Lecture 18: Network I/O via Sockets

Mythili Vutukuru
IIT Bombay
Sockets

- Computers in a computer system exchange messages over the network
  - OS provides system calls to support this communication
- Sockets = abstraction to communicate between two processes
  - Each process opens socket, and pair of sockets can be connected
  - **Client-server paradigm**: one process opens socket first (server) and another process connects its socket to the first one (client)
  - One process writes a message into one socket, another process can read it, and vice versa (bidirectional communication)
  - Processes can be in same machine or on different machines

- In this lecture: system calls to send/receive messages via sockets
- Next lecture: how socket system calls are implemented
Types of sockets (1)

• **Unix domain (local) sockets** are used to communicate between processes on the same machine
  • Server process opens socket, and gives it a name (pathname)
  • Client process opens socket, connects to the server socket using its name

• **Internet sockets** are used to communicate between processes in different machines
  • Every machine on the Internet has an address = **IP address**
  • Multiple sockets on the machine get unique **port numbers** (16 bits)
  • Server process opens socket at a well known port number
  • Client process opens socket, connects to the server socket using its IP address and port number

• Client and server sockets differentiated by who starts first and who connects later: server sockets listens for communication on a well-known “address”, client process connects to server using the server address
  • Mechanisms exist for clients to learn server addresses
Types of sockets (2)

- **Connection-based sockets**: one client socket and one server socket are explicitly connected to each other.
  - After connection, the two sockets can only send and receive messages to each other.
- **Connection-less sockets**: one socket can send/receive messages to/from multiple other sockets.
  - Address of other endpoint can be mentioned on each message.
- Type of socket (local or internet, connection-oriented or connection-less) is specified as arguments to system call that creates sockets.
- Connection-based Internet sockets are called **TCP sockets**.
  - TCP is a protocol to guarantee in-order reliable delivery of messages across Internet.
- Connection-less Internet sockets are called **UDP sockets**.
  - UDP is a protocol to exchange messages on Internet without any reliability.
- More on TCP and UDP protocols later in the course.
Creating a socket

- System call “socket” used to create a socket
  - Takes type of socket as arguments
  - Returns socket file descriptor
- Socket file descriptor is similar to file descriptor returned when file is opened
  - Index of entry in file descriptor array, points to socket-based data structures
  - Used as handle for all future operations on the socket
- A socket can optionally bind to an address (pathname or IP address/port number) using “bind” system call
  - Server sockets must always bind to well known address, so that clients can connect
  - Client sockets need not bind, OS can assign temporary address
- Close system call closes a socket when done
Data exchange using connection-less sockets

- Function `sendto` is used to send a message from one socket to another connection-less socket in another process
  - Arguments: socket fd, message to send, address of remote socket
- Function `recvfrom` is used to receive a message from a socket
  - Arguments: socket fd, message buffer into which received message is copied, socket address structure into which address of remote endpoint is filled
- When a process receives a message on connection-less socket, it can find out the address of other endpoint which sent message, and use this address to reply back

Client:
```c
sockfd = socket(..)
char message[1024]
sendto(sockfd, message, server_sockaddr, ..)
```

Server:
```c
sockfd = socket(..)
bind(sockfd, server_address)
recvfrom(sockfd, message, client_sockaddr, ..)
```
Connecting sockets

- Connection-oriented sockets must be explicitly connected to each other before exchanging messages.
- After server binds socket to well-known address, it uses “listen” system call to make the socket listen for new connections.
- Client uses “connect” system call to connect to a server listen socket:
  - Connect system call blocks until messages exchanged with server to complete connection procedure (more later).
- Server uses “accept” system call to accept new connection requests:
  - Accept system call blocks until new connection is received.
  - Returns a new socket file descriptor to communicate exclusively with a connected client.
- At server: one listen socket to accept new connections, one connected socket for every connected client to send/recv messages.

Client:
sockfd = socket(..)
connect(sockfd, server_sockaddr, ..)

Server:
sockfd = socket(..)
bind(sockfd, server_address)
listen(sockfd, ..)
accept(sockfd, ..)
newsockfd = accept(sockfd, ..)
Data exchange using connected sockets

- After client connects to server, pair of sockets used to exchange data
  - Note that per-client connected socket is used at server, not listen socket
  - System calls `send/write` used to send message on a connected socket
  - System calls `recv/read` used to receive message on a connected socket

- Arguments to `send/recv`: socket fd, message buffer, buffer length, flags
  - Return value is number of bytes read/written or error
  - No need to specify socket address on every message, as connected already
  - Send/recv has extra flags argument, as compared to read/write system calls
  - Flags control where system call blocks and other behavior

```c
sockfd = socket(..)
bind(sockfd, server_sockaddr, ..)
listen(sockfd, ..)
newsockfd = accept(sockfd, ..)
n = recv(newsockfd, req_buf, req_len, ..)
n = send(newsockfd, resp_buf, resp_len, ..)
```
Concurrent network I/O

- What if server is connected to multiple clients?
  - Multiple sockets to manage: listen socket, per-client connected sockets
  - Accept on listen socket, read/receive on connected socket can block until data arrives
  - If server process blocks on one socket, it can neglect other sockets

- How to concurrently handle multiple clients?
  - Server process can create multiple child processes/threads, one per connected client
  - Main server process blocks at accept on listen socket
  - Child processes/threads can block at reading from connected client sockets
  - New client connections as well as existing client communication handled
  - Processes/threads multiplexed on the same core, or run in parallel on different cores

- Cannot support very large number of clients due to limit on number of processes or threads a system can support

```c
sockfd = socket(..)
bind(sockfd, server_address)
listen(sockfd, ..)
newsockfd = accept(sockfd, ..)
n = recv(newsockfd, req_buf, req_len)
n = send(newsockfd, resp_buf, resp_len)
```
Event-driven I/O

- Event-driven or asynchronous I/O API used to concurrently handle I/O from multiple sockets in a single process
  - Example: `select`, `epoll`

- Overview of `epoll` API
  - Process creates an `epoll` instance, adds file descriptors of interest to be monitored
  - Process blocks on `epoll_wait`, which returns when there is an event on any of the socket file descriptors (OS takes care of monitoring all fds)
  - When `epoll_wait` returns, process handles events by performing suitable actions (accept, `recv` etc.) on the ready file descriptors, and calls `epoll_wait` when done
  - Single-threaded process enough to handle I/O from multiple concurrent clients
  - Process should not do any blocking action when handling event

- Event-driven APIs available for network I/O, not popular for disk
Network I/O architecture

• Most components in a computer system need to do some network I/O, sometimes as clients and sometimes as servers
  • Web server receives requests from users, contacts database, returns response
• Programming language libraries may provide better APIs for network I/O than the basic socket API discussed here
  • Example: remote procedure call (RPC) APIs invoke server code like function calls
• With any API, design choice to be made between two architectures
  • One thread per connection, blocking/synchronous API
  • Fewer threads, event-driven/asynchronous API
• Event-driven APIs usually have lesser overhead and higher performance, but harder to program, difficult to scale to multiple cores
Summary

• In this lecture:
  • Socket API to communicate between processes
  • Multi-process/multi-thread vs event-driven architecture

• Programming exercise: write code for a client and server that exchange data using sockets. Add functionality for concurrent handling of multiple clients.