CS110 Spring 2021

Lecture 13: Intro To Networking

Principles of Computer Systems

Stanford University, Dept. Of Computer Science

Lecturers: Jerry Cain & Roz Cyrus

Version 1

Accessing Code Examples

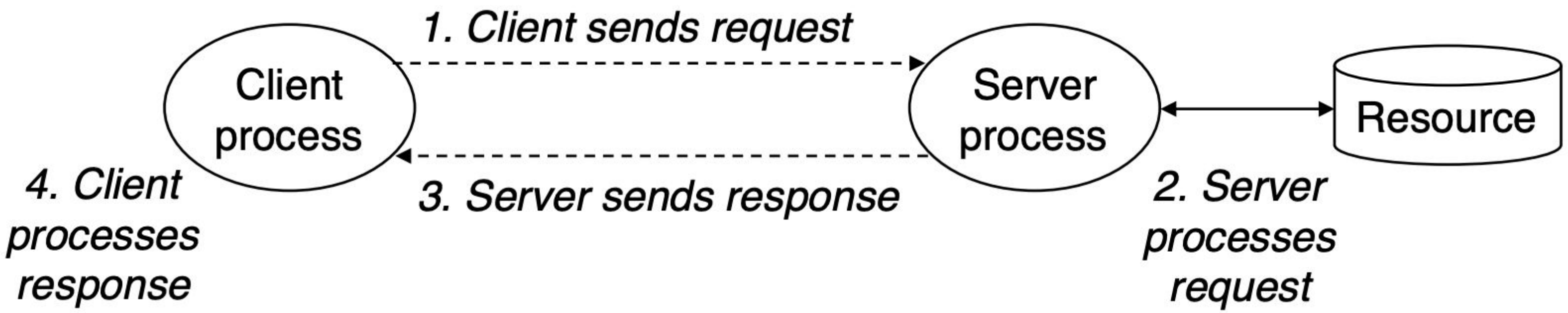
- Today's lecture examples reside within:
  `/usr/class/cs110/lecture-examples/networking`.
  - First **`ssh`** into a myth machine (ssh yourusername@myth.stanford.edu). When prompted for your password, it is normal for the text not to appear as you enter your password. Once logged onto a myth machine, **`cd`** into the above directory.
  - To get started, type:

    ```
    git clone /usr/class/cs110/lecture-examples cs110-lecture-examples
    ```

    at the command prompt to create a local copy of the master.
  - Each time I mention there are new examples (or whenever you think to), descend into your local copy and type **`git pull`**. Doing so will update your local copy to match whatever the master has become.

Networking

Reading: B&O's Network Programming chapter



- Networking is simply communication between two computers connected on a network. You can actually set up a network connection on a single computer as well.
- A network requires one computer to act as the **server**, waiting patiently for an incoming connection from another computer, the **client**.
- Note: clients and servers are processes and not machines (or hosts, as they are often called). A single host can run many different clients and servers concurrently, and a client and server transaction can be on the same or different hosts.
- To a host, a network is just another I/O device that serves as a source and sink for data.

Networking

- Every computer on a network has a unique **IP address** that identifies it on the network.
    - When you want to connect to a server, you need to know its IP address.
    - By convention, each of the four bytes in a 32-bit IP address is represented by its decimal value and separated by a period. An example IP address is "192.168.1.1".
    - "500.304.259.1" is not a valid IP address, since the numbers 500, 304, and 259 are greater than 255 and thus can't be stored in single bytes.

Networking

- Humans aren't generally good at remembering strings of numbers like IP addresses.
  - The **Domain Name System (DNS)** translates human-friendly hostnames (also called **domain names**) into IP addresses.
  - **`nslookup`** is a program to query Internet domain name servers. You can make a DNS query to ask for the IP address of "web.stanford.edu," and you'll get back 171.67.215.200:

```
$ nslookup stanford.edu
[omitted]
Non-authoritative answer:
Name:    stanford.edu
Address: 171.67.215.200
```

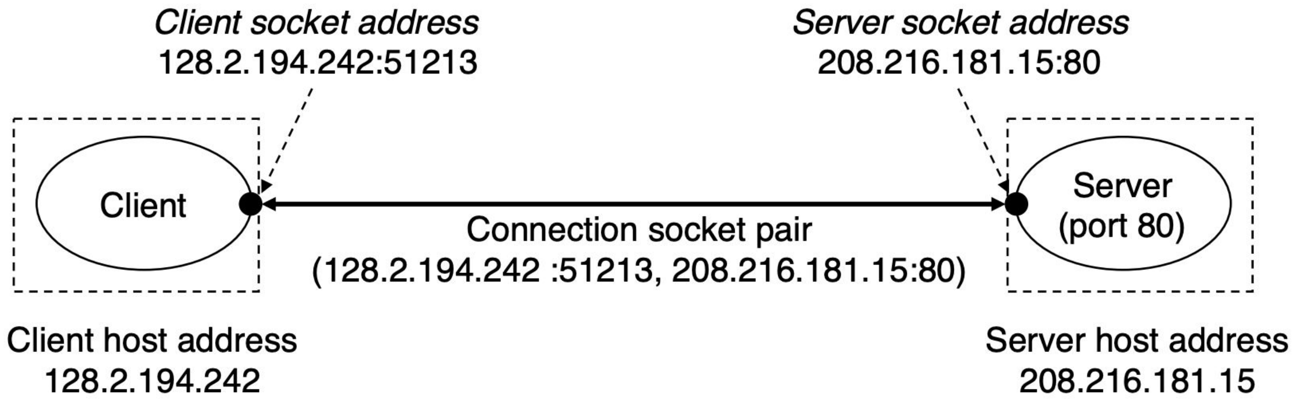
Networking

Reading: B&O's Network Programming chapter

- Internet **clients** and **servers** communicate by sending and receiving streams of bytes over **connections**.
  - A connection is **point-to-point** in the sense that it connects a pair of processes.
  - It is **full duplex** in the sense that data can flow in both directions at the same time.
  - It is **reliable** in the sense that the stream of bytes sent by the source process is eventually received by the destination process in the same order it was sent (barring some catastrophic failure).

Networking

Reading: B&O's Network Programming chapter



Server-side applications set up a **socket** that listens on a particular **port**. From the perspective of the Linux kernel, **a socket is an endpoint of a connection**. But to a Linux program, a socket is an open file with a corresponding descriptor.

- A port number is like a virtual process ID that the host associates with the true pid of the application's process.
- Each socket has a **socket address** that consists of an internet address and a 16-bit integer port. The socket address is denoted by the notation **address:port**.
- The port in the client's socket address is assigned automatically by the kernel when the client makes a connection request and this port is known as an **ephemeral port**.
- By contrast, the port in the server's socket address is usually some well-known port that is permanently associated with the service (example: **http**, the well-known service name for the Web service, is port 80; **https** is 443).
- A connection is uniquely identified by a **socket pair**: the socket addresses of its two end points.
    - The socket pair is denoted by the tuple **(client address:client port, server address:server port)**.

Networking

- You can see some of the ports your computer is listening to with the **`netstat`** command:

```
myth59$ netstat -plnt
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:587           0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      -
tcp6       0      0 :::22                   :::*                    LISTEN      -
tcp6       0      0 ::1:631                 :::*                    LISTEN      -
```

- Some common ports are listed above. You can see a full list here and here.
  - Ports 25 and 587 are SMTP (Simple Mail Transfer Protocol), for sending and receiving email.
  - Port 53 is the DNS (Domain Name Service) port, for associating names with IP addresses.
  - Port 22 is the port for SSH (Secure Shell)
  - Port 631 is for IPP (internet printing protocol)
- For your own programs, generally try to stay away from port numbers listed in the links above, but otherwise, ports are up for grabs to any program that wants one.

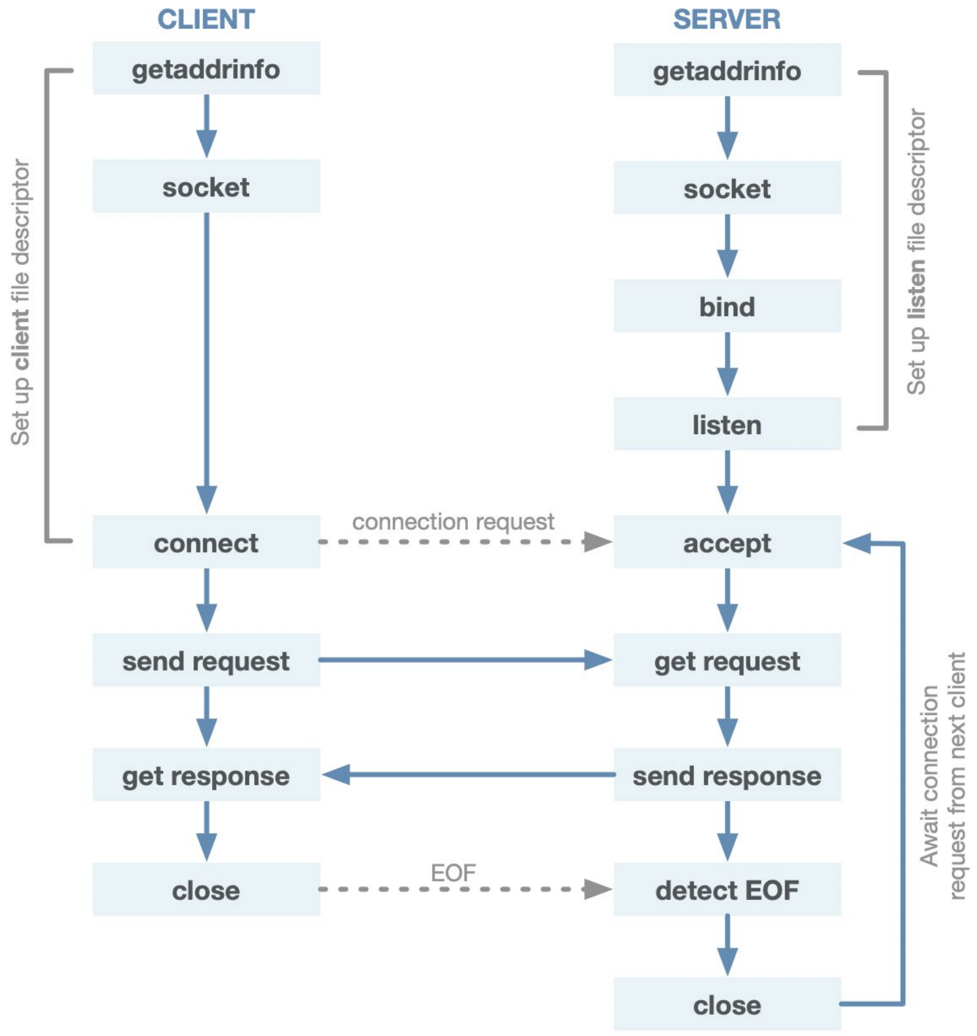
Networking

- If you're curious, mappings between well-known ports and service names are contained in the file `/etc/services` on each Linux machine.

```
myth59$ cat /etc/services
...
tcpmux       1/tcp                      # TCP port service multiplexer
echo         7/tcp
echo         7/udp
...
ftp          21/tcp
fsp          21/udp       fspd
ssh          22/tcp                     # SSH Remote Login Protocol
telnet       23/tcp
smtp         25/tcp       mail
time         37/tcp       timserver
time         37/udp       timserver
whois        43/tcp       nicname
tacacs       49/tcp                     # Login Host Protocol (TACACS)
tacacs       49/udp
domain       53/tcp                     # Domain Name Server
domain       53/udp
bootps       67/udp
...
http         80/tcp       www           # WorldWideWeb HTTP
...
```

Networking

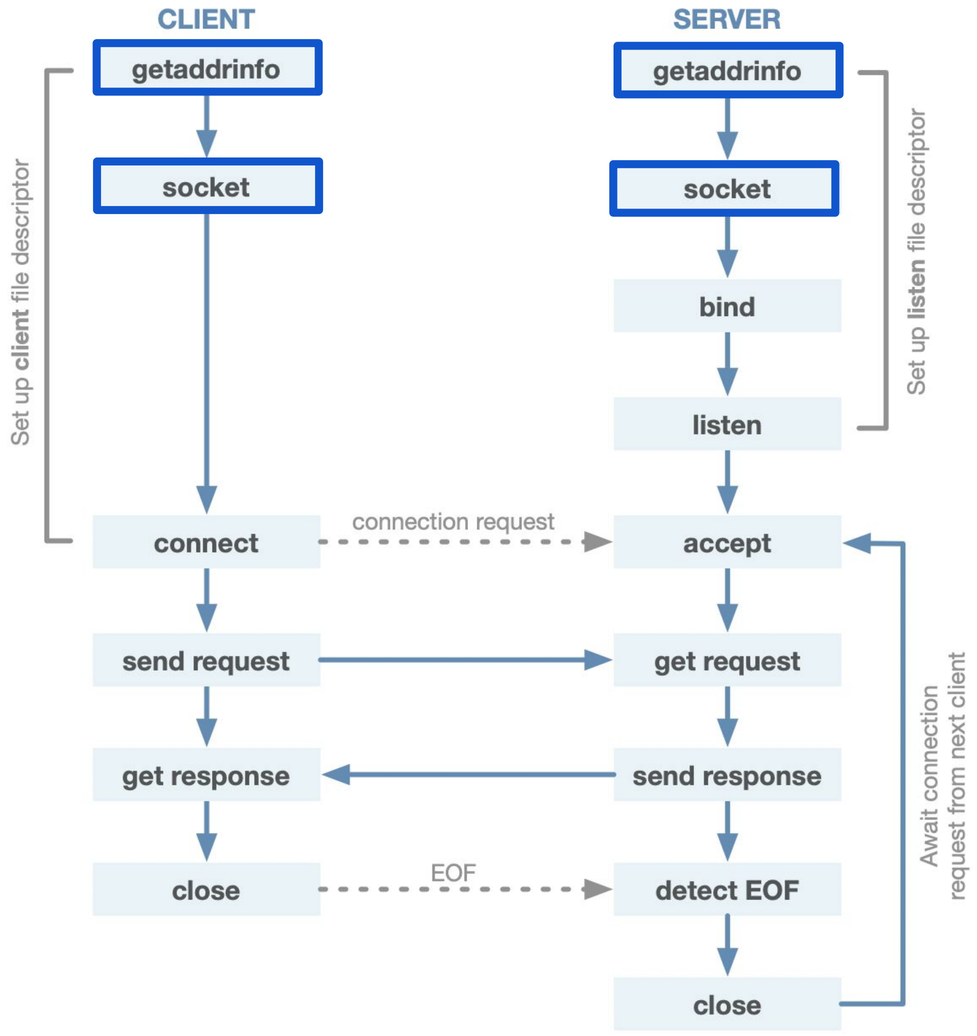
The sockets interface



Networking

The sockets interface

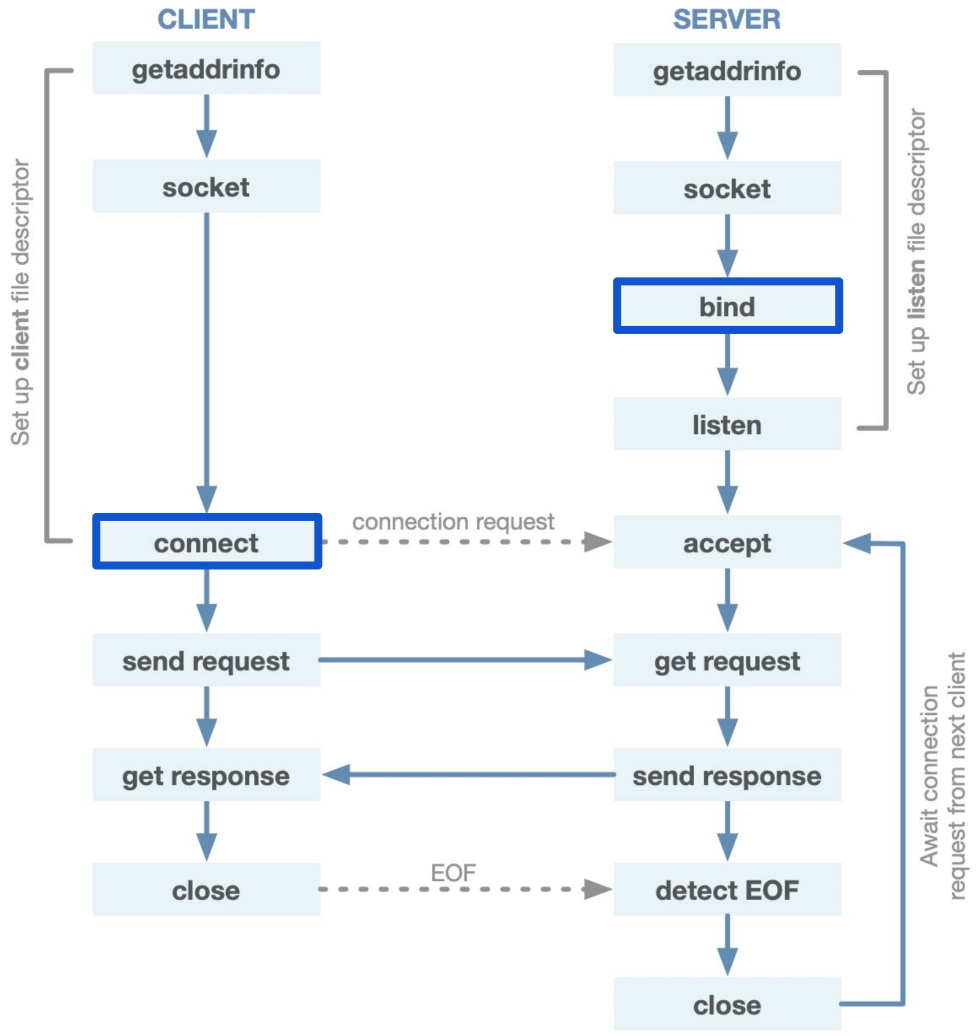
- Clients and servers use the **`socket`** function to create a **socket descriptor**. We can set up the socket to be the endpoint of a connection by calling **`socket`** with specific arguments or by using **`getaddrinfo`** to generate these parameters automatically.



Networking

The sockets interface

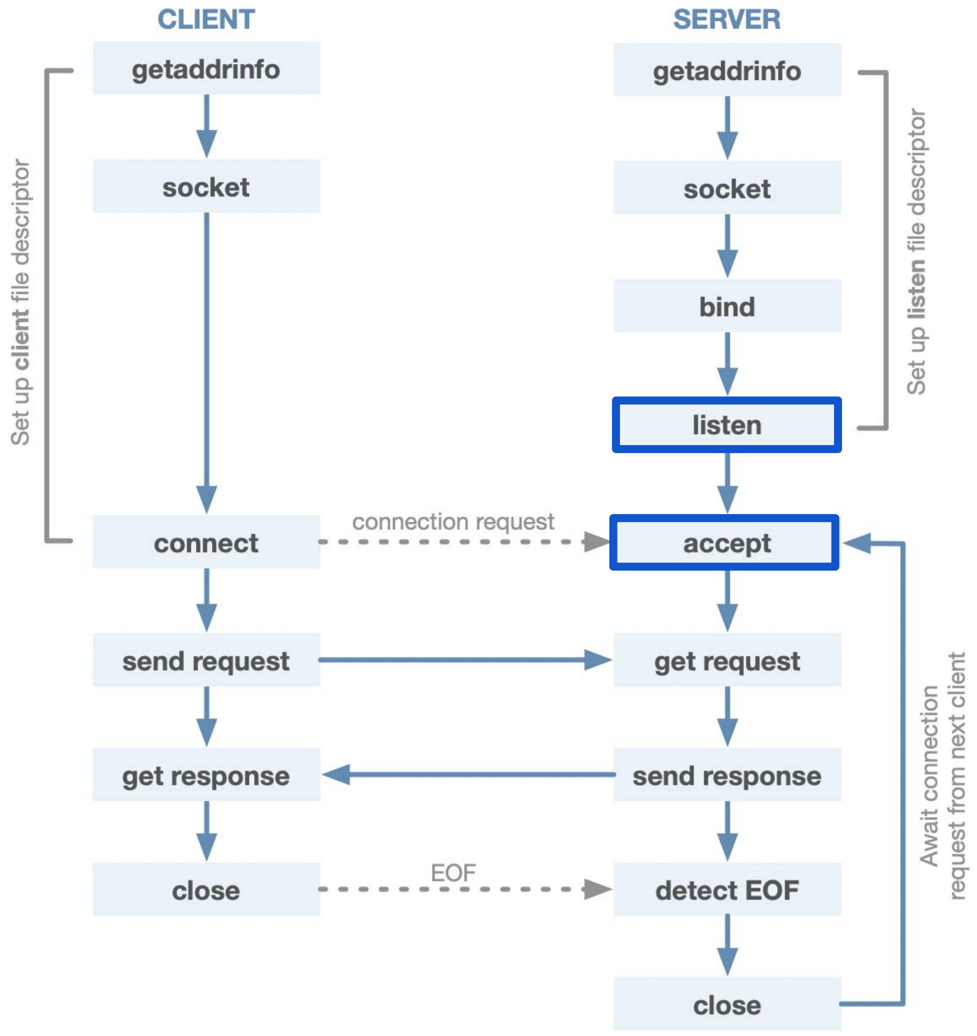
- The **bind** function asks the kernel to associate a server's socket address with a socket descriptor.
- A client establishes a connection with a server by calling the **connect** function.



Networking

The sockets interface

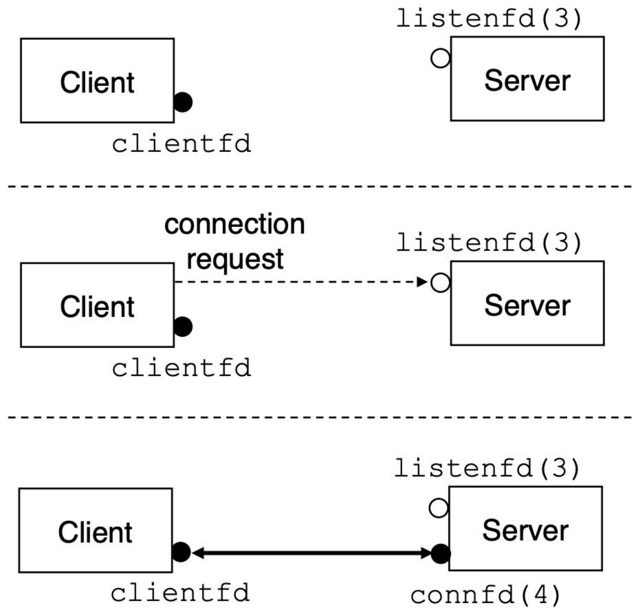
- The **`listen`** function tells the kernel that the descriptor will be used by a server instead of a client; this is important because clients are active entities that initiate connection requests, whereas servers are passive and wait for connection requests from clients. By default, the kernel assumes that a descriptor created by the socket function is active.
- Servers wait for connection requests from clients by calling the **`accept`** function. The function returns a **connected descriptor** that can be used to communicate with the client using Unix I/O functions.



Networking

Difference between a listening descriptor and a connected descriptor:

- The **listening descriptor** serves as an endpoint for client connection *requests*. It is typically created once and exists for the lifetime of the server.
- The **connected descriptor** is the endpoint of the connection that is established between the client and the server. It is created each time the server *accepts* a connection request and exists only as long as it takes the server to service a client.



1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`.

2. Client makes connection request by calling and blocking in `connect`.

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`.

Networking

- Let's create our first server (entire program here):

```
int main(int argc, char *argv[]) {
  int server = createServerSocket(12345);
  while (true) {
    int client = accept(server, NULL, NULL); // the two NULLs could instead be used to
                                             // surface the IP address of the client
    publishTime(client);
  }
  return 0;
}
```

- **`accept`** (found in **`sys/socket.h`**) returns a descriptor that can be written to and read from. Whatever's written is sent to the client, and whatever the client sends back is readable here.
  - This descriptor is one end of a bidirectional pipe bridging two processes—even if they are on different machines!

Networking

- The **`publishTime`** function is straightforward:

```c
static void publishTime(int client) {
  time_t rawtime;
  time(&rawtime);
  struct tm *ptm = gmtime(&rawtime);
  char timestr[128]; // more than big enough
  /* size_t len = */ strftime(timestr, sizeof(timestr), "%c\n", ptm);
  size_t numBytesWritten = 0, numBytesToWrite = strlen(timestr);
  while (numBytesWritten < numBytesToWrite) {
    numBytesWritten += write(client,
                             timestr + numBytesWritten,
                             numBytesToWrite - numBytesWritten);
  }
  close(client);
}
```

- The first five lines here produce the full time string that should be published.
  - Let these five lines represent more generally the server-side computation needed for the service to produce output. Here, the payload is the current time, but it could have been a static HTML page, a Google search result, an RSS document, or a movie on Netflix.
- The remaining lines publish the time string to the client socket using the low-level I/O we've seen before.

Networking

- Note that the **`while`** loop for writing bytes is a bit more important now that we are networking: we are more likely to need to write multiple times on a network.
  - The socket descriptor is bound to a network driver that may have a limited amount of space.
  - That means **`write`**'s return value could very well be less than what was supplied by the third argument.
- Ideally, we'd rely on either C streams (e.g. the **`FILE *`**) or C++ streams (e.g. the **`iostream`** class hierarchy) to layer over data buffers and manage the **`while`** loop around exposed **`write`** calls for us.
- Fortunately, there's a stable, easy-to-use third-party library—one called **`socket++`** that provides exactly this.
  - **`socket++`** provides iostream subclasses that respond to **`operator<<`**, **`operator>>`**, **`getline`**, **`endl`**, and so forth, just like **`cin`**, **`cout`**, and file streams do.
  - We are going to operate as if this third-party library was just part of standard C++.
- The next slide shows a prettier version of **`publishTime`**.

Networking

- Here's the new implementation of **`publishTime`** (and the updated code):

```
static void publishTime(int client) {
  time_t rawtime;
  time(&rawtime);
  struct tm *ptm = gmtime(&rawtime);
  char timestr[128]; // more than big enough
  /* size_t len = */ strftime(timestr, sizeof(timestr), "%c", ptm);
  sockbuf sb(client);
  iosockstream ss(&sb);
  ss << timestr << endl;
} // sockbuf destructor closes client
```

- We rely on the same C library functions to generate the time string.
- This time, however, we insert that string into an **`iosockstream`** that itself layers over the client socket.
- Note that the intermediary **`sockbuf`** class takes ownership of the socket and closes it when its destructor is called.

Networking

- Multithreading can significantly improve the performance of networked applications.
- Our time server can benefit from multithreading as well. The work a server needs to do in order to meet the client's request might be time consuming—so time consuming, in fact, that the server is slow to iterate and accept new client connections.
- As soon as **`accept`** returns a socket descriptor, spawn a child thread—or reuse an existing one within a **`ThreadPool`**—to get any intense, time consuming computation off of the main thread. The child thread can use a second processor or a second core, and the main thread can quickly move on to its next **`accept`** call.
- Here's a new version of our time server, which uses a **`ThreadPool`** (which you're implementing for Assignment 5) to get the computation off the main thread.

```
int main(int argc, char *argv[]) {
    int server = createServerSocket(12345);
    ThreadPool pool(4);
    while (true) {
        int client = accept(server, NULL, NULL); // the two NULLs could instead be used
                                                 // to surface the IP address of the client
        pool.schedule([client] { publishTime(client); });
    }
    return 0;
}
```

Networking

- The implementation of **`publishTime`** needs to change just a little if it's to be thread safe.
- The change is simple but important: we need to call a different version of **`gmtime`**.
- **`gmtime`** returns a pointer to a single, statically allocated global that's used by all calls.
- If two threads make competing calls to it, then both threads race to pull time information from the shared, statically allocated record.
- Of course, one solution would be to use a **`mutex`** to ensure that a thread can call **`gmtime`** without competition and subsequently extract the data from the global into local copy.
- Another solution—one that doesn't require locking and one I think is better—makes use of a second version of the same function called **`gmtime_r`**. This second, reentrant version just requires that space for a dedicated return value be passed in.
- A function is **reentrant** if a call to it can be interrupted in the middle of its execution and called a second time before the first call has completed.
- While not all reentrant functions are thread-safe, **`gmtime_r`** itself is, since it doesn't depend on any shared resources.
- The thread-safe version of **`publishTime`** is presented on the next slide.

Networking

- Here's the updated version of **publishTime** (and the updated code):

```
static void publishTime(int client) {
    time_t rawtime;
    time(&rawtime);
    struct tm tm;
    gmtime_r(&rawtime, &tm);
    char timestr[128]; // more than big enough
    /* size_t len = */ strftime(timestr, sizeof(timestr), "%c", &tm);
    sockbuf sb(client); // destructor closes socket
    iosockstream ss(&sb);
    ss << timestr << endl;
}
```

Networking

- Implementing your first client! (code here)
- The **protocol**—that's the set of rules both client and server must follow if they're to speak with one another—is very simple.
    - The client connects to a specific server and port number. The server responds to the connection by publishing the current time into its own end of the connection and then hanging up. The client ingests the single line of text and then itself hangs up.

```
int main(int argc, char *argv[]) {
    int clientSocket = createClientSocket("myth61.stanford.edu", 12345);
    assert(client >= 0);
    sockbuf sb(clientSocket); // destructor closes socket
    iosockstream ss(&sb);
    string timeline;
    getline(ss, timeline);
    cout << timeline << endl;
    return 0;
}
```

- We'll soon discuss the implementation of `createClientSocket`. For now, view it as a built-in that sets up a bidirectional pipe between a client and a server running on the specified host (e.g. `myth61`) and bound to the specified port number (e.g. 12345).

Networking

- Here’s some output when running our server and client (in this example, on the same machine):

```
rcyrus@myth61$ ./time-server-concurrent &
[1] 15016
rcyrus@myth61$ Server listening on port 12345.
rcyrus@myth61$ ./time-client myth61.stanford.edu 12345
Fri May 14 20:49:51 2021
rcyrus@myth61$ ./time-client myth61.stanford.edu 12345
Fri May 14 20:50:42 2021
rcyrus@myth61$ ./time-client myth61.stanford.edu 12345
Fri May 14 20:50:43 2021
rcyrus@myth61$ ./time-client myth61.stanford.edu 12345
Fri May 14 20:50:44 2021
rcyrus@myth61$
```

Question: will this work if we use two different myth machines?

Networking

- Yes! See the following example:

```
On myth61:
rcyrus@myth61$ ./time-server-concurrent
rcyrus@myth61$ Server listening on port 12345.

On another myth:
rcyrus@myth66$ ./time-client myth61.stanford.edu 12345
Fri May 14 20:49:51 2021
rcyrus@myth66$ ./time-client myth61.stanford.edu 12345
Fri May 14 20:50:42 2021
rcyrus@myth66$ ./time-client myth61.stanford.edu 12345
Fri May 14 20:50:43 2021
rcyrus@myth66$ ./time-client myth61.stanford.edu 12345
Fri May 14 20:50:44 2021
rcyrus@myth66$
```

End of Lecture 13

SO THE ROUTER SAYS TO THE DOCTOR
IT HURTS WHEN IP.
Copyright 1997 Jeffrey Jeffords
quickmeme.com