Socket Programming

See:
- socket/ example code
- U Illinois notes
- UWO notes
- MSDN Winsock2

16 Mar 2009
CMPT166
Dr. Sean Ho
Trinity Western University

BSD sockets

- Sockets are a protocol-independent way of communicating between processes
  - Foundation of the Internet, including HTTP, FTP, IM, streaming media, etc.
- Local or Internet: same host or diff hosts?
- Connection-based or connectionless: does each packet need to specify destination?
- Packets or streams: message boundaries?
- Reliable or unreliable: Can messages be lost, duplicated, reordered, or corrupted?

TCP vs. UDP

- All data on the Internet is sent via packets conforming to the Internet Protocol (IP)
- Two most common types of packets:
  - TCP: Transmission Control Protocol:
    - Virtual circuit: connection-based
    - Client-server model
  - UDP: User Datagram Protocol:
    - Connectionless: peer-to-peer, less overhead
    - No guarantees about arrival, ordering, duplication of packets
- We can create both kinds of sockets

TCP client-server

- TCP is connection-based:
  - Phone analogy
  - Initial setup, but subsequent packets do not need to specify destination again
  - Server: waits, listens for client
  - Client: initiates connection (phone call)
  - Once connection is established, communication may be two-way (send/receive)
  - Either client or server may terminate

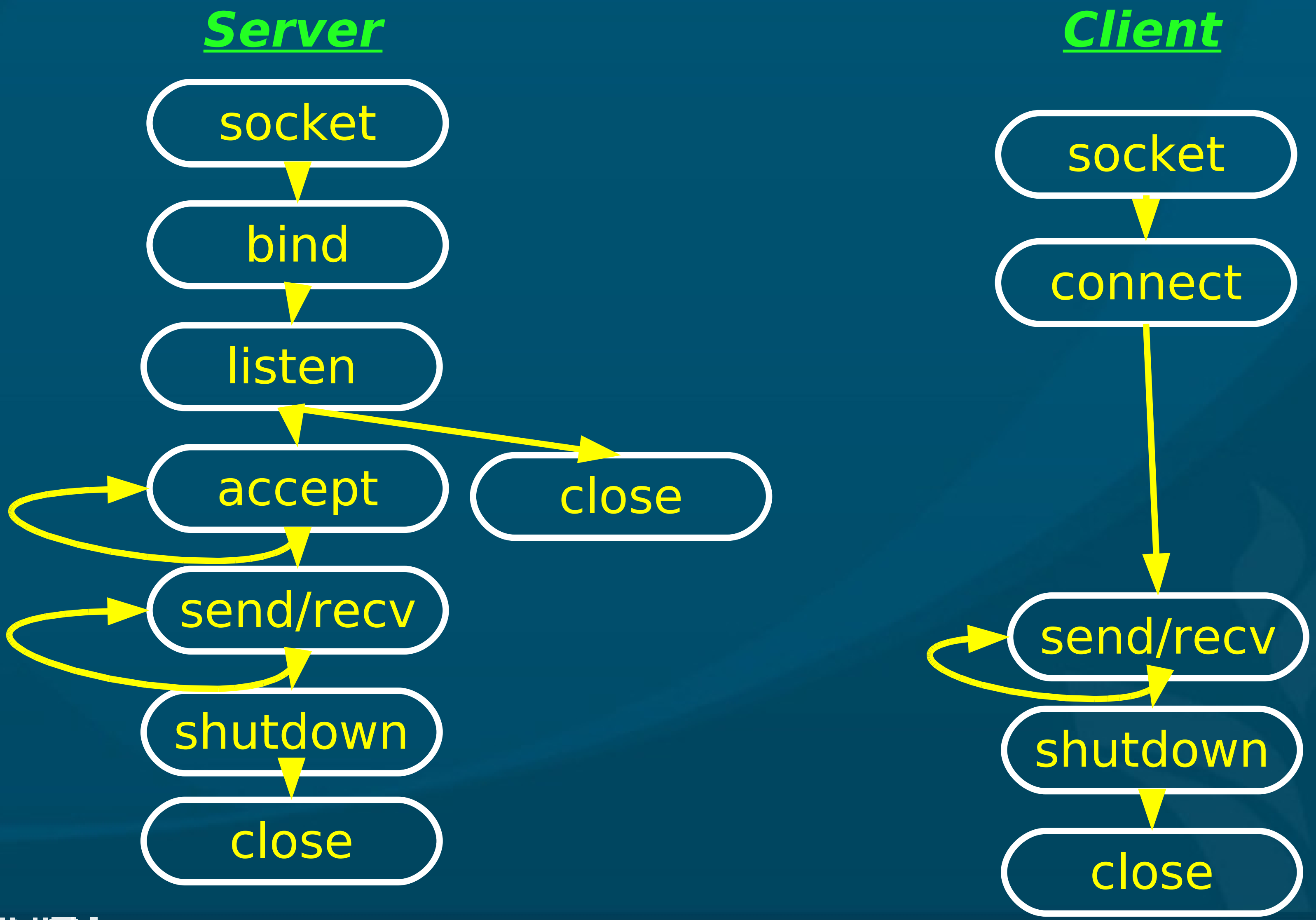
Steps for TCP server

- socket(): create socket *(buy a phone)*
- bind(): specify server port *(get a phone number)*
- listen(): specify length of connection queue *(call-waiting)* and enable socket for listening
- accept(): wait for client and establish connection *(wait for and answer phone)*
- send()/recv() (repeated): communicate (via buffers of bytes/chars)
- shutdown(): mute or end call
- close(): release data structures

Steps for TCP client

- socket(): create socket *(buy a phone)*
- connect(): connect to a server *(dial phone number)*
- send()/recv() (repeated): communicate (via buffers of bytes/chars)
- shutdown(): mute or end call
- close(): release data structures

TCP client-server diagram



Sockets API: socket()

- Create a new socket:
  - #include <sys/types.h>
  - #include <sys/socket.h>
  - int socket(AF_INET, *type*, 0);
- Domain: AF_INET for internet or AF_LOCAL
- Type: SOCK_STREAM (connection), SOCK_DGRAM (connectionless datagram), SOCK_SEQPACKET (sequenced, reliable connection; not usually available)
- Protocol: 0 chooses TCP/UDP according to type
- Returns a socket ID (akin to a file handle)

Sockets API: bind()

- Associates a socket with an address
  - int bind(*sid, addrPtr, len*);
- sid: socket ID (from return value of socket())
- addrPtr: pointer to the address struct
  - Type: struct sockaddr*
  - Structure depends on the address family
  - For IP, need: IP address and port
    - Type: struct sockaddr_in*
- len: size (in bytes) of *addrPtr

Address structs: sockaddr

- The address struct (addrPtr parameter) can be:
- If the address family is IP:
  - struct sockaddr_in {
    - sa_family_t sin_family;   // AF_INET
    - in_port_t sin_port;   // port number
    - struct in_addr sin_addr;   // IP address struct
  - }
- If the address family is a local Unix socket:
  - struct sockaddr_un {
    - uint8_t sun_length;
    - short sun_family;   // AF_LOCAL
    - char sun_path[100];   // path/filename

Sockets API: listen()

- int listen(*sid*, *size*);

- Set number of pending connection requests allowed (any incoming requests beyond this will get rejected)
- SID: socket ID (from socket())
- size: max length of connection queue
- Typically limited by OS to only 5!
- Returns 0 on success, -1 on failure

Sockets API: accept()

- Blocks (waits) until a client initiates a connection request
    - int accept(*sid, addrPtr, lenPtr*)
- When received, creates a new connection ID (handle) for this client
- Return value is the connection ID
- *addrPtr is the address info of the client
    - struct sockaddr* addrPtr, int* lenPtr
- addrPtr or lenPtr are 0 if no client or no address info

How do we accept clients?

- Iterating server: only one client at a time
  - One operator answering phones
  - Simplest to implement
- Forking server:
  - Split off a child thread for each connection
  - Original master thread continues to listen
  - Switchboard
- Concurrent single server:
  - Use select to simultaneously wait on all open socket IDs

More on forking server

- Multiple threads running concurrently
- Master thread listens on port
- When a client connects, fork off a thread
  - Thread handles communication with that client
- Master thread continues listening for other connections (switchboard)
- Overhead in forking new threads: so keep pool of available threads, and reuse dormant threads