Socket-based Communication

CMPUT-379
Lab

Outline

- Inter-process Communication using Sockets
- Unix and I-Net Sockets
- Server side
- Client side
- Server initialization
- Client initialization
- Sending and receiving
- Byte orders
- Closing the connection
- Examples

IPC using Sockets

- Communication between two processes:
    - Pipes, signals, etc.
    - Sockets
- Sockets provide a communication channel.
- They are essentially file descriptors:
    - We can read/write using regular file commands.
- Processes on the same machine: UNIX Sockets.
- Communication over network: INET Sockets.
- We have a client and a server.

Socket APIs

- `socket`: creates a socket of a given domain, type, protocol (buy a phone)
- `bind`: assigns a name to the socket (get a telephone number)
- `listen`: specifies the number of pending connections that can be queued for a server socket. (call waiting allowance)
- `accept`: server accepts a connection request from a client (answer phone)
- `connect`: client requests a connection request to a server (call)
- `send, sendto`: write to connection (speak)
- `recv, recvfrom`: read from connection (listen)
- `shutdown`: end the call

Server side: abstract view

- Server listens for incoming connections.
- Accepts (potentially many) connections.
- What do we need?
    - An "identifier" distinguishing us from other listeners
        - UNIX Communication: a "name".
        - INET Communication: an IP address and port #.
    - A socket on which we accept incoming connections.
    - A way of binding this socket to that "identifier".
    - A way of bringing the server up and listening.
    - A way of accepting new connections.
    - A file descriptor (socket) for the communication.

Connection-based communication

Server performs the following actions

- `socket`: create the socket
- `bind`: give the address of the socket on the server
- `listen`: specifies the maximum number of connection requests that can be pending for this process
- `accept`: establish the connection with a specific client
- `send,recv`: stream-based equivalents of read and write (repeated)
- `shutdown`: end reading or writing
- `close`: release kernel data structures

Client Side: abstract view

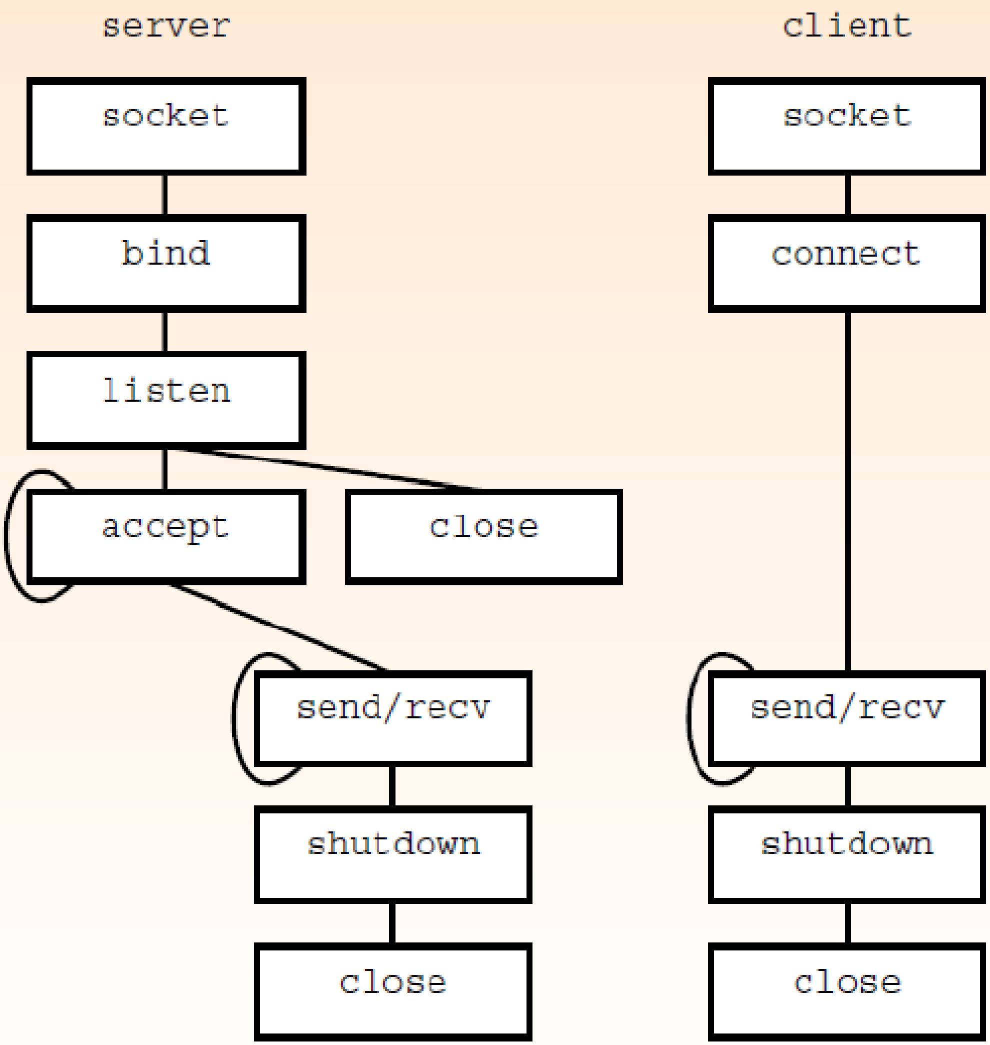
- Client initiates the connection to the server.
- What do we need?
  - The server "identifier".
  - A socket to use for the connection.
  - A way of connecting to the server using this socket and that "identifier".

TCP client

Client performs the following actions

- `socket`: create the socket
- `connect`: connect to a server
- `send,recv`: (repeated)
- `shutdown`
- `close`

TCP-based sockets



Server Initialization

- Create a socket (click on function names):
  - s = socket()
- Set server info (identifier):
  - struct sockaddr_un
  - struct sockaddr_in
- Bind the socket to the identifier:
  - bind()
- Start to listen for connections:
  - listen()
- Accept incoming connections:
  - s = accept()

Client Initialization

- Create the socket (click on function names):
  - `s = socket()`
- Set address info (server identifier):
  - `struct sockaddr_un`
  - `struct sockaddr_in`
- Start the connection:
  - `connect()`

Sending and receiving

- You can use sockets as regular file descriptors.
- To read:
  - `read()`
- To write
  - `write()`
- There are also two other functions for sockets:
  - `send()`
  - `recv()`
- The difference is that the latter provide socket-specific options and messages.

Byte orders

- Different machines may have different byte orders
  - Little-Endian vs. Big-Endian.
- Transmitting a number as it is stored in our own machine may lead to incorrect ordering on the other machine. The same for receiving.
- What to do?
  - Have a default "Network Byte Order".
  - Each machine knows his own byte order & network's.
  - Convert all values to and from "Network Byte Order" before sending and after receiving:
    - `htons(), ntohs(), htonl(), ntohl()`

Closing the connection

- When you are done, you should close the connection:
  - `close()`
    - This closes the file descriptor completely.
    - Sending/receiving from the other end generates error.
  - `shutdown()`
    - This closes parts of a socket functionality.
    - Does not free the file descriptor; you still use `close()`

Port usage

Note that the initiator of communications needs a fixed port to target communications.
This means that some ports must be reserved for these “well known” ports.
Port usage:

- 0-1023: These ports can only be binded to by root
- 1024-5000: well known ports
- 5001-64K-1: ephemeral ports

Examples

- Check the four example clients and servers.
- Pay attention to the way the system calls are used.
- Try running them and see the results.
- For in-depth explanation of what you have seen so far, make sure to check the APUE book.

References

- B. Hall, "Beej's Guide to Network Programming", http://beej.us/guide/bgnet/, Accessed: Oct 18, 2011