

CS 111: Operating System Principles  
Lecture 16

# Distributed Systems

3.0.0

Jon Eyolfson

November 23, 2021



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/)

## Sockets are Another Form of IPC

We've seen pipes, shared memory, and signals

These forms of IPC assume that the processes are on the same physical machine

Sockets enable IPC between physical machines, typically over the network

## Servers Follow 4 Steps to Use Sockets

These are all system calls, and have the usual C wrappers:

1. `socket`  
Create the socket
2. `bind`  
Attach the socket to some location (a file, IP:port, etc.)
3. `listen`  
Indicate you're accepting connections, and set the queue limit
4. `accept`  
Return the next incoming connection for you to handle

## Clients Follow 2 Steps to Use Sockets

Clients have a much easier time, they use one socket per connection

1. socket  
Create the socket
2. connect  
Connect to some location, the socket can now send/receive data

## The socket System Call Sets the Protocol and Type of Socket

```
int socket(int domain, int type, int protocol);
```

`domain` is the general protocol, further specified with `protocol` (mostly unused)

`AF_UNIX` is for local communication (on the same physical machine)

`AF_INET` is for IPv4 protocol using your network interface

`AF_INET6` is for IPv6 protocol using your network interface

`type` is (usually) one of two options: stream or datagram sockets

## Stream Sockets Use TCP

All data sent by a client appears in the same order on the server

Forms a persistent connection between client and server

Reliable, but may be slow

## Datagram Sockets Use UDP

Sends messages between the client and server

No persistent connection between client and server

Fast but messages may be reordered, or dropped

## The bind System Call Sets a Socket to an Address

```
int bind(int socket, const struct sockaddr *address,  
         socklen_t address_len);
```

socket is the file descriptor returned from the socket system call

There's different sockaddr structures for different protocols

struct sockaddr\_un for local communication (just a path)

struct sockaddr\_in for IPv4, a IPv4 address (e.g. 8.8.8.8)

struct sockaddr\_in6 for IPv6, a IPv6 address (e.g. 2001:4860:4860::8888)



## The `listen` System Call Sets Queue Limits for Incoming Connections

```
int listen(int socket, int backlog);
```

`socket` is still the file descriptor returned from the `socket` system call

`backlog` is the limit of the outstanding (not accepted) connections

The kernel manages this queue, and if full will not allow new connections

We'll set this to `0` to use the default kernel queue size

## The accept System Call Blocks Until There's a Connection

```
int accept(int socket, struct sockaddr *restrict address,  
           socklen_t *restrict address_len);
```

socket is *still* the file descriptor returned from the socket system call

address and address\_len are locations to write the connecting address  
Acts as an optional return value, set both to NULL to ignore

This returns a new file descriptor, we can read or write to as usual

## The connect System Call Allows a Client to Connect to an Address

```
int connect(int sockfd, const struct sockaddr *addr,  
            socklen_t addrlen);
```

`sockfd` is the file descriptor returned by the `socket` system call

The client would need to be using the same protocol and type as the server

`addr` and `addrlen` is the address to connect to, exactly like `bind`

If this call succeeds then `sockfd` is may be used as a normal file descriptor

## Our Example Server Sends “Hello there!” to Every Client and Disconnects

Please see `examples/lecture-16` in your `cs111` repository

Relevant source files: `client.c` and `server.c`

We use a local socket just for demonstration, but you could use IPv4 or IPv6

We use `example.sock` in the current directory as our socket address

Our server uses signals to clean up and terminate from our infinite accept loop

## Instead of read/write There's Also send/recv System Calls

These system calls are basically the same thing, except they have flags

Some examples are:

MSG\_OOB — Send/receive out-of-band data

MSG\_PEEK — Look at data without reading

MSG\_DONTROUTE — Send data without routing packets

Except for maybe MSG\_PEEK, you do not need to know these

sendto/recvfrom take an additional address

The kernel ignores the address for stream sockets (there's a connection)

## Sockets Form a Basis For Distributed Systems

You can use a remote procedure call (RPC) to run a function on another machine  
We send a request to a server, it executes and sends a reply back

RPC can be done asynchronously, your process sends a request and doesn't block  
You can continue working in other threads to keep the process running

Easy to program, looks like a normal function running on a server  
Details are hidden, converting data to bytes to send (wire format)

## A RPC Can Fail in More Ways than a Regular Call

Maybe a lost packet, interrupted connection, a slow or unresponsive server

We don't know what happened on the server:

- It never saw the request
- It crashed while executing the request (or just after)
- It executed the request, but there was a network error during the reply

## “Best Effort” RPC Scheme Retries Multiple Times

It sends a request, and if there's no response within a timeout, resend

Try resending a set number of times before giving up and returning an error

What if the function increments a variable on the server?

The server could receive the packet after the timeout period

The client resends it and a single call increments twice (or more)!



## An “At Most Once” Scheme is Better

The client includes a unique ID with each request

If there's duplicate requests, send the previous reply

This is particular useful if the function is not idempotent

Idempotent means one or more calls have the same effect

You may want something like “exactly once”, and that's complex (another course)

## File Systems Can Be Networked as Well

For example: NFS is a protocol designed to appear as a file system  
However, the data is on a server accessed over the network

A server accessed using protocols like NFS is called a NAS  
NAS stands for Network-attached storage

A NAS is typically a single server with a large RAID array

## You Could Distribute the File System Across Multiple Servers

Google File System (GFS) is an example of a distributed file systems

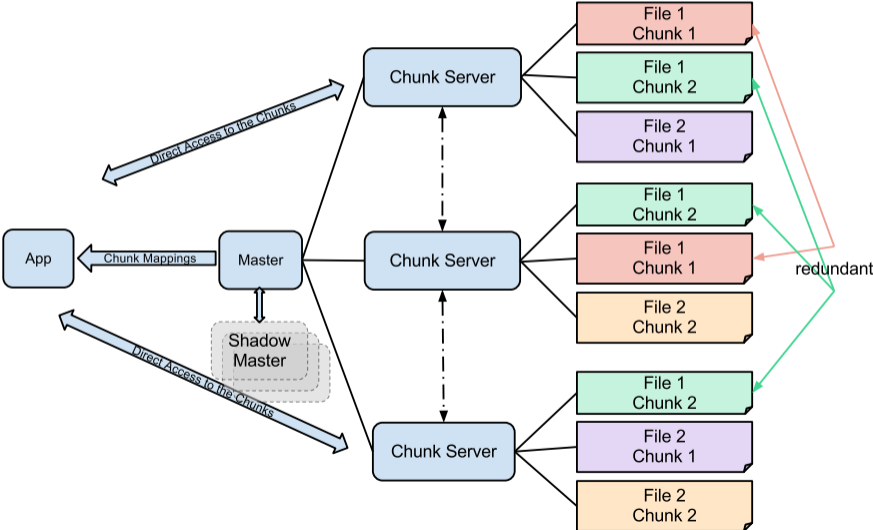
It breaks files up into fixed-sized 64 MB chunks and replicates it (e.g. 3 times)

Each chunk gets an immutable and unique identifier

A single server manages the file to chunk mappings

Clients then access the chunk servers directly

# GFS Architecture



## GFS Uses Weak Consistency to Scale

What about consistency? What if two clients modify a file using different servers?

Ideally we'd like the same result as if it was a single server

The single server ensures file namespace mutations are atomic

All the metadata is atomic

For "record appends" (data) is appended "at least once"

It's up to clients to detect duplicates

This system is tailored for large Google scale files

We won't go into all the details (there's a SOSP paper if you're interested)

## You Can Distribute Other Tasks Too

- One machine can only handle so many connections

  - One machine only has so much bandwidth

- You can have multiple web servers all serving the same content

  - You can use a load balancer as your websites entry point

    - A loader balancer selects the server with the lowest load

## There Are Limits, Sometimes People Maliciously Overload Systems

A denial-of-service attack occurs when you flood a target with bogus requests  
Sometimes the requests look legitimate, and are hard to detect

A distributed denial-of-service (DDoS) occurs from multiple sources  
It's even harder to block, often backed by a botnet

# Distributed Systems Start with Sockets

There's networking and distributed systems courses!

However, today we learned the basics:

- Sockets are IPC across physical machines
  - Sockets require an address (e.g. local and IPv4/IPv6)
  - There are two types of sockets: stream and datagram
  - Servers need to bind to an address, listen, and accept connections
  - Clients need to connect to an address
- Networked file systems (NFS)
- Distributed file systems (GFS)
- Denial-of-service attacks are a unique concern