Network Computing & Programming

Lecture 4

Key Problems of Client & Server Design

2019-2020-1

Slides are modified from Xiang(Shaun) Zhang

Issues in C/S Programming

① Identifying the Server.

② Looking up an IP address.

③ Looking up a well known port name.

④ Specifying a local IP address.

⑤ UDP/TCP client design.

Identifying the Server

- **Options:**
  1. Hard-coded into the client program.
  2. Require that the user identify the server.
  3. Read from a configuration file.
  4. 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.**
  - Usually use **host names** instead of IP addresses
    - Two parameter: `ostec.uestc.edu.cn http`
    - One parameter: `ostec.uestc.edu.cn:http`

Converting an IP address

```
in_addr_t inet_addr(const char *strptr);
```

- Convert **ASCII dotted-decimal IP address** to **network byte ordered 32-bit (IPv4) value**. Just like `inet_aton()` learned before.
- Returns a 32-bit binary network byte ordered IPv4 address and **INADDR_NONE** on error

```
(...)
  struct sockaddr_in dest;
  memset(&dest, '\0', sizeof(dest));
  dest.sin_addr.s_addr = inet_addr(”202.115.16.62");
(...)
```

Looking up an IP address

struct hostent *gethostbyname(const char *hostname); (but obsolete, use **getaddrinfo()** now)

```
struct hostent{
    Char   *h_name;
    Char   ** h_aliases;
    Short  h_addrtype;
    Short  h_length;
    Char   ** h_addr_list;};
```

```
(...)
struct hostent *remoteHost;
remoteHost = gethostbyname(hostname)
(...)
```

Looking up a well known port name

```
struct servent *getservbyname(const char * name,
  const char *proto);
   struct servent{
     char    *s_name;
     char    **s_aliases;
     int     s_port;
     char    *s_proto;};
(...)
struct servent *remoteSvr;
remoteSvr = getservbyname("domain", "udp")
(...)
```

Looking up a protocal

```
struct protoent *getprotobynumber(int proto)
```

- Look up /etc/protocols to get result.

```
struct protoent {
  char    *p_name;
  char    *p_aliases;
  short   *p_proto;};
```

```
(...)
struct protoent *pptr;
pptr = getprotobyname(“udp”)
(...)
```

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 working 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 <u>establish</u> 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 known 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.
- `Shutdown()` 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

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 Statelessness

- 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 **`recvfrom()`**, 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
  - call pthread_create instead
- Using threads makes it easier to have sibling processes share information
  - less overhead
- Sharing information must be done carefully
  - use pthread_mutex

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

- Having too many preforked children can be bad.

- Using dynamic process allocation instead of a hard-coded number of children can avoid problems.

- 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.