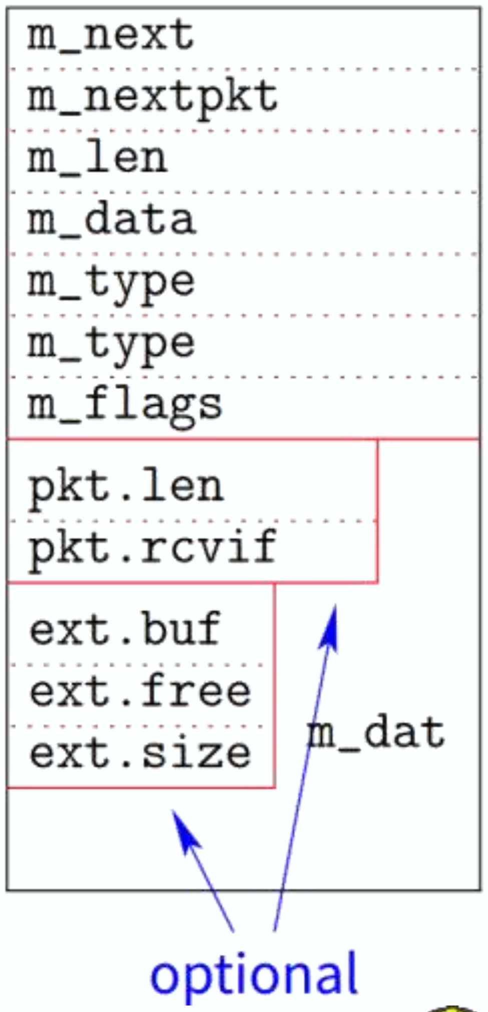
Socket Implementation

- Need to implement *Layering* efficiently
  - Add UDP *Header* to *Message*, Add IP *Header* to UDP *Datagram*, etc.
  - De-encapsulate Ethernet *Packet* so IP-handling code doesn't get confused by Ethernet *Header*
- Don't store entire *Packets* contiguously in *Memory*
  - Moving Data to make room for new *Header* would be slow
- BSD solution: **`mbufs`** [Leffler] (Note: [Leffler] calls **`m_nextpkt`** by old name **`m_act`**)
  - Small, fixed-size (256-Byte) **`struct`**s
  - Makes allocation/deallocation easy (no *Fragmentation*)
- BSD **`mbufs`** working example in this Lecture
  - Linux uses **`sk_buffs`**, which are similar idea

Implementing Networking in the Kernel

`mbuf` Details

- *Packets* made up of multiple **`mbuf`**s
  - *Chained* together by **`m_next`**
  - Such linked **`mbuf`**s called *Chains*
- *Chains* linked with **`m_nextpkt`**
  - Linked *Chains* known as *Queues*
  - e.g. *Device Output Queue*
- Total **`mbuf`** size 256 Bytes ⇒ ~230 Bytes of Data – in **`m_dat`** (depends on size of Pointers)
  - First **`mbuf`** in *Chain* ("*Head*" of *Chain*) holds the *Packet Header*
- *Cluster* **`mbuf`**s have more Data
  - **`ext`** *Header* points to Data
  - Up to 2 KB non-collocated with **`mbuf`**
    - **`m_dat`** unused in this case
- **`m_flags`** is bitwise OR of various bits
  - e.g. if *Cluster*, or if *Packet Header* used



Adding/Deleting Data with `mbuf`s

- `m_data` is used to point to the **start of Data**
  - Can either be `m_dat`, or `ext.buf` for a *Cluster* `mbuf`
  - But also: Can even point into middle of that area
    - Flexible handling of stripping away a front chunk from the Data
- To strip off a *Packet Header* (e.g. TCP/IP)
  - Increment `m_data`, decrement `m_len`
- To strip off end of *Packet*
  - Decrement `m_len`
- Can add more data to `mbuf` if its Data buffer is not already full
- Otherwise, can append more data to the *Chain*
  - Chain a new `mbuf` at *Head*/*Tail* of existing *Chain*

mbuf Utility Functions

```
mbuf * m_copym(mbuf * m, int off, int len, int wait);
```

- Creates a copy of a subset of a **mbuf** *Chain*
- Doesn't copy *Clusters*, just increments reference-counting
- **wait** says what to do if no *Memory* (wait or return NULL)

```
void m_adj(struct mbuf * mp, int len);
```

- Trim **len** Bytes from *Head* or (if negative) *Tail* of *Chain*

```
mbuf * m_pullup(struct mbuf * m, int len);
```

- Put first **len** Bytes existing in a **mbuf** *Chain* (starting at **m**) contiguously into the Data area of the *Head* **mbuf** structure (returns the *Head* **mbuf** of the *Chain* – now altered)
- *Example* with *Layering*: An Ethernet *Frame* containing an IP *Packet*
  - Trim Ethernet *Header* using **m_adj()**
  - Call **m_pullup(n, sizeof (ip_hdr));**
  - Access IP *Header* as regular C data **struct** (now Contiguous)

Implementing Networking in the Kernel

Socket Implementation

- Each *Socket* **`fd`** has associated *Socket* structure with:
  - Send and Receive Buffers
  - *Queues* of incoming *Connections* (on Listen *Socket*)
  - A *Protocol Control Block (PCB)*
  - A *Protocol Handle* (**`struct protosw *`**)
- *Protocol Control Block* (PCB) contains Protocol-specific info

  e.g. for TCP:
  - A *5-Tuple* (Protocol (TCP), Source IP Address & Port, Destination IP Address & Port)
  - Information about received *Packets* & position in stream
  - Information about Un-*Ack*-nowledged sent *Packets*
  - Information about Timeouts
  - Information about Connection State (Setup/Teardown)

Implementing Networking in the Kernel

`protosw` Structure

- Goal: abstract away differences between Protocols
  - In C++, might use virtual functions on a generic *Socket* struct
  - In C, put function Pointers in **`protosw`** structure
- Also includes a few data fields
  - **`type`**, **`domain`**, **`protocol`**: To match **`socket()`** args, to know which **`protosw`** to select
  - **`flags`**: To specify important properties of Protocol
- Some Protocol **`flags`**:
  - **`atomic`**: Exchange atomic Messages only (like UDP, not TCP)
  - **`addr`**: Address given with Messages (like Unconnected UDP)
  - **`connrequired`**: Requires Connection (like TCP)
  - **`wantrcvd`**: Notify *Socket* of consumed Data (e.g. so TCP can wake-up a sending *Process* blocked by Flow Control)

Implementing Networking in the Kernel

`protosw` Functions (Function Pointers)

- **`pr_slowtimo`** – Called every 1/2 sec for Timeout processing
- **`pr_drain`** – Called when system low on space
- **`pr_input`** – Returns **`mbuf`** *Chain* of Data read from *Socket*
- **`pr_output`** – Takes **`mbuf`** *Chain* of Data written to *Socket*
- **`pr_usrreq`** – Multi-purpose user-request hook
  - Used for bind/listen/accept/connect/disconnect operations
  - Used for out-of-band Data

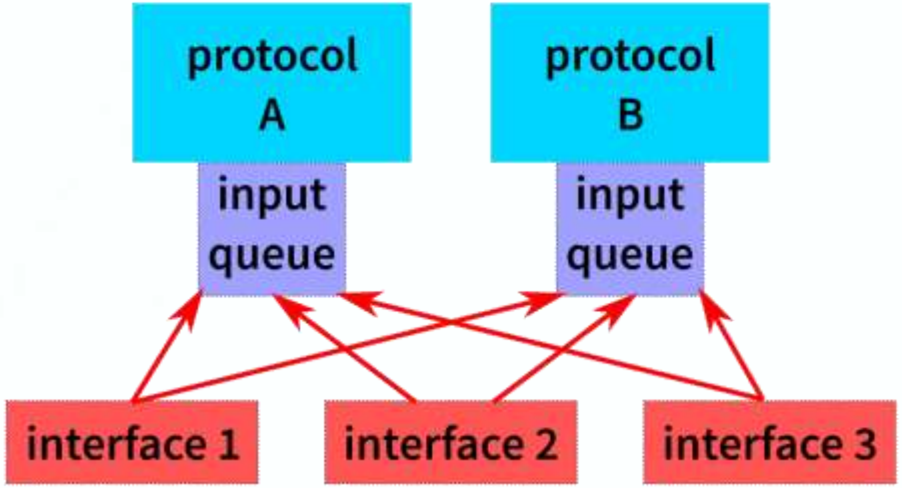
Implementing Networking in the Kernel

Network Interface Cards (NICs)

- Each NIC Driver provides an **`ifnet`** data structure
  - Like **`protosw`**, tries to abstract away the details
- Data fields:
  - *Interface* Name (e.g. **`eth0`**)
  - Address List (e.g. Ethernet Address, Broadcast Address, ...)
  - Maximum *Packet* Size
  - *Send Queue*
- Functions (Function Pointers):
  - **`if_output`** – To prepend *Header* and enqueue *Packet*
  - **`if_start`** – To start transmitting Queued *Packets*
  - Also **`ioctl`**, **`timeout`**, **`initialize`**, **`reset`**

Implementing Networking in the Kernel

Input Handling



- NIC Driver figures out Protocol from incoming *Packet*
- Enqueues *Packet* for appropriate *Protocol Handler*
  - If *Queue* full, drop *Packet* (can create "Livelock" [Mogul])
- Posts "*Soft Interrupt*" for Protocol-Layer processing
  - Runs at lower Priority than Hardware (NIC) *Interrupt*
  - … but at higher Priority than *Process-Context* Kernel code

Implementing Networking in the Kernel

Routing

- An OS must *Route* all transmitted *Packets*
  - Destination Machine may have multiple NICs plus **`loopback`** *Interface*
  - Which *Interface* should a *Packet* be sent to, and what MAC Address should *Packet* have?

Note: Addressing (requires IP & MAC Address – discovered through ARP) is not the same as *Routing*

- *Routing* is based purely on the Destination IP Address in the IP *Header* of the *Packet*
  - Even if Host has multiple NICs with different MAC Addresses
    - OSI model adds a *Layer* to allow for path discovery to the next gateway; this *Layer* is responsible for *Routing*, but knows nothing about the MAC Address
  - OS maintains "*Routing Table*": Maps IP Address & Prefix-length → Next Hop
    - Uses *Radix Tree* for efficient Lookup:
      - Branch at each node in Tree based on single bit of Target
      - When a leaf node is reached, that is your Next Hop
- Most OSes provide *Packet Forwarding*
  - Received *Packets* intended for a non-local Address get *Routed* out another *Interface*

Network File Systems

Network File System (NFS)

- Looks like a *Filesystem* (e.g. FFS) to Applications
  - But Data potentially stored on another Machine
  - Reads and writes must go over the Network
  - Also called "*Distributed Filesystems*"

Advantages:

- Easy to Share if *Files* available on multiple Machines
- Often easier to administer Servers than Clients
- Access way more data than fits on your local Disk
- Network + Remote-buffer Cache can be faster than local Disk

Disadvantages

- Network + Remote Disk slower than local Disk
- Network or Server may fail even when Client OK
- Complexity, *Security* issues

Network File Systems

NFS version 2 [Sandberg]

- Background: Sun *Network Disk* (ND) Protocol (specified in [RFC 1050])
  - Creates Disk-like Device even on Diskless Workstations
  - Can create a regular (e.g. FFS) *Filesystem* on it
  - But no *File* Sharing – Why?
    - FFS assumes Disk doesn't change under it
- ND idea still used today by Linux *Network Block Device* (NBD)
  - Useful for Network Booting/Diskless Machines
  - But not for *File* Sharing
- *Network File System* (NFS) – goals (NFS v2 Protocol specified in [RFC 1094])
  - *File* Sharing: Access same *Filesystem* from multiple Machines simultaneously
  - Maintain Unix semantics
  - Crash recovery
  - Competitive performance with ND Protocol

NFS Implementation

- *Virtualized* the *Filesystem* with **`vnode`**s
  - *Remember*: Poor man's C++ (like **`struct protosw`**)

- **`vnode`** structure represents an open (or openable) *File*
- Defines a generic set of "**`vnode`** operations" (i.e. contains Function Pointers):
  - **`lookup`**, **`create`**, **`open`**, **`close`**, **`getattr`**, **`setattr`**, **`read`**, **`write`**, **`fsync`**, **`remove`**, **`link`**, **`rename`**, **`mkdir`**, **`rmdir`**, **`symlink`**, **`readdir`**, **`readlink`**, …
  - Called through Function Pointers, so most *System Calls* don't care what type of *Filesystem* a *File* resides on
- NFS **`vnode`** operations perform *Remote Procedure Calls* (RPC)
  - Client sends *Request* to Server over Network, awaits *Response*
  - Each *Filesystem*-related *System Call* may require the execution of a series of RPCs
  - System mostly determined by RPC [RFC 1831] Protocol
  - Uses *eXternal Data Representation (XDR)* language [RFC 1832]

Stateless Operation

- Designed for "*Stateless* Operation"
  - Motivated by need to recover from Server Crashes
- *Requests* are self-contained
- *Requests* are **mostly** *Idempotent* (no matter how many times executed, result is the same)
  - Unreliable UDP Transport
  - Client retransmits *Requests* until it gets a reply
  - Writes must be stable before Server returns
- Why **mostly**?
  - Of course, *Filesystem* is not stateless – It stores *Files*
    - e.g. **`mkdir`** can't be *Idempotent* – 2nd time performed, the *Directory* already exists
    - But many operations, e.g. **`read`**, **`write`** are *Idempotent*

NFS version 3

- Same general architecture as NFS v2
- Specified in [RFC 1813] (subset of Open Group spec)
  - XDR defines C structures that can be sent over Network; includes *Tagged* **`union`***s* (to know which **`union`** field is active)
  - Protocol defined as a set of *Remote Procedure Calls* (RPCs)
- New RPC for Access
  - Supports Clients and Servers with different **`uid`**s/**`gid`**s
- Better support for Caching
  - Unstable writes while data still Cached at Client
  - More information for Cache *Consistency*
- Better support for exclusive *File* creation

NFS v3 *File Handles*

```
struct nfs_fh3 {
  /* XDR notation for variable-length array with 0-64 opaque bytes: */
  opaque data<NFS3_FHSIZE>;  /* NFS3_FHSIZE defined as 64 */
};
```

- Server assigns an opaque *File Handle* to each *File*
  - Client obtains 1st *File Handle "Out-Of-Band (OOB)"* – Serarate *Mount* Protocol
    - *File Handle* hard to guess – *Security* enforced at *Mount* Time
  - But any subsequent *File Handles* obtained through Lookups
- *File Handle* internally specifies *Filesystem* & *File*
  - Device number, *i-number*, generation number, ...
  - Generation number changes when *inode* recycled
- *File Handle* generally doesn't contain *Filename*
  - Clients may keep accessing an open *File* even after it has been renamed

File Attributes

- Most operations can optionally return **fattr3**
- Contains *Attributes* used for Cache-Consistency

```
struct fattr3 {
  ftype3 type;
  uint32 mode;
  uint32 nlink;
  uint32 uid;
  uint32 gid;
  uint64 size;
  uint64 used;
  specdata3 rdev;
  uint64 fsid;
  uint64 fileid;
  nfstime3 atime;
  nfstime3 mtime;
  nfstime3 ctime;
};
```

Note (RPC *Data Description Language*):

```
struct post_op_attr {
  bool_t attributes_follow;
  union {
    fattr3 attributes;
  } post_op_attr_u;
};
```

```
union post_op_attr switch (bool attributes_follow) {
  case TRUE:
    fattr3 attributes;
  case FALSE:
    void;
};
```

Lookup

- RPC Procedure: `LOOKUP3res NFSPROC3_LOOKUP(LOOKUP3args);`

Need this: →

```
struct LOOKUP3args {
  diropargs3 what;
};
```

```
struct diropargs3 {
  nfs_fh3 dir;
  filename3 name;
};
```

```
union LOOKUP3res switch (nfsstat3 status) {
  case NFS3_OK:
    LOOKUP3resok resok;
  default:
    LOOKUP3resfail resfail;
};
```

```
struct LOOKUP3resok {
  nfs_fh3 object;
  post_op_attr obj_attributes;
  post_op_attr dir_attributes;
};
struct LOOKUP3resfail {
  post_op_attr dir_attributes;
};
```

- Maps < *Directory Handle* (`dir`), *Filename* (`name`) > → *File Handle* (`object`)
  - Client walks Hierarchy one *File* at a time
  - No *Symlinks* or *Filesystem* boundaries crossed
  - Client must expand *Symlinks*

Create

- RPC Procedure: **`CREATE3res NFSPROC3_CREATE(CREATE3args);`**

```
struct CREATE3args {
  diropargs3 where;
  createhow3 how;
};
```

Need this too:

```
union createhow3 switch (createmode3 mode) {
  case UNCHECKED:
  case GUARDED:
    sattr3 obj_attributes;
  case EXCLUSIVE:
    createverf3 verf;
};
```

- **UNCHECKED** – Succeed even if duplicate *File* exists
- **GUARDED** – Check if duplicate *File* exists; Fail in this case
- **EXCLUSIVE** – Persistent record of **CREATE** (instead of Client providing just *Attributes*, it directly provides a *Verifier* that is recorded in file attributes; if same Client re-attempts **EXCLUSIVE CREATE**, Server returns success, if another Client attempts **EXCLUSIVE CREATE** around same time, its *Verifier* won't match what's recorded in file, Server returns **NFS3ERR_EXIST**)

```
enum createmode3 {
  UNCHECKED = 0,
  GUARDED = 1,
  EXCLUSIVE = 2
};
```

Read

- RPC Procedure: **`READ3res NFSPROC3_READ(READ3args);`**

```
struct READ3args {
  nfs_fh3 file;
  uint64 offset;
  uint32 count;
};

union READ3res switch (nfsstat3 status) {
  case NFS3_OK:
    READ3resok resok;
  default:
    READ3resfail resfail;
};
```

```
struct READ3resok {
  post_op_attr file_attributes;
  uint32 count;
  bool eof;
  opaque data<>;
};

struct READ3resfail {
  post_op_attr file_attributes;
};
```

- **`offset`** explicitly specified (not implicit in *File Handle* – Remember: *Stateless* Design)
- Client can Cache result

Data Caching

- Client can Cache *File* Blocks of Data read and written *Consistency* based on times in `fattr3`
- **`nfstime3 mtime`**: Time of last modification to *File*
- **`nfstime3 ctime`**: Time of last change to *inode*
  (Changed by explicitly setting **`mtime`**, increasing size of *File*, changing *Permissions*, etc.)

Algorithm:

- If our Cached values for **`mtime`** or **`ctime`** changed (e.g. by another Client), flush Cached *File* Blocks

Write Discussion

When is it okay to lose Data after a Crash?

- Local *Filesystem*?
  - If no calls to **`fsync()`**, OK to lose 30 seconds of work after Crash
- *Network File System*?
  - What if the Server Crashes, but not the Client?
  - Application not killed, so we shouldn't have to lose previous writes
- NFS v2 addresses problem by having Server first complete the writing of Data to Disk, before replying to write RPC → Caused performance problems
- Could NFS v2 Clients just perform *Write-Back*?
  - Implementation issues – Used *Blocking* Kernel *Threads* on **`write()`**
  - Semantics – How to guarantee Consistency after Server Crash
  - Solution: Try to minimize # of pending write RPCs, but *Write-Through* on **`close()`**; if Server crashes, Client keeps re-writing until *Ack*-ed

NFS v2 *Write*

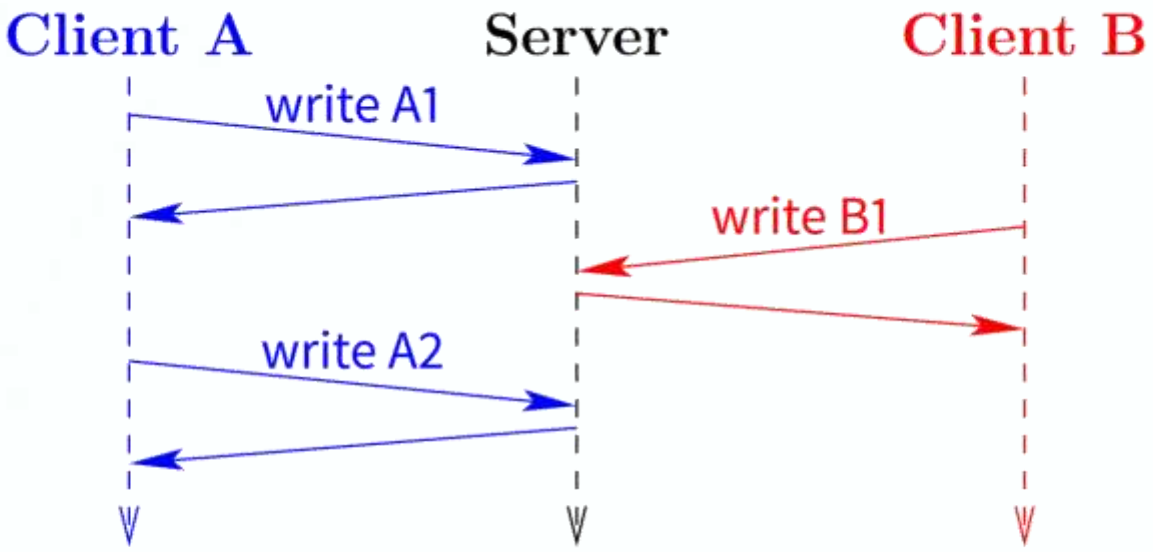
- RPC Procedure: **`attrstat NFSPROC_WRITE(writeargs);`**

Need this: →

```
struct writeargs {
  fhandle file;
  unsigned beginoffset;
  unsigned offset;
  unsigned totalcount;
  opaque data<NFS_MAXDATA>;
};
union attrstat switch (stat status) {
  case NFS_OK:
    fattr attributes;
  default:
    void;
};
```

- On successful **`WRITE`**, returns new *File Attributes* (as **`fattr`**)
- Can NFS v2 keep a Cached copy of the *File* after writing it?

Write Race Condition



- Suppose Client overwrites a 2-Block *File*
  - Client A knows *File Attributes* after **`write()`** A1 & **`write()`** A2
  - But client B could overwrite Block 1 inbetween A1 & A2
  - No way for Client A to know this hasn't happened
  - Must flush Cache before next *File* **`read()`** (or at least **`open()`**)

NFS v3 *Write*

- RPC Procedure: `WRITE3res NFSPROC3_WRITE(WRITE3args);`

```
struct WRITE3args {
  nfs_fh3 file;
  uint64 offset;
  uint32 count;
  stable_how stable;
  opaque data<>;
};
```

Need this too:

```
enum stable_how {
  UNSTABLE = 0,
  DATA_SYNC = 1,
  FILE_SYNC = 2
};
```

```
struct WRITE3resok {
  wcc_data file_wcc;
  count3 count;
  stable_how committed;
  writeverf3 verf;
};
struct WRITE3resfail {
  wcc_data file_wcc;
};

union WRITE3res switch (nfsstat3 status)
{
  case NFS3_OK:
    WRITE3resok resok;
  default:
    WRITE3resfail resfail;
};
```

Two goals for NFS v3 write:

- Don’t force Clients to flush Cache after writes
- Don’t equate Cache Consistency with Crash Consistency
  - i.e. don’t wait for Disk just so another Client can see Data

NFS v3 *Write* Results

- RPC Procedure: `WRITE3res NFSPROC3_WRITE(WRITE3args);`

```
struct wcc_attr {
  uint64 size;
  nfstime3 mtime;
  nfstime3 ctime;
};
```

```
struct wcc_data {
wcc_attr * before;
post_op_attr after;
};
```

```
struct WRITE3resok {
  wcc_data file_wcc;
  count3 count;
  stable_how committed;
  writeverf3 verf;
};
struct WRITE3resfail {
  wcc_data file_wcc;
};
```

- Write Results
  - Several fields added to achieve these goals

```
union WRITE3res switch (nfsstat3 status)
{
  case NFS3_OK:
    WRITE3resok resok;
  default:
    WRITE3resfail resfail;
};
```

Data Caching after a Write

- **`WRITE`** will change **`mtime`**/**`ctime`** of a *File*
  - Field **`after`** (type: **`post_op_attr`**) will contain new times
  - Like with NFS v2, but relying on just this information required flushing the Cache
- With NFS v3, field **`before`** (type: **`wcc_attr`**) additionally contains previous values before this **`WRITE`**
  - If **`before`** matches our Cached values, no other Client has gotten to change the *File* inbetween
  - OK to update *Attributes* without flushing Data Cache

Write Stability

- Server write must be at least as **`stable`** (type: **`stable_how`**)as requested
- If Server returns write **`UNSTABLE`**:
  - Means *Permissions* okay, enough free Disk space
  - But Data not on Disk and might disappear (e.g. after a Crash)
- If Server returns **`DATA_SYNC`**:
  - Data on Disk, maybe not *Attributes*
- If Server returns **`FILE_SYNC`**:
  - Operation complete and stable

Network File Systems

Commit Operation

- Client cannot discard any **`UNSTABLE`** write
  - If Server crashes, Data will be lost
- RPC Procedure: **`COMMIT3res NFSPROC3_COMMIT(COMMIT3args);`**
  - Provides synchronization mechanism to be used with asynchronous **`WRITE`** operations
- **`NSFPROC3_COMMIT`** RPC commits a range of a *File* to Disk
  - Invoked by Client when Client cleaning buffer Cache
  - Invoked by Client when User closes/flushes a *File*
- How does Client know if Server crashed?
  - **`WRITE`** and **`COMMIT`** RPCs return: **`writeverf3 WRITE3res.verf`**
  - Value changes after each Server Crash (e.g. can be the Boot Time)
  - Client must resend all **`UNSTABLE`** writes if **`verf`** value changes

Attribute Caching

- NFS has no **`OPEN`** or **`CLOSE`** operations (i.e. RPCs)
- "*Close-to-Open*" Cache *Consistency*
  - Client should not maintain a Cached *File* version after **`close()`** (e.g to re-**`open()`**)
  - Client responsible to **`WRITE/COMMIT`** any changes on **`close()`**
  - Client responsible to **`GETATTR/ACCESS`** on **`open()`**s to fetch *Attributes* from Server

Note: No **`OPEN/CLOSE`** RPCs, NFS is *Stateless*

- However, we can have lots of other needs for Cached *Attributes* (e.g. **`ls -al`**)
  - *Attributes* Cached between 5 and 60 seconds
    - *Files* recently changed more likely to change again
    - Do *Weighted Cache Expiration* based on age of *File*
- Drawbacks of combined a) *"Close-to-Open"* Cache *Consistency* & b) *Attribute* Caching:
  - Must pay for *Round-Trip* to Server on every *File* **`open()`**
  - Can get stale info when **`stat`**-ting a *File*

CS-446/646

Time for Questions !