Inetd and Implementing Protocols

Gopher, Finger and the web that wasn't

Joseph Hallett

February 6, 2025

This is going to be a bit of a silly lecture…

Sorry about that.

We've talked a little bit about web protocols

Matt's spoken to you about HTTP

- We practiced sending HTTP responses with netcat (nc) in the lab

```
HTTP/1.1 200 OK
Content-type: text/plain
Content-length: 16
Connection: close

Hello over HTTP!
```

If you sent it into a webbrowser you could make some text appear…

- If you loaded up a webserver you could send it back and build a website

Okay but what actually is HTTP?

If we tell you what to write you can echo it…

- But where is all this documented?
- What other things can it do?
- Can I write extensions to HTTP to do other things?
- Can I write whole new protocols to do other things?

Okay but what actually is HTTP? (Answers edition)

If we tell you what to write you can echo it…

But where is all this documented? In standards

What other things can it do? Loads!

Can I write extensions to HTTP to do other things? Sure

Can I write whole new protocols to do other things? Sure

And how do you do it?

In this lecture…

- We're going to look at the IETF's RFC specs
- We're going to talk a little bit about programming with sockets
- We're going to show you `inetd` as an easy way to write servers
  And for the sillier bit… a bit of history
  - Y'know we didn't settle on the web being HTTP and HTML and Social Media platforms immediately
  - There were competing protocols…
  - Some are still used…

  In the lab…
- We're going to try implementing one of the simpler ones!

Specifications

So where is HTTP documented?

HTTP 1.0 (the original) is documented in IETF RFC 1945

Network Working Group
Request for Comments: 1945
Category: Informational

T. Berners-Lee
MIT/LCS
R. Fielding
UC Irvine
H. Frystyk
MIT/LCS
May 1996

Hypertext Transfer Protocol -- HTTP/1.0

Status of This Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

IESG Note:

The IESG has concerns about this protocol, and expects this document to be replaced relatively soon by a standards track document.

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed

What is this?

Plaintext specification in coding fonts looks a bit sketchy, no?

- Honestly this is what they do

The IETF is the Internet Engineering Task Force and they're the standards organization for the Internet.

- Its a non-profit
- Anyone can join by taking part
- They decide what the internet should be by rough consensus

As part of this they publish requests for comments that describe what they think the protocols should be

- Anyone can take part in the discussions
  - But not companies!
- Running code is best
- Very conservative towards change
- Sometimes a bit trolly…

Lets have a look at an RFC in a bit more detail…

RFC 2324 is the Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)

- Proposed back in 1998 as a protocol for controlling Internet Connected Smart Coffee Pots
- Intended sort of as a joke then… but quite prescient now…
- See also Internet Toaster protocol (RFC 2235)

```
2. HTCPCP Protocol

   The HTCPCP protocol is built on top of HTTP, with the addition of a
   few new methods, header fields and return codes.  All HTCPCP servers
   should be referred to with the "coffee:" URI scheme (Section 4).
```

BREW

HTTP has GET and POST commands… HTCPCP has a few more

2.1.1 The BREW method, and the use of POST

Commands to control a coffee pot are sent from client to coffee server using either the BREW or POST method, and a message body with Content-Type set to "application/coffee-pot-command".

A coffee pot server MUST accept both the BREW and POST method equivalently. However, the use of POST for causing actions to happen is deprecated.

Coffee pots heat water using electronic mechanisms, so there is no fire. Thus, no firewalls are necessary, and firewall control policy is irrelevant. However, POST may be a trademark for coffee, and so the BREW method has been added. The BREW method may be used with other HTTP-based protocols (e.g., the Hyper Text Brewery Control Protocol).

2.1.4 WHEN method

When coffee is poured, and milk is offered, it is necessary for the holder of the recipient of milk to say "when" at the time when sufficient milk has been introduced into the coffee. For this purpose, the "WHEN" method has been added to HTCPCP. Enough? Say WHEN.

A few new header fields

2.2.2 New header fields

2.2.2.1 The Accept-Additions header field

In HTTP, the "Accept" request-header field is used to specify media types which are acceptable for the response. However, in HTCPCP, the response may result in additional actions on the part of the automated pot. For this reason, HTCPCP adds a new header field, "Accept-Additions":

```
Accept-Additions = "Accept-Additions" ":"
                   #( addition-range [ accept-params ] )

 addition-type   = ( "*"
                   | milk-type
                   | syrup-type
                   | sweetener-type
                   | spice-type
                   | alcohol-type
                   ) *( ";" parameter )
 milk-type       = ( "Cream" | "Half-and-half" | "Whole-milk"
                   | "Part-Skim" | "Skim" | "Non-Dairy" )
 syrup-type      = ( "Vanilla" | "Almond" | "Raspberry"
                   | "Chocolate" )
 alcohol-type    = ( "Whisky" | "Rum" | "Kahlua" | "Aquavit" )
```

And new return codes...

2.3 HTCPCP return codes

Normal HTTP return codes are used to indicate difficulties of the HTCPCP server. This section identifies special interpretations and new return codes.

2.3.2 418 I'm a teapot

Any attempt to brew coffee with a teapot should result in the error code "418 I'm a teapot". The resulting entity body MAY be short and stout.

Wahey

We can see all the message types and codes

- So long as we follow the protocol we should be able to talk to anything that understands it
- So long as we implement the protocol anything that talks it should be able to talk to us

We may now brew coffee!

Reading isn't everything…

By reading the protocol specs we can work out how different devices talk…

- But it doesn't tell us how to actually say things
- We still need to get the messages from the client to the server.
- How do all these servers, and tool do it?

When we're dealing with files we have a few primitives

- open, close, read and write systemcalls (and the higher level APIs)
- What do we have for dealing with remote connections?

Sockets

The POSIX way of doing things is with sockets

- Similar to a file descriptor (low level file)
- But reads and writes go via the network

If we were doing things in C for a file it would look like:

```
int fd = open("myfile.txt", O_RDWR);
char buffer[BUFSIZ];
ssize_t len = read(fd, buffer, sizeof(buffer)-1);
write(fd, buffer, len);
```

Or via the more usual `fopen()`, `fread()` and `fwrite()` C API

What's it going to look like for a socket?

What's it going to look like for a socket?

Well it depends…

- The server (the thing waiting for a connection) is going to need to open a socket and start listening to see who to talk to
- The client (the thing starting a connection) is going to need to know who it wants to talk to

Code differs a little depending on what you're doing.

We'll go over both

But I'm gonna skip the error handling! Read the `man` pages!

High-level plan

- Set up connection details
- Create the socket and connect to it

For client

- Start talking

For server

- Wait for someone to connect
- Fork so we're not blocked
- Start talking

Client!

First we need to find where we're going to connect to…

```
struct addrinfo *servinfo, hints;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // IPv4 or IPv6 or don't care?
hints.ai_socktype = SOCK_STREAM; // TCP or UDP style?

// Fills servinfo with a linked list of servers
getaddrinfo("bristol.ac.uk", "80", &hints, &servinfo);
```

`hints` `struct` says how to connect

`getaddrinfo()` fills in information you need to connect (via DNS)

Finding us a server

Why might you get more than one server?

- IPv4 vs IPv6? Multiple network devices?

Lets connect to the first available one

```
struct addrinfo *servinfo, hints;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // IPv4 or IPv6 or don't care?
hints.ai_socktype = SOCK_STREAM; // TCP or UDP style?

// Fills servinfo with a linked list of servers
getaddrinfo("bristol.ac.uk", "80", &hints, &servinfo);

int sockfd;
for (; servinfo != NULL; servinfo = servinfo->ai_next) {
  sockfd = socket(servinfo->ai_family, servinfo->ai_socktype, servinfo->ai_protocol);
  connect(sockfd, servinfo->ai_addr, servinfo->ai_addrlen);
  break;
}
```

`socket()` creates the socket (remember to close it!)

`connect()` connects it to the server (and will fail if the server can't be reached)

Finally we can chat…

If we haven't had an error so far we can now chat!

```
struct addrinfo *servinfo, hints;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // IPv4 or IPv6 or don't care?
hints.ai_socktype = SOCK_STREAM; // TCP or UDP style?

// Fills servinfo with a linked list of servers
getaddrinfo("bristol.ac.uk", "80", &hints, &servinfo);

int sockfd;
for (; servinfo != NULL; servinfo = servinfo->ai_next) {
  sockfd = socket(servinfo->ai_family, servinfo->ai_socktype, servinfo->ai_protocol);
  connect(sockfd, servinfo->ai_addr, servinfo->ai_addrlen);
  break;
}

// To receive bytes
char buf[BUFSIZ];
int numbytes;

send(sockfd, "GET␣/\n\n", 7, 0);
numbytes = recv(sockfd, buf, BUFSIZ, 0);
for (int i=0; i < numbytes; i++) putchar(buf[i]);
```

And we get back!

```
HTTP/1.0 302 Moved Temporarily
Location: https:///
Server: BigIP
Connection: close
Content-Length: 0
```

`send()` sends a message (like `write()`)

`recv()` blocks until it gets a response (like `read()`)

Yay we can make a connection manually now!

Time to write a server!

Server setup

Right lets set up our server connection now!

```
struct addrinfo hints, *servinfo;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // Still don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // Still TCP style talk
hints.ai_flags = AI_PASSIVE; // Use my IP

getaddrinfo(NULL, "8080", &hints, &servinfo);
```

Only real difference here is were not connecting to anywhere and instead listening on our own IP.

Start listening

```
<<server-getaddrinfo>>

int sockfd;
int yes = 1;
for (; servinfo != NULL; servinfo = servinfo->ai_next) {
  sockfd = socket(servinfo->ai_family, servinfo->ai_socktype, servinfo->ai_protocol);
  setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof int);
  bind(sockfd, servinfo->ai_addr, servinfo->ai_addrlen);
  break;
}
listen(sockfd, 10);
```

`setsockopt( … SO_REUSEADDR … )` avoids port in use errors

`bind()` attach your program to the socket but don't connect to a specific connection yet

`listen()` listen to see if someone has connected to you (and keep 10 a backlog of 10 connection if multiple people try at once)

A wild connection appears!

```
int new_fd;
struct sockaddr_storage their_addr;
while (1) {
  new_fd = accept(sockfd, &their_addr, sizeof their_addr);
  if (!fork()) {
    // Or whatever...
    send(new_fd, "PING", 4, 0);
    }
  close(new_fd);
 }
```

- We `accept()` the connection and create a `new_fd` to talk to this connection with
- We `fork()` to create a running copy of our program and have the child deal with the new connection then die, and we continue dealing with new connections.

Wahey we have a server and a client!

Please don't actually write network code like this!

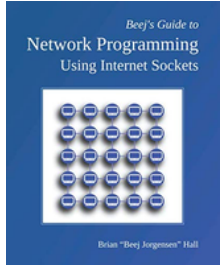
- There's no error handling (and you'll get a tonne of errors)
- Dead forks should be reaped otherwise you'll leave junk in the process list

Beej's Guide to Network Programming Using Internet Sockets

- Even has a little bit on Windows `:-0`

Free online (I read it every time I have to do this)

- `https://beej.us/guide/bgnet/`



So that's low level network programming

Wrappers for this exist in your favorite programming language

- …and even for C
- …but this is the low-level (ish) version of it

Okay there's a little bit of set up to talk to a socket

- but once you have one its `send()` and `recv()` in place of `write()` and `read()`
- Can we abstract this more?

(If you wanna get super low check out your kernel's TCP/IP stack)

Abstraction, abstraction, abstraction

All we're really doing is listening on a port

- When something connects run a service with an input and output set up down that socket
- Shutting it down when done

Why do we have to care that we're talking on a socket at all?

- Why can't we talk on the usual `stdin`, `stdout`
- And leave the connection set up to something else?

Inetd

The Internet Services Daemon!

- An internet super service (a service for services)
- Does exactly this
- Will run whatever with `stdin`, `stdout` set up down the socket

First appeared in 1978

- Really rather forgotten about now `:-(`
- Considered somewhat insecure
  - Mostly because it had a large and prominent attack surface

"Modern" versions exist

- But tend to do nothing by default
  - `xinetd` on Linux
  - Similar functionality in `systemd` on Linux
  - Similar functionality in `launchd` on MacOS
- Consult your OS's manual

My server runs OpenBSD

It's like Linux but worse in every conceivable way

- It's simple (to the point of being a bit slow and annoying)
- You can debug it (but it mostly just works)
- It's attack surface is small (so maybe secure-ish?)

It uses the old `inetd` format…

- So in `/etc/inetd.conf` it has…

Service	Kind	Conn	Options	User	Process
finger	stream	tcp	nowait	_fingerd	/usr/libexec/fingerd
finger	stream	tcp6	nowait	_fingerd	/usr/libexec/fingerd
gopher	stream	tcp	nowait	_gophernicus	/usr/local/libexec/in.gophernicus

Cool, well it looks like I'm running services for `finger` (IPv4 and IPv6) and `gopher` (just IPv4)

- `nowait` means allow running more than one at a time
- Each service will run as a separate user
- Programs it will run are in `/libexec/` not `/bin/`?

If we check `/etc/services`

Service	Runs on
gopher	70/tcp
gopher	70/udp
finger	79/tcp

What do these /libexec/ programs look like?

In the case of my `fingerd` it is literally a shellscript

- `inetd` connects `stdin`, `stdout` to the socket
- I modified it to do a little bit less than the default `fingerd`
- Could be *C* and `scanf()` and `printf()`
- Or Python and `read()` and `print()`
- Or whatever else you like!

Really simple

- I can run my Gopherhole and Finger server hapilly on a cheap VPS `:-D`
- You could run a whole HTML server like this (if you were mad)

Recap

So…

Network protocols documented in IETF RFCs

- Just text going over network
- Coffee pots have a language?

Socket programming to write servers

- Bit fiddly but not that bad if you don't have an existing one

Inetd to run simple network protocols

- Read the docs
- Not the modern way to do things

In the lab we'll start implementing network protocols…

I feel I've skipped something...

WTF are finger and gopher?

The web that wasn't

The modern internet is predominantly based around HTTP

- Online content is served on website through HTTP
- We have social media running on websites served through HTTP
- Even chatrooms like Discord run in a web app served with HTTP

It wasn't always this way

- Every task used to have its own protocol
- HTTP wasn't the only way to deliver a website

I like computing history…

- Lets talk about some of the ways things didn't go

In the distant past…

Way back in the early weeks of this course we created users…

- It asked for a real name, a telephone number, and a room?
- Why?!

Early UNIX

In the olden times computers didn't really sit on your desk

- You had big mainframes in the basement of your Uni you dialed in to from the little terminal on your desk
- Suppose someone was breaking things…
  - Would be nice to be able to contact them to ask what they're doing
- Only a dozen or so networked computers on the early pre-internet
  - Would be nice to know who you're talking to when an outside user dials in

It would be nice if you could put your finger on who they were

Like an old address book…

RFC 1288 Finger Protocol

You send a TCP message over port 79

- Couple of flags for how much info you want
- If it's a username, then finger that user
- If it's an @ list all the users on that system
- If it's a username @ an address the forward the fingering message on to the address
  - Like a jump host in SSH

Server sends you back whatever it fancies as a response

And for example

If you install a finger program...

```
finger fishfinger@tilde.club
```

```
[tilde.club]

Login: fishfinger
Name:
Directory: /home/fishfinger
Shell: /bin/bash

No mail found.

Plan:
Teaching my students about the old ways on the web!
```

.plan

So what did we get back?

- My name and shell
- The sessions where I was logged in
- Whether I'd checked my email recently (hey it was the 90s)

My plan…

- You could stick a `.plan` file in your home directory to tell people what you were working on
- People absolutely used them as an early precursor to Twitter…

John Carmack's `.plan` files

John Carmack kept logs of the work he did every day from 1996–2010 in a `.plan` file

- He programmed DOOM and Quake with John Romero, founded ID Software, invented modern FPS

Initially just lists of jobs…

- But also notes about what ID were doing
- Cool development tricks
- How game engines were designed
- Frustrations with porting

Archived online to read

- `https://github.com/ESWAT/john-carmack-plan-archive`
- If you're interested in game development, or the history of our industry you can learn from one of the greats
- (Also read Fabien Sanglard's Game Engine Black Book: DOOM)

So what?

So some crusty old programmer kept logs in a text file?

- So what?

Disadvantages to a blog/twitter

- No analytics
- No advertising
- Almost no overhead
- No images (except text art, or inventive tricks with escape codes)

Mastodon

Decentralized alternative to Twitter, run by no one

- Check out Grunfink's SNAC2 if you fancy running it

Instead of one central source of accounts (like Twitter or BlueSky) anyone can run a server

- ActivityPub protocol shares ~~tweets~~ toots between servers

But how do you discover who is on what servers?

RFC 7033 WebFinger

```
curl https://mastodon.social/.well-known/webfinger?resource=acct:Gargron@mastodon.social
```


```
{ "subject":"acct:Gargron@mastodon.social",
  "aliases":["https://mastodon.social/@Gargron",
             "https://mastodon.social/users/Gargron"],
  "links":[{"rel":"http://webfinger.net/rel/profile-page",
            "type":"text/html",
            "href":"https://mastodon.social/@Gargron"},
           {"rel":"self",
            "type":"application/activity+json",
            "href":"https://mastodon.social/users/Gargron"},
           {"rel":"http://ostatus.org/schema/1.0/subscribe",
            "template":"https://mastodon.social/authorize_interaction?uri={uri}"},
           {"rel":"http://webfinger.net/rel/avatar",
            "type":"image/png",
            "href":"https://files.mastodon.social/accounts/avatars/000/000/001/original/a0a49d80c3de5f
```


Neat huh? What was just plaintext is now an HTTP GET and some JSON!

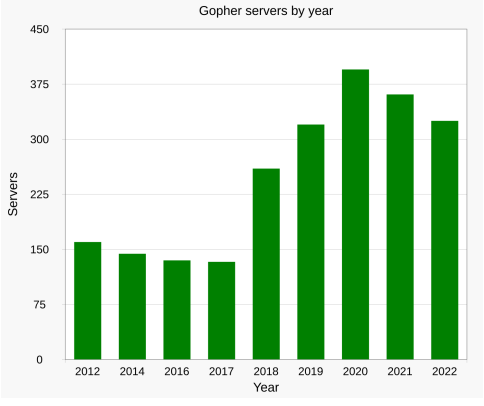
Okay so we could replace social networks with fingering…

What about websites?

Has HTTP and HTML had any competitors?

RFC 1436 Gopher

- Developed in 1991 at the University of Minesotta
- Intended for directory listing and file retrieval
- Far more structured than HTML
- Servers available (I use Gophernicus)



(<sarcasm> See modern and relevant still! </sarcasm>)

Gophermap

```
To create a site you write a Gophermap. This is really just plaintext plus links
but only certain link types are allowed; for example:

1Folder of blogs /blogs gopher.server 70
0Notes /notes.txt gopher.server 70
IMy cat /pictures/nigel.png gopher.server 70
hGoogle URL:http://www.google.com
gThis is fine /pictures/this-is-fine.gif gopher.server 70

Some other types and tricks ...but that's essentially it

0Finger my user jo gopher.server 79

Firefox used to support Gopher :-( but other browsers exist like

hCastor URL:https://sr.ht/~julienxx/Castor
hChawan URL:https://sr.ht/~bptato/chawan
hLinks2 URL:http://links.twibright.com
```

Dead gophers

Gopher ultimately failed compared to more general HTML

- Attempted to commercialize in 1993 and charge for using protocol
- Mimetypes beat out fixed lists of link types
- You can't style a Gopherhole
- Encryption?
- Only ~300 servers still going :-(

Yet some people still love it…

- Many Tilde/Shell servers will still give you a Gopherhole
  - `https://tilde.club` `https://tilde.town` `https://sdf.org`
- Ads on Gopherspace aren't a thing
- Nor really are analytics

Gemini Protocol and the Smolweb

People are starting to get fed up of the modern web

- Twitter and Facebook are hateful
- Websites are bloated and full of malware
- We all block ads right?
  - If not see me and Matt in the next lab
  - But does your Granny

Smolweb makes it for users and admins again

Go run a gopherhole… its easy.
Make yourself fingerable!

- 2 years ago you could finger each other on the lab machines `:-(`

Gemini protocol aims to simplify

- Much more focus on content (no CSS/JS)
- Simple text of Gopher
- But with some of the good ideas of HTML
- `https://geminiprotocol.net`
- Few thousand servers
  - (and often mirrored as Gopherholes for the curmudgeonly)!

That's all folks!

We covered:

- Protocol specs in the IETF RFCs
- Simple socket programming
- Inetd
- Finger and Gopher