CS 50011: Introduction to Systems II

Prof. Jeff Turkstra

Computer Science Department
Purdue University

Copyright 2017

Sockets and Ports

- A port defines an end-point (application) in the machine itself
- There are well-known ports:
  - HTTP Port 80
  - SSH Port 22
  - FTP Port 21
- If you are building an application that will be deployed globally, you may request your own port number.
- A socket is a file descriptor that can be used to receive incoming connections or to read/write data to a client or server.

Sockets and Ports

- A TCP connection is defined uniquely in the entire Internet by four values:
  <src-ip-addr, src-port, dest-ip-addr, dest-port>
- Example: A runs an HTTP server in port 80
- B connects to A's HTTP server using source port 5000
  - The connection is <IB, 5000, IA, 80>
- C connects also to A's HTTP server using src port 8000
  - The connection is <IC, 8000, IA, 80>
- Another browser in B using port 6000 connects to A
  - The connection is <IB, 6000, IA, 80>
- Another browser in C using port 5000 connects to A
  - The connection is <IC, 5000, IA, 80>
- The OS uses this 4 values to know what data corresponds to what application/socket.

Sockets API

- They were introduced by UNIX BSD (Berkeley Standard Distribution).
- They provide a standard API for TCP/IP.
- A program that uses sockets can be easily ported to other OS's that implement sockets: Example: Windows.
- Sockets were designed general enough to be used for other platforms besides TCP/IP. That also makes sockets more difficult to use.

Sockets API

- Sockets offer:
  - Stream interface for TCP.

    Read/Write is similar to writing to a file or pipe.
  - Message based interface for UDP

    Communication is done using messages.
- The first applications were written using sockets: FTP, mail, finger, DNS etc.
- Sockets are still used for applications where direct control of the network is required.
- Communication is programmed as a conversation between client and server mostly using ASCII Text.

Programming With Sockets

Client Side

```
int cs =socket(PF_INET, SOCK_STREAM, proto)
…
Connect(cs, addr, sizeof(addr))
…
Write(cs, buf, len)
Read(cs, buf, len);
Close(cs)
```

See:

Lab 03 client.cpp

Programming With Sockets

- Server Side

```
...
int masterSocket = socket(PF_INET, SOCK_STREAM, 0);
…
int err = setsockopt(masterSocket, SOL_SOCKET, SO_REUSEADDR, (char *) &optval,
      sizeof( int ) );
…
int error = bind( masterSocket, (struct sockaddr *)&serverIPAddress, sizeof(serverIPAddress) );
…
error = listen( masterSocket, QueueLength);
…
while ( 1 ) {
…
    int slaveSocket = accept( masterSocket,
        (struct sockaddr*)&clientIPAddress, (socklen_t*)&alen);
    read(slaveSocket, buf, len);
    write(slaveSocket, buf, len);
   close(slaveSocket);
}
```

- See: http://www.cs.purdue.edu/homes/cs354/lab5-http-server/lab5-src/daytime-server.cc

Client for Daytime Server

client.c

Daytime Server

daytime-server.c

Types of Server Concurrency

- Iterative Server
- Fork Process After Request
- Create New Thread After Request
- Pool of Threads
- Pool of Processes

Iterative Server

```
void iterativeServer( int masterSocket) {
  while (1) {
    int slaveSocket =accept(masterSocket,
          &sockInfo, &alen);
    if (slaveSocket >= 0) {
  dispatchHTTP(slaveSocket);
    }
  }
}
```

Note: We assume that dispatchHTTP itself closes slaveSocket.

Fork Process After Request

```
void forkServer( int masterSocket) {
   while (1) {
       int slaveSocket = accept(masterSocket,
                                &sockInfo, &alen);
      if (slaveSocket >= 0) {
        int ret = fork();
`       if (ret == 0) {
             dispatchHTTP(slaveSocket);
             exit(0);
        }
        close(slaveSocket);
      }
   }
}
```

Create Thread After Request

```
void createThreadForEachRequest(int masterSocket)
{
  while (1) {
    int slaveSocket = accept(masterSocket, &sockInfo, &alen);
    if (slaveSocket >= 0) {
      // When the thread ends resources are recycled
      pthread_attr_t attr;
      pthread_attr_init(&attr);
      pthread_attr_setdetachstate(&attr,
                    PTHREAD_CREATE_DETACHED);
      pthread_create(&thread, &attr,
             dispatchHTTP, (void *) slaveSocket);
    }
  }
}
```

Pool of Threads

```
void poolOfThreads( int masterSocket ) {
   for (int i=0; i<4; i++) {
    pthread_create(&thread[i], NULL, loopthread,
                   masterSocket);
   }
   loopthread (masterSocket);
}

void *loopthread (int masterSocket) {
  while (1) {
    int slaveSocket = accept(masterSocket,
                      &sockInfo, &alen);
    if (slaveSocket >= 0) {
      dispatchHTTP(slaveSocket);
    }
  }
}
```

Pool of Processes

```
void poolOfProcesses( int masterSocket ) {
   for (int i=0; i<4; i++) {
     int pid = fork();
     if (pid ==0) {
       loopthread (masterSocket);
     }
   }
   loopthread (masterSocket);
}

void *loopthread (int masterSocket) {
  while (1) {
    int slaveSocket = accept(masterSocket,
                      &sockInfo, &alen);
    if (slaveSocket >= 0) {
      dispatchHTTP(slaveSocket);
    }
  }
}
```

Notes:

- In Pool of Threads and Pool of processes, sometimes the OS does not allow multiple threads/processes to call accept() on the same masterSocket.
- In other cases it allows it but with some overhead.
- To get around it, you can add a mutex_lock/mutex_unlock around the accept call.

```
mutex_lock(&mutex);
int slaveSocket = accept(masterSocket,
                         &sockInfo, 0);
mutex_unlock(&mutex);
```

- In the pool of processes, the mutex will have to be created in shared memory.

Questions?