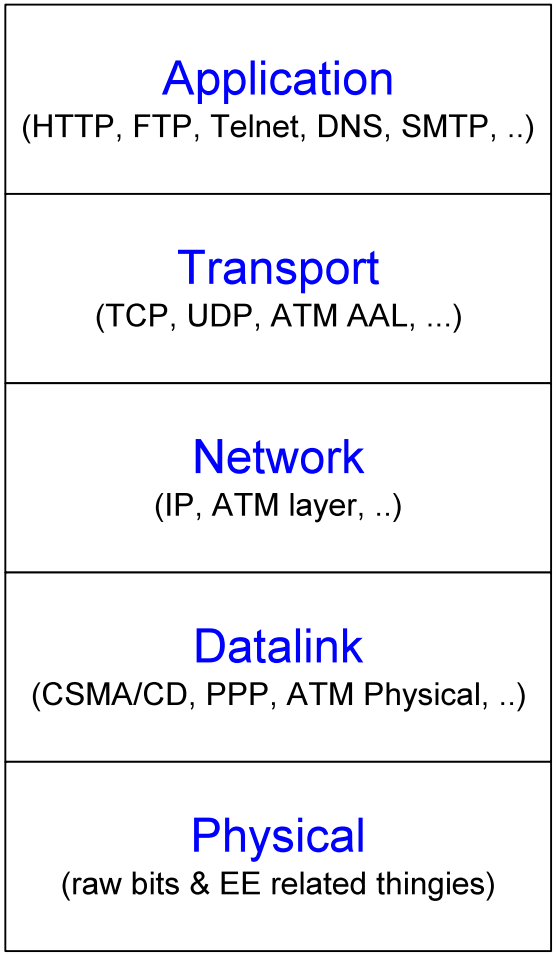
Last Lecture

- Introduction to Networking and the Internet
- Protocol Architecture

This Lecture

- Elementary BSD socket API for network programming in **C** under Unix
- Client/server design alternatives
- (*No time for*) Some brief mentioning of several advanced features
  - Multicasting/broadcasting, Socket options, IPv4/v6, interoperability, Daemon processes, Raw sockets, Out of band data

TCP/IP Protocol Suite – A Reminder



Supports Network Applications

Transports applications' messages
TCP: connection-oriented, reliable
UDP: connectionless, unreliable

Routes data packets from hosts to hosts
IP: Internet Protocol, and many routing protocols

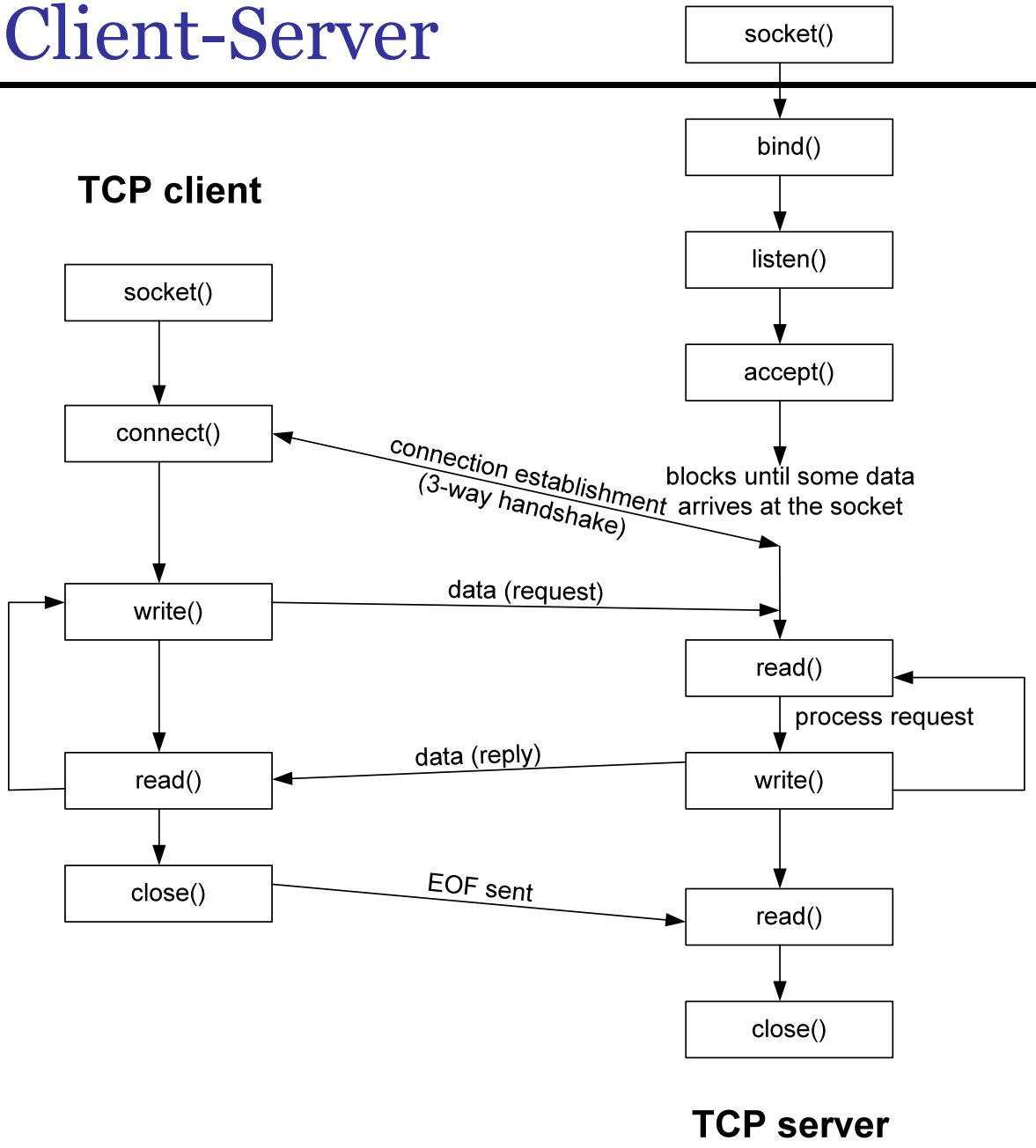
Deals with algorithms to achieve reliable, efficient communication between two adjacent machines

Moves raw bits (0/1) between adjacent nodes depending on the physical medium used

TCP Overview

- Establish connection: 3-way handshake
- Data transmission
  - Byte-stream service
  - Reliable (retransmission with timer)
  - In-order delivery (reorder packets if necessary)
  - Support flow control (fast sender vs slow receiver)
  - Full-duplex (data transferred both ways)
- Close connection

Typical TCP Client-Server

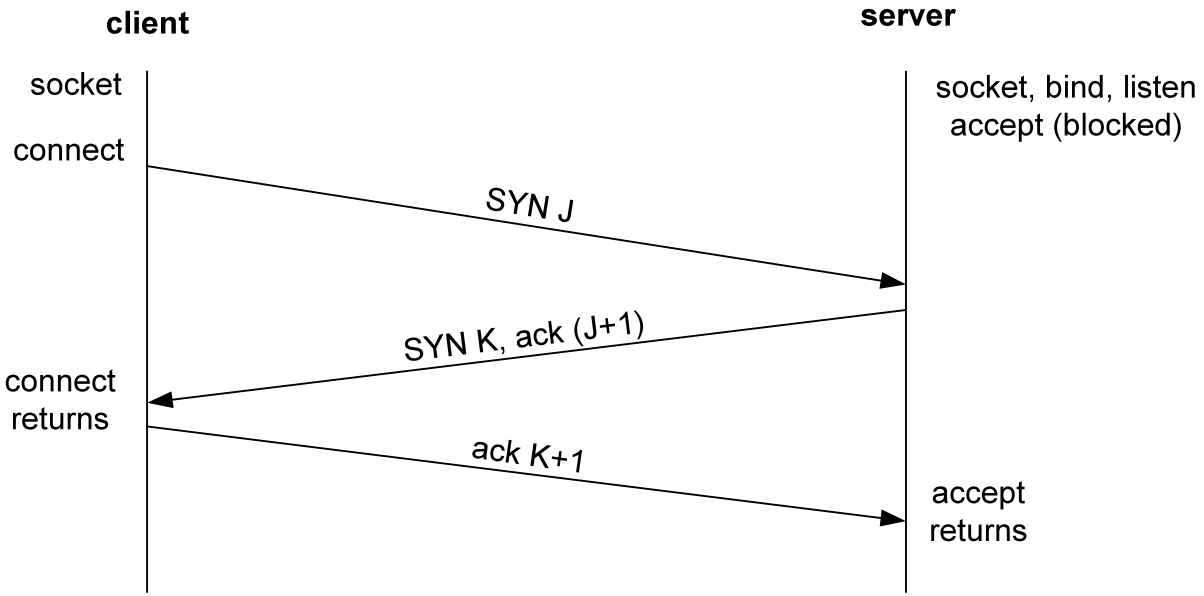


Why Connection Establishment?

- TCP is a "reliable" transport protocol
- Before the protocol can be realized, connection establishment phase is needed for
  - Allowing each side to know the other exists
  - Negotiation of optional parameters
    - Max segment size
    - Initial Sequence Numbers (ISN)
  - Triggering allocation of transport entity resources
    - Buffer
    - Timers (if any)
- The above are done by mutual agreement

TCP Connection Establishment

- Server gets ready (`socket, bind, listen`)
- Client gets ready (`socket`)
- Client requests connection (`connect`)



Timeout of Connection Establishment

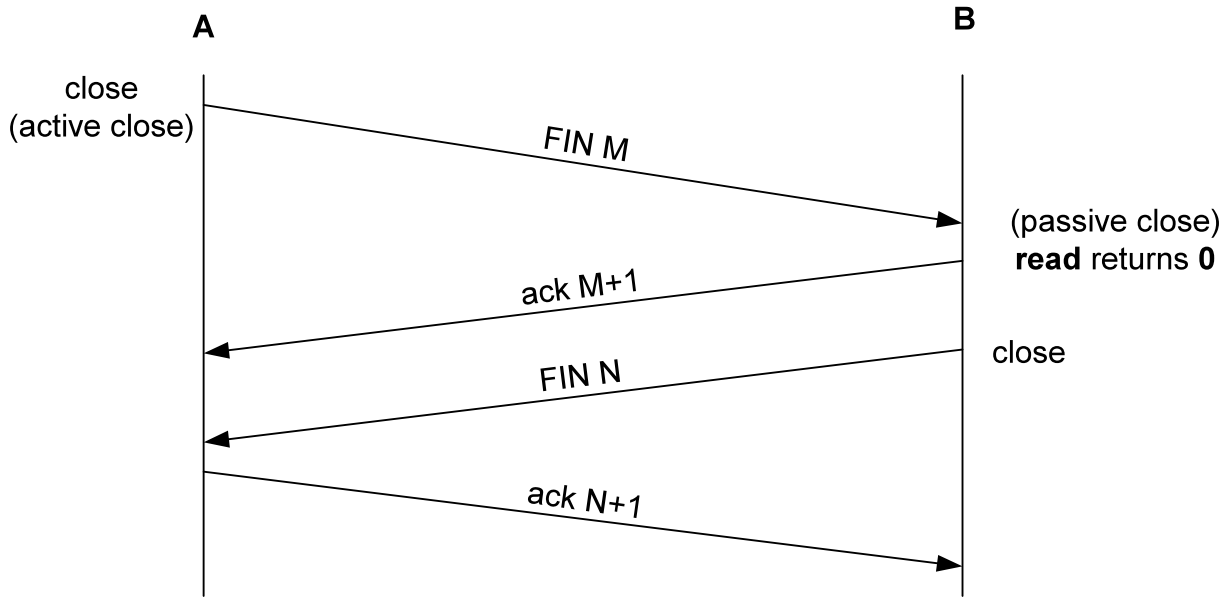
- Retransmissions of SYNs (typically on Unix):
  - 6 seconds after the first SYN
  - 24 seconds after the second SYN
  - 48 seconds after the third SYN
  - give up
  - Most Berkeley derived OSs have an upper limit of 75 sec
- In reality, there is one timer that goes off every 500ms
  - All timeouts are based on this timer (just a count of the number of ticks)

Tips and Tricks

- What's a quick way to find out how long your machine's TCP module would try to establish a connection?

TCP Connection Termination

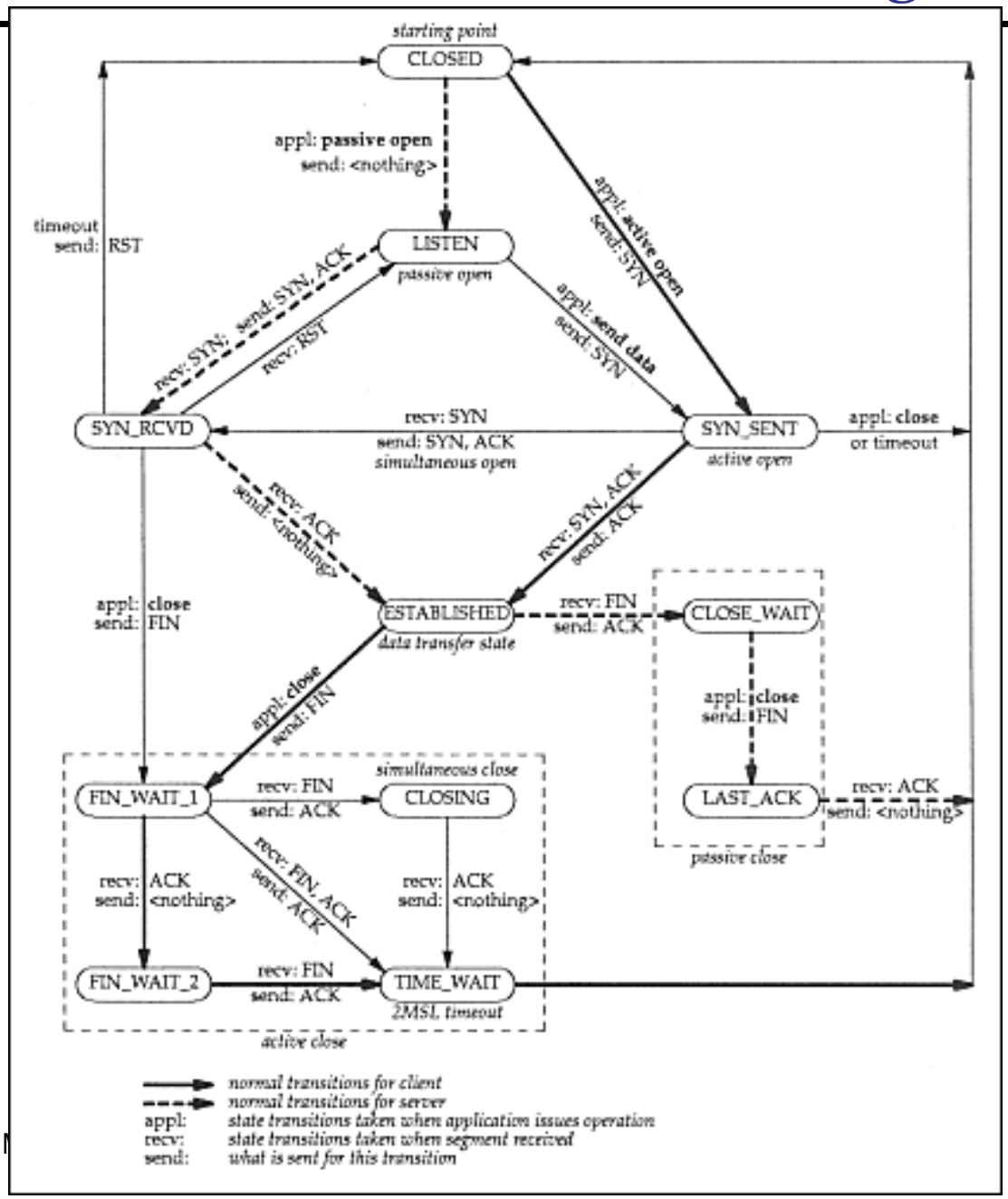
- **A** performs active close, sends FIN
- **B** performs passive close & acknowledges
- **B** closes its socket and sends FIN
- **A** acknowledges the FIN



Ports, Sockets, Socket Pairs

- Ports
  - Well-known ports: 0 – 1023 (assigned by IANA)
  - Registered ports: 1024 – 49151 (not controlled but recommended by IANA for some services)
  - Dynamic (ephemeral) ports: 49152 – 65535
- Socket (prot, local_IP, local_port, remote_IP, remote_port)
  - prot (i.e. protocol) is TCP for TCP sockets
  - For each TCP/UDP connection, the quadruple (local_IP, local_port, remote_IP, remote_port) must be unique!
  - The quadruple is also called a *socket pair*

TCP State Transition Diagram



The Time_Wait (2MSL) State

- MSL: *Maximum Segment Lifetime* (common implementations are either 30sec, 1min, 2min)
- Let TCP resend the final ACK if needed (when?)
- The socket can only be reused after 2MSL (why?)
  - Sometime you can't bind a server port because of this 2MSL state
  - However, setting socket option SO_REUSEADDR allows us to reuse the port number (violation of RFC)
  - But still, no two identical socket pairs
- "Quiet time" (RFC 793): no connection creation within 2MSL after crashing (what for?)

Half-Open Connections

- One end quits w/o the knowledge of the other
- For example, you turn off your PC while a telnet is still on, or your OS freezes and you have to reboot

- How to solve this problem?
  - Timing out alone does not work (look at a web browser)
  - TCP has the *keep alive option*

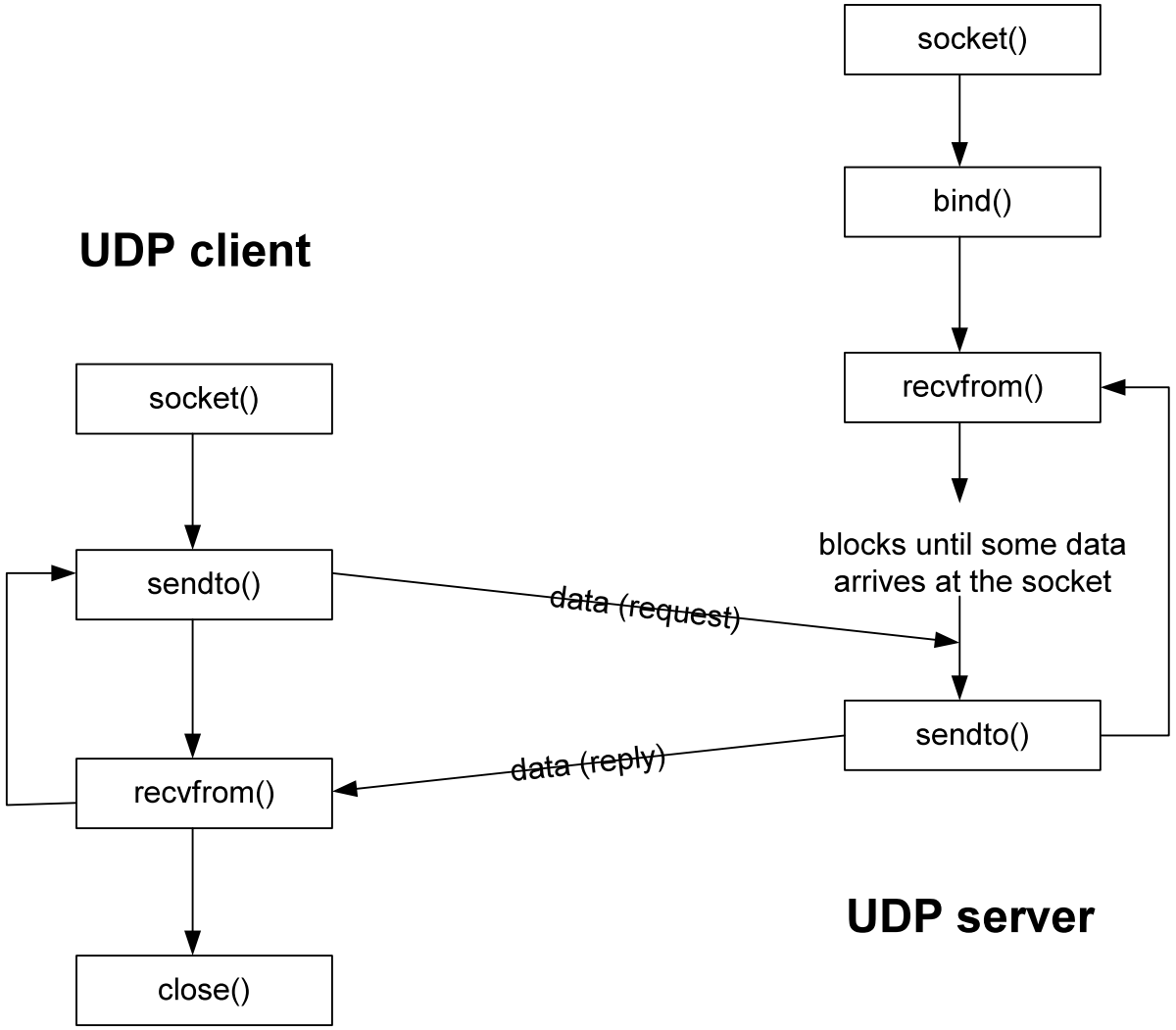
RESET Segment

- RST is used to
  - Reply to connection requests to some port no-one is listening on
  - Reply to connection requests within 2MSL after crashing
- In UDP, an *ICMP port unreachable* is generated instead

UDP Overview

- Client gets ready (`socket`)
- Server gets ready (`socket, bind`)
- Data transfer
  - client `sendto` - server `recvfrom`
  - server `sendto` – client `recvfrom`
- Client closes its socket (`close`)
- Server keeps waiting for other data

Typical UDP Client-Server



Choices, choices

- `read()/write()`
  - Everything in Unix is a “file”
  - Used on TCP/UDP *connected* sockets
  - Doesnot work in Windows!
- `send()/recv()`
  - Used on TCP/UDP connected sockets
- `sendto()/recvfrom()`
  - Used on UDP sockets
  - Good if you need peer’s information (IP/port)

"Connected" UDP Socket

- Why?
  - Get (`ICMP`) error report `ECONNREFUSED` on the next system call to socket
  - Can `write()/send()` without specifying the destination address/port, can `read()/recv()` too
  - Can `sendto()` with a `null` destination address
  - Packets from other IP/port are ignored
  - Can call `connect()` multiple times to different IP/port
- Why not?
  - Packets from other IP/port are ignored

TCP vs. UDP, When to Use What?

- You should be able to partly answer this question on your own

Common Applications & Protocol Usage

Applications	IP	ICMP	UDP	TCP
Ping		x		
Traceroute		x	x	
OSPF	x			
RIP			x	
BGP				x
BOOTP, DHCP, NTP, TFTP, SNMP			x	
SMTP, Telnet, FTP, HTTP, NNTP				x
DNS, NFS, Sun RPC, DCE RPC			x	x

The Sample Codes

1. UDP echo client server
2. TCP echo client server, two types of clients
   - Plain client
   - IO-multiplexing client (using select)
   - Multi-processing server (using fork)
3. Echo server with IO-multiplexing which can handle both TCP & UDP
4. Echo server with multi-threading which can handle both TCP & UDP

- *Ordered in increasing level of sophistication*

SC 1: UDP Echo Client

- Specify server address (IP, port)
- Create socket [socket(AF_INET, SOCK_DGRAM, 0)]
- Read from socket with recvfrom()
- Write to socket with sendto()

- inet_pton() inet_ntop()

SC 1: UDP Echo Server

- Create socket
- Bind to port
  - Tell kernel: give me things sent to this port
  - INADDR_ANY or a specific IP
- Read and write with recvfrom() and sendto()

SC 1: Problems

- From client: which (IP, port) kernel has assigned?
  - getsockname(…)
- From server: which (IP, port) the request is from?
  - getpeername(…)
- Start client first, type something and press "Return", then start the server:
  - the client has no reaction on your typing anymore, even Ctrl^D does not work.
  - You're to explain and fix this problem yourselves in assignment 1

SC 1: Problems

- The UDP socket in this example is not "connected," thus asynchronous errors are not reported
  - Suppose server is down (ICMP "port unreachable")
  - sendto() does not return the error (asynchronous error)
- Solution: connect() the UDP socket. By products:
  - Can no longer specify destination IP, port
  - Should use read() and write() instead of sendto() and recvfrom() on this socket
  - Asynchronous errors are returned to the socket
- connect() can be called multiple times on a UDP socket
  - To specify a new (IP, port)
  - To disconnect

About Coding

"Computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute."

Harold Belson and Gerald Jay Sussman

[*From the "Structure and Interpretation of Computer Programs"*]

SC 2: TCP Echo Client

Plain Client:

- Create "active" socket
  [socket(AF_INET, SOCK_STREAM, 0)]
- Specify server address (IP, port)
- Connect to server [connect()]
- Read from and write to the socket

SC 2: TCP Echo Server

Server:

- Create "passive" socket [socket(), bind(), listen()]
- Accept incoming connection request [accept()]
- New child handles new request [fork()]
- Parent closes new socket
- Child closes listening socket

SC 2: Big Endians and Little Endians

- There are two ways to store an integer (2 byte, 4 byte, etc) in memory
  - Big endian: least significant bit (LSB) at highest address
  - Little endian: LSB at lowest address
  - There are pros and cons in terms of computational efficiency
- Little-endian
  - Intel 80x86 families & Pentiums
  - AMD Duron, Athlon, Thunderbird
  - Apple 6502, DEC PDP & VAX
- Big-endian
  - Sun's SPARC, Motorola 68K, PowerPC, Cray
  - JVM & most IBM processors

SC 2: The Endians & Network Programming

- TCP/IP use big-endian for all its integers (IP addresses, port numbers, etc)
  - We **have to** convert them to big-endian (*network byte order*) before use
- Serious run-time performance penalties occur when using TCP/IP on a little endian processor.

- *Thus, it may be unwise to select a little endian processor for use in a device, such as a router or gateway, with an abundance of network functionality.*

SC 2: Where do these terms come from?

From the book "*Gulliver's Travels*" (1726) by Jonathan Swift.

Gulliver, a traveler, finds himself stranded on the island of Lilliput (a land of tiny men and women). The Lilliputians inform him of a civil war that happened on their island many years before he arrived. The debate was between two groups of people ... those who chose to break their eggs on the larger end ("the big endians" or the Blefuscudians) and those who broke their eggs on the smaller end ("the little endians" - Lilliputians). The little endians won in Lilliput and the big endians were exiled to another island (Blefuscud).

In 1981, Danny Cohen coined the term in an article in IEEE Computer (vol. 14, no. 10)

SC 2: Byte Order Conversion Functions

```
#include <netinet/in.h>

uint16_t   htons(uin16_t   host16bitvalue);
uint32_t   htonl(uin32_t   host32bitvalue);

uint16_t   ntohs(uin16_t   net16bitvalue);
uint32_t   ntohl(uin32_t   net16bitvalue);
```

SC 2: Handling `SIGCHLD`

- Do not want any zombies, thus wait for all children who have finished
- `wait()` and `waitpid()`
- `accept()` might be interrupted (w/o error), need to handle this gracefully (check for `EINTR` and continues accepting)

SC 2: SIGPIPE & Other Errors

- Normally `read()` returns 0 if the other side already closed the connection
- Sometimes `read()` returns an error with `errno=ECONNRESET`
  - The other end already closed (`FIN` was sent but not read)
  - Some time later `read()` is called, we get `RST` instead
  - `SIGPIPE` is delivered in this case
- To check this: let client sleep for a few seconds before read the echoed message
- Some common errors: `ETIMEDOUT, EHOSTUNREACH, ENETUNREACH`

SC 2: Some C Programming Notes

- (My) Error handling routines use variable-length arguments
- Network utility routines sometimes a static function
  - Try the `rio_*` routines
- Separation of header files from source files
- `Makefile` contains most things you need to compile it under Linux, Solaris, Mac OS X
  - A little change and it'll work with Windows too (install cygwin)

SC 2: Some Network Programming Notes

- `readn(), writen(), readline()` are self-explanatory in terms of functionalities
- `rio_*` are also self-explanatory in terms of functionalities; please look into the codes

- We need them to read/write/read a line gracefully (even in cases when there's some signal interrupting the system calls)
- Use them as I provided, do not re-invent the wheel

SC 2: Plain Client Problems

- If server (child process) is killed, client doesn't know since it's hung on `fgets()`
  [try `ls –la` and `kill -9` to see]
  - I/O multiplexing can fix this
- Suppose parent is killed, child keeps serving
  - Restart server and `bind()` fails [why?]
  - `SO_REUSEADDR` socket option will fix this

Tips & Tricks

- truss

I/O Models under Unix

- Blocking I/O
  - read(), write(), sendto(), recvfrom(), connect(), accept(), …
- Nonblocking I/O (use fcntl() and select())
- I/O Multiplexing (select() and poll())
- Signal Driven I/O (SIGIO)
- Asynchronous I/O (Posix.1 aio_ functions)
  - This is relatively new, not all systems support it. Process tells kernel to perform an operation and then let it know when the operation is completed.
  - aio_read(), aio_write(), …

SC 2: TCP echo client with select

- Client select() on the socket and stdin
- This basically solves the problem of client being hung on the call to fgets()
- More sophisticated use of select in the next sample code

SC 2: I/O Multiplexing with select()

```
#include <sys/select.h>
#inlucde <sys/time.h>
int   select(int maxfdp1, fd_set *read_set, fd_set *write_set,
             fd_set *exceptset, const struct timeval *timeout)
```

- If timeout is NULL, wait until one fd is ready; otherwise return when time is up
- If timeout indicates 0, returns immediately if nothing is ready (this is called polling)
- Must remember
  - *maxfdp1 = max fd plus one*!
  - FD_SET descriptors **inside** the loop

Tips and Tricks

- How do you write a program in C or Java or Scheme (or any other language of choice) that prints itself out?
- What's a speculative implication of this?

Tips and Tricks

- Bill Joy's quote:

"*If I had to rewrite Unix from scratch, I could do it in a summer, easily, And it would be much better. A much, much better job. The ideas are old*."

SC 3: Generic Server

- Reuse address (allow parent to die and restart while child still running):

  setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on))
- Take care of EINTR for select() too

- Nothing else is really new

For the Programming Projects

- More useful functions, in case you don't know
  - `gethostname()`
  - `gethostbyname()` [not reentrant]
  - `gethostbyaddr()` [not reentrant]
  - Reentrancy problem because both functions return a pointer to a static structure
  - `getaddrinfo()`
  - `freeaddrinfo()`
  - `getnameinfo()`

Client Server Design Alternatives

1. Iterative server (no forking, no IO Mux)
2. Concurrent server, one new child per request
3. Concurrent server, one new thread per request
4. Concurrent server with I/O multiplexing
   select() and poll()
5. Concurrent server with pre-forking
6. Concurrent server with pre-threading
7. Some combination of 4, 5 and 6
8. …

http://www.kegel.com/c10k.html

Some initial findings

- If server is not heavily loaded, one fork per client is fine
- Creating a pool of children or a pool of threads is a good idea
- Threading creates some technicalities, but it often runs faster – still, it depends on the operating environment
- Having all children or threads call accept() is often faster than having the parent doing it (and then passing it to the child
- Need experimentation to know what to do with your system!

Advanced features

- Socket options
- IPv4 and IPv6 interoperability
- Name & address conversions
- Raw Socket
- Data-link access
- Daemons
- Out-of-band data
- Debugging techniques and tools
- Broadcasting and Multicasting

AF 1: Socket Options

getsockopt(), setsockopt(), fnctl(), ioctl()

There are many, a few more important ones are

- SO_KEEPALIVE, TCP_KEEPALIVE
- SO_LINGER (affects close() – send buffered data or not?)
- SO_REUSEADDR
- TCP_NODELAY (disables Nagle algorithm)

AF 2: IPv4 and IPv6 interoperability

- Most Linux distributions now run *dual stacks*
  - In x years, v4-stack will be turned off
    (x is a solution to the *three-body problem* ☺)
- Assuming dual-stack, in most cases servers and clients can talk to each other, except when
  - A IPv6 client connects to an IPv4 server using AAAA record
- Changes mostly occur with name/address conversions and specification
  - Create IPv6 socket is easy (struct sockaddr_in6, …)
  - gethostbyname() and gethostbyaddr() are protocol dependent
  - Use getaddrinfo() instead (see my solution to project 1)

Tips & Tricks

- What does bcheck() do?

AF 3: Name and Address Conversions

- See my solution to project 1
- getaddrinfo(), getnameinfo() are reentrant functions and IPv4/v6 compatible

AF 4: Raw sockets

- Explore on your own

AF 5: Datalink access

- Explore on your own

AF 6: Writing Daemons

- Explore on your own

AF 7: Out-of-band Data

- Explore on your own

AF 8: Debugging Techniques and Tools

- Explore on your own

AF 9: Broadcasting & Multicasting

- Explore on your own