

# Issues in Client/Server Programming

Refs: Chapter 27

# Issues in Client Programming

- Identifying the Server.
- Looking up a IP address.
- Looking up a well known port name.
- Specifying a local IP address.
- UDP client design.
- TCP client design.

# Identifying the Server

- Options:
  - hard-coded into the client program.
  - require that the user identify the server.
  - read from a configuration file.
  - use a separate protocol/network service to lookup the identity of the server.

# Identifying a TCP/IP server.

- Need an IP address, protocol and port.
  - We often use *host names* instead of IP addresses.
  - usually the protocol (UDP vs. TCP) is not specified by the user.
  - often the port is not specified by the user.



Can you name one common exception ?

# Services and Ports

- Many services are available via “well known” addresses (names).
- There is a mapping of service names to port numbers:

```
struct *servent getservbyname( char *service,  
                               char *protocol );
```

- `servent->s_port` is the port number in network byte order.

# Specifying a Local Address

- When a client creates and binds a socket it must specify a local port and IP address.
- Typically a client doesn't care what port it is on:

```
haddr->port = htons(0);
```



**give me any available port !**

# Local IP address

- A client can also ask the operating system to take care of specifying the local IP address:

```
haddr->sin_addr.s_addr=  
    htonl ( INADDR_ANY );
```



**Give me the appropriate address**

# UDP Client Design

- Establish server address (IP and port).
- Allocate a socket.
- Specify that any valid local port and IP address can be used.
- Communicate with server (send, recv)
- Close the socket.

# Connected mode UDP

- A UDP client can call `connect()` to establish the address of the server.
- The UDP client can then use `read()` and `write()` or `send()` and `recv()`.
- A UDP client using a connected mode socket can only talk to one server (using the connected-mode socket).

# TCP Client Design

- Establish server address (IP and port).
- Allocate a socket.
- Specify that any valid local port and IP address can be used.
- Call `connect()`
- Communicate with server (read,write).
- Close the connection.

# Closing a TCP socket

- Many TCP based application protocols support multiple requests and/or variable length requests over a single TCP connection.
- How does the server know when the client is done (and it is OK to close the socket) ?

# Partial Close

- One solution is for the client to shut down only it's writing end of the socket.
- The shutdown() system call provides this function.

```
shutdown( int s, int direction);
```

- direction can be 0 to close the reading end or 1 to close the writing end.
- shutdown sends info to the other process!

# TCP sockets programming

- Common problem areas:
  - null termination of strings.
  - reads don't correspond to writes.
  - synchronization (including `close()`).
  - ambiguous protocol.

# TCP Reads

- Each call to read() on a TCP socket returns any available data (up to a maximum).
- TCP buffers data at both ends of the connection.
- *You must be prepared to accept data 1 byte at a time from a TCP socket!*

# Server Design

Iterative  
Connectionless

Iterative  
Connection-Oriented

Concurrent  
Connectionless

Concurrent  
Connection-Oriented

# Concurrent vs. Iterative

- An iterative server handles a single client request at one time.
- A concurrent server can handle multiple client requests at one time.

# Concurrent vs. Iterative

## Concurrent

- Large or variable size requests
  - Harder to program
- Typically uses more system resources

## Iterative

- Small, fixed size requests
  - Easy to program

# Connectionless vs. Connection-Oriented

## Connection-Oriented

- EASY TO PROGRAM
- transport protocol handles the tough stuff.
- requires separate socket for each connection.

## Connectionless

- less overhead
- no limitation on number of clients

# Statelessness

- *State*: Information that a server maintains about the status of ongoing client interactions.
- Connectionless servers that keep state information must be designed carefully!

**Messages can be duplicated!**

# The Dangers of Statefulness

- Clients can go down at any time.
- Client hosts can reboot many times.
- The network can lose messages.
- The network can duplicate messages.

# Concurrent Server Design Alternatives

One child per client

Spawn one thread per client

Preforking multiple processes

Prethreaded Server

# One child per client

- Traditional Unix server:
  - TCP: after call to `accept()`, call `fork()`.
  - UDP: after `readfrom()`, call `fork()`.
  - Each process needs only a few sockets.
  - Small requests can be serviced in a small amount of time.
- Parent process needs to clean up after children!!!! (call `wait()`).

# One thread per client

- Almost like using `fork()` - just call `pthread_create` instead.
- Using threads makes it easier (less overhead) to have sibling processes share information.
- Sharing information must be done carefully (use `pthread_mutex`)

# Prefork ( )'d Server

- Creating a new process for each client is expensive.
- We can create a bunch of processes, each of which can take care of a client.
- Each child process is an iterative server.

# Prefork ( )'d TCP Server

- Initial process creates socket and binds to well known address.
- Process now calls `fork()` a bunch of times.
- All children call `accept()`.
- The next incoming connection will be handed to one child.

# Preforking

- As the book shows, having too many preforked children can be bad.
- Using dynamic process allocation instead of a hard-coded number of children can avoid problems.
- The parent process just manages the children, doesn't worry about clients.

# Sockets library vs. system call

- A preforked TCP server won't usually work the way we want if *sockets* is not part of the kernel:
  - calling `accept()` is a library call, not an atomic operation.
- We can get around this by making sure only one child calls `accept()` at a time using some locking scheme.

# Prethreaded Server

- Same benefits as preforking.
- Can also have the main thread do all the calls to `accept()` and hand off each client to an existing thread.

# What's the best server design for my application?

- Many factors:
  - expected number of simultaneous clients.
  - Transaction size (time to compute or lookup the answer)
  - Variability in transaction size.
  - Available system resources (perhaps what resources can be required in order to run the service).

# Server Design

- It is important to understand the issues and options.
- Knowledge of queuing theory can be a big help.
- You might need to test a few alternatives to determine the best design.