Introduction to Unix Sockets

- A socket is a type of descriptor that defines a bi-directional endpoint of communication
- Sockets are used as a basic building block for interprocess communication
- Associated with a socket is a data structure that includes a send buffer and a receive buffer
- A socket is created using the \texttt{socket()} system call, which takes as parameters:
  - protocol family
  - type of communication
  - specific protocol (often implicit)
- A socket is created without a name (address); a name is later \textit{bound} to the socket

Why do we need sockets?

- The ultimate receiver is a \textit{process}. Why not just use the transport layer directly?
- Ports or sockets are a set of abstract destination points and are usually identified by a positive integer. Some ports are reserved for specific services and are “published.” (see /etc/services)
  - packets arriving for a particular port are usually queued until a process extracts them
  - processes waiting at a port are typically blocked until packets arrive
- To communicate with a port, the sender must know the internet address of the destination machine as well as the port id.

Socket Naming

- Every socket must be \textit{bound} with a name (or address) before it can be referenced
- Different protocols associated with a socket may have different naming structures
- The name space defined by a protocol family is called a \textit{domain}
- The most commonly used families are:
  - Unix Domain (AF_UNIX)
    - UNIX system internal protocols
    - interprocess communication within same host and file system
    - socket name is a file path name
  - Internet Domain (AF_INET)
    - interprocess communication among different hosts
    - socket name includes internet address and port
    - some port numbers are reserved for system use (see /etc/services)
    - the name of a socket may be obtained using the system call \texttt{getsockname()}

Socket Types

- The socket has the indicated \textit{type}, which specifies the semantics of communication. Currently defined types are:
- datagram socket (SOCK_DGRAM)
  - unreliable datagram communication
  - record boundary (fixed maximum) is preserved
  - implemented using the UDP protocol
- stream socket (SOCK_STREAM)
  - reliable virtual circuit communication
  - “record” boundary is not preserved
  - implemented using the TCP protocol
- sequenced packet socket (SOCK_SEQPACKET)
  - reliable virtual circuit communication
  - record boundary is preserved
- raw socket (SOCK_RAW)
  - direct interface to the IP protocol
  - used in new protocol development
- reliably delivered message socket (SOCK_RDM)
  - acknowledged datagram communication
Using Sockets with UDP

- Specify use of datagrams upon socket creation.
- If only one-way communication, the sending process need not bind a name to its socket.
- The receiving process must bind a name to its socket. The sending process references the name of the receiver's socket.

- Procedures:
  - socket creation: `socket(domain, type, protocol)`
  - name (address) binding: `bind(socket, name)`
  - send or receive through the socket: (several primitives available; see man pages)
  - close the socket: `close(socket)`

- Sending and receiving datagrams
  - `sendto()`: used for unconnected sockets
  - `read()`: primitive routine for sockets and other entities
  - `recvfrom()`: also retrieves name of sending socket

---

Unix Stream Sockets

- Also created with `socket()` system call
- Socket type: SOCK_STREAM
  - reliable virtual circuit communication
  - "record" boundary is not preserved
  - When used in AF_INET domain, implemented using the TCP protocol
- `socketpair()` - create a pair of connected sockets
  - generalization of pipes
  - only supports the AF_UNIX domain
  - same machine

- Timeouts and broken connections: If data is not transmitted within a reasonable length of time, then the connection is broken and subsequent calls will fail with ETIMEDOUT.

- Use `bind()` system call for both datagram and stream sockets
  - socket descriptor
  - name (family, host addr, port)
  - length of name
  - often let the OS choose the port

Stream Socket – Passive Side

- Await (listen for) connections
- Use `listen()` system call to prepare operating system for connection requests

- Parameters:
  - socket descriptor
  - backlog: defines the maximum length the queue of pending connections (usually 5)

---

Passive Side (cont.)

- Use the `accept()` system call
- Parameters
  - socket descriptor
  - name (family, host addr, port)
  - length of name

- Extracts the first connection on the queue of pending connections, creates a new socket, and allocates a new file descriptor for the socket. Another system call like this?
  - Normally blocks, but this can be turned off
  - Returns descriptor of new socket.
  - Why does `accept()` work this way?
Stream Socket - Active Side

- Initiate a connection on a socket
- Use the `connect()` system call

**Parameters:**
- socket descriptor
- name (family, host addr, port)
- length of name

- For stream sockets, attempts to make a connection to the socket named in the call.

- For datagram sockets?

**Errors**
- ETIMEDOUT
- ECONNREFUSED
- ENETDOWN or EHOSTDOWN
- ENETUNREACH or EHOSTREACH

Transferring Data

- Transmitting
  - `write()` as for files
  - `send()` uses flags to specify such requests as out-of-band transmission (MSG_OOB)

- Receiving
  - `read()` as for files
  - `recv()` uses flags to specify such requests as examining data without reading it (MSG_PEEK)

Using Sockets

- Create a socket.

- Binding of names to a socket.

Using Sockets

- Establishing Socket Connections.
  - Server side.

- Client side.

- Closing a connection.
Using Sockets

- Transferring Data.
  - read and write.

- send and recv.