15-213

"The course that gives CMU its Zip!"

Internet Services
Nov 30, 2000

Topics

- A tour of the Tiny Web server
- The DNS service

The Tiny Web server

Tiny is a minimal Web server written in 250 lines of C.

Serves static and dynamic content with the GET method.

- text files, HTML files, GIFs, and JPGs.
- supports CGI programs

Neither robust, secure, nor complete.

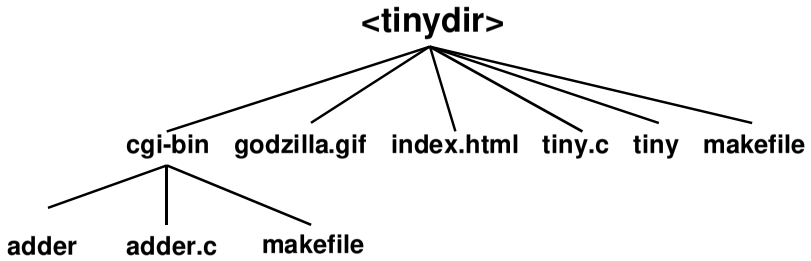
- It doesn't set all of the CGI environment variables.
- Only implements GET method.
- Weak on error checking.

Interesting to study as a template for a real Web server.

Ties together many of the subjects we have studied this semester:

- VM (mmap)
- process management (fork, wait, exec)
- network programming (sockets interface to TCP)

The Tiny directory hierarchy



Usage:

- `cd <tinydir>`
- `tiny <port>`

Serves static content from <tinydir>

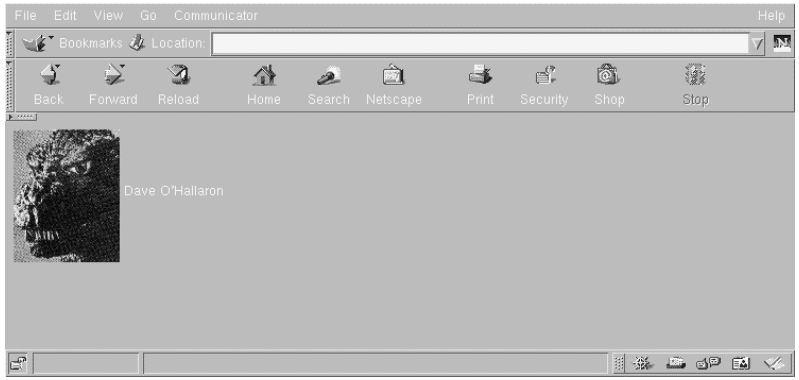
- `http://<host>:<port>`

Serves dynamic content from <tinydir>/cgi-bin

- `http://<host>:<port>/cgi-bin/adder?1&2`

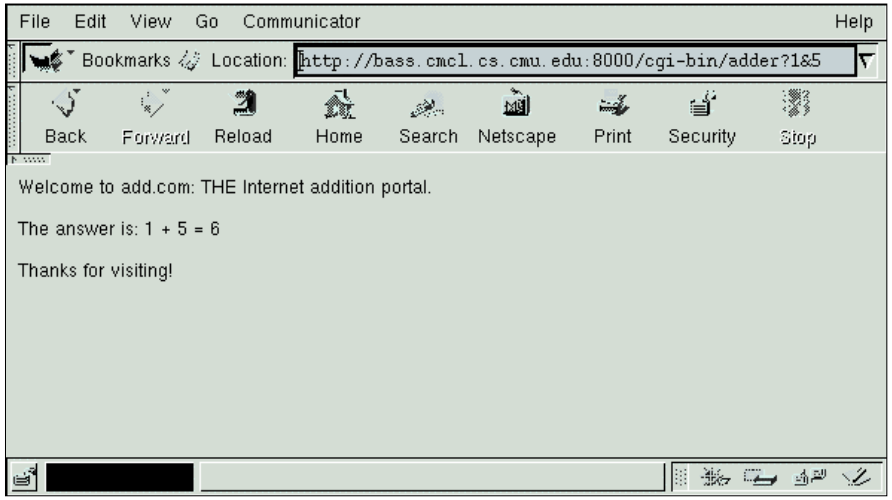
Serving static content with tiny

http://<host>:<port>/index.html



Serving dynamic content with Tiny

http://<host>.<port>/cgi-bin/adder?1&5



Tiny error handler

```
/*
 * error - wrapper for perror used for bad syscalls
 */
void error(char *msg) {
  perror(msg);
  exit(1);
}
```


Tiny: `cerror`

`cerror()` returns HTML error messages to the client.

- **`stream` is the `connfd` socket opened as a Unix stream so that we can use handy routines such as fprintf and fgets instead of read and write.**

```
/*
 * cerror - returns an error message to the client
 */
void cerror(FILE *stream, char *cause, char *errno,
            char *shortmsg, char *longmsg) {
  fprintf(stream, "HTTP/1.1 %s %s\n", errno, shortmsg);
  fprintf(stream, "Content-type: text/html\n");
  fprintf(stream, "\n");
  fprintf(stream, "<html><title>Tiny Error</title>");
  fprintf(stream, "<body bgcolor=""ffffff"">\n");
  fprintf(stream, "%s: %s\n", errno, shortmsg);
  fprintf(stream, "<p>%s: %s\n", longmsg, cause);
  fprintf(stream, "<hr><em>The Tiny Web server</em>\n");
}
```


Tiny: main loop

Tiny loops continuously, serving client requests for static and dynamic content.

```
/* open listening socket */
...

while(1) {
  /* wait for connection request */
  /* read and parse HTTP header */
  /* if request is for static content, retrieve file */
  /* if request is for dynamic content, run CGI program */
}
```

Tiny: open listening socket

```
/* open socket descriptor */
 listenfd = socket(AF_INET, SOCK_STREAM, 0);
 if (listenfd < 0)
   error("ERROR opening socket");

 /* allows us to restart server immediately */
 optval = 1;
 setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
            (const void *)&optval , sizeof(int));

 /* bind port to socket */
 bzero((char *) &serveraddr, sizeof(serveraddr));
 serveraddr.sin_family = AF_INET;
 serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
 serveraddr.sin_port = htons((unsigned short)portno);
 if (bind(listenfd, (struct sockaddr *) &serveraddr,
          sizeof(serveraddr)) < 0)
   error("ERROR on binding");

 /* get us ready to accept connection requests */
 if (listen(listenfd, 5) < 0) /* allow 5 requests to queue up */
   error("ERROR on listen");
```


Tiny: accept a connection request

```
  clientlen = sizeof(clientaddr);
  requestno = 0;
  while (1) {

    /* wait for a connection request */
    connfd = accept(listenfd, (struct sockaddr *) &clientaddr,
                    &clientlen);
    if (connfd < 0)
      error("ERROR on accept");

#ifdef DEBUGDOT
    if ((requestno % 50) == 0)
      printf("\n%6d", requestno);
    else
      printf(".");
    fflush(stdout);
#endif
    requestno++;
```


Tiny: read HTTP request

```
    /* open the connection socket descriptor as a stream */
    if ((stream = fdopen(connfd, "r+")) == NULL)
      error("ERROR on fdopen");

    /* get the HTTP request line */
    fgets(buf, BUFSIZE, stream);
    sscanf(buf, "%s %s %s\n", method, uri, version);

    /* tiny only supports the GET method */
    if (strcasecmp(method, "GET")) {
      cerror(stream, method, "501", "Not Implemented",
             "Tiny does not implement this method");
      fclose(stream);
      continue;
    }

    /* read (and ignore) the HTTP headers */
    fgets(buf, BUFSIZE, stream);
    while(strcmp(buf, "\r\n")) {
      fgets(buf, BUFSIZE, stream);
    }
```


Tiny: parse the URI in the HTTP request

```
    /* parse the uri */
    if (!strstr(uri, "cgi-bin")) { /* static content */
      is_static = 1;
      strcpy(cgiargs, "");
      strcpy(filename, ".");
      strcat(filename, uri);
      if (uri[strlen(uri)-1] == '/')
        strcat(filename, "index.html");
    }
    else { /* dynamic content: get filename and its args */
      is_static = 0;
      p = index(uri, '?'); /* ? separates file from args */
      if (p) {
        strcpy(cgiargs, p+1);
        *p = '\0';
      }
      else {
        strcpy(cgiargs, "");
      }
      strcpy(filename, ".");
      strcat(filename, uri);
    }
```

Tiny: access check

A real server would do extensive checking of access permissions here.

```
/* make sure the file exists */
if (stat(filename, &sbuf) < 0) {
  cerror(stream, filename, "404", "Not found",
         "Tiny couldn't find this file");
  fclose(stream);
  continue;
}
```


Tiny: serve static content

A real server would serve many more file types.

```
/* serve static content */
if (is_static) {
  if (strstr(filename, ".html"))
    strcpy(filetype, "text/html");
  else if (strstr(filename, ".gif"))
    strcpy(filetype, "image/gif");
  else if (strstr(filename, ".jpg"))
    strcpy(filetype, "image/jpg");
  else
    strcpy(filetype, "text/plain");

  /* print response header */
  fprintf(stream, "HTTP/1.1 200 OK\n");
  fprintf(stream, "Server: Tiny Web Server\n");
  fprintf(stream, "Content-length: %d\n", (int)sbuf.st_size);
  fprintf(stream, "Content-type: %s\n", filetype);
  fprintf(stream, "\r\n");
  fflush(stream);
  ...
```


Tiny: serve static content (cont)

Notice the use of mmap() to copy the file that the client requested back to the client, via the stream associated with the child socket descriptor.

```
/* Use mmap to return arbitrary-sized response body */
if ((fd = open(filename, O_RDONLY)) < 0)
   error("ERROR in mmap fd open");
if ((p = mmap(0, sbuf.st_size, PROT_READ, MAP_PRIVATE, fd, 0)) < 0)
   error("ERROR in mmap");
fwrite(p, 1, sbuf.st_size, stream);
if (munmap(p, sbuf.st_size) < 0)
   error("ERROR in munmap");
if (close(fd) < 0)
   error("ERROR in mmap close");
```


Tiny: serve dynamic content

A real server would do more complete access checking and would initialize all of the CGI environment variables.

```
/* serve dynamic content */
else {
  /* make sure file is a regular executable file */
  if (!(S_IFREG & sbuf.st_mode) || !(S_IXUSR & sbuf.st_mode)) {
    cerror(stream, filename, "403", "Forbidden",
           "You are not allow to access this item");
    fclose(stream);
    continue;
  }

  /* initialize the CGI environment variables */
  setenv("QUERY_STRING", cgiargs, 1);

  ...
```

Tiny: serve dynamic content (cont)

Next, the server sends as much of the HTTP response header to the client as it can.

Only the CGI program knows the content type and size.

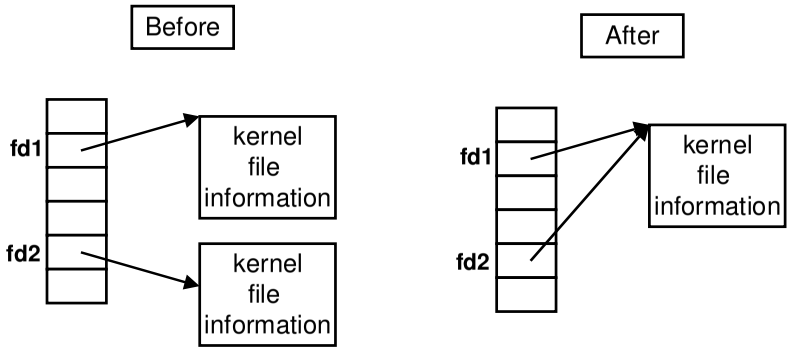
Notice that we don't mix stream (fprintf) and basic (write) I/O. Mixed outputs don't generally go out in program order.

```
/* print first part of response header */
sprintf(buf, "HTTP/1.1 200 OK\n");
write(connfd, buf, strlen(buf));
sprintf(buf, "Server: Tiny Web Server\n");
write(connfd, buf, strlen(buf));

...
```


dup system call

`dup2(fd1, fd2)` makes descriptor `fd2` to be a copy of `fd1`, closing `fd2` if necessary.



Tiny: serve dynamic content (cont)

`dup2(fd1, fd2)` makes descriptor `fd2` to be a copy of `fd1`, closing `fd2` if necessary.

```
/* create and run the child CGI process */
pid = fork();
if (pid < 0) {
  perror("ERROR in fork");
  exit(1);
}
else if (pid > 0) { /* parent */
  wait(&wait_status);
}
else { /* child */
  close(0); /* close stdin */
  dup2(connfd, 1); /* map socket to stdout */
  dup2(connfd, 2); /* map socket to stderr */
  if (execve(filename, NULL, environ) < 0) {
    perror("ERROR in execve");
  }
}
} /* end while(1) loop */
```

Notice the use of libc's global `environ` variable in the `execve` call.

The `dup2` calls are the reason that the bytes that the child sends to `stdout` end up back at the client.

CGI program: `adder.c`

```
int main() {
  char *buf, *p;
  char arg1[BUFSIZE];
  char arg2[BUFSIZE];
  char content[CONTENTSIZE];
  int n1, n2;

  /* parse the argument list */
  if ((buf = getenv("QUERY_STRING")) == NULL) {
    exit(1);
  }

  p = strchr(buf, '&');
  *p = '\0';
  strcpy(arg1, buf);
  strcpy(arg2, p+1);
  n1 = atoi(arg1);
  n2 = atoi(arg2);
```

`adder.c` CGI program (cont)

```
 /* generate the result */
  sprintf(content, "Welcome to add.com: THE Internet addition\
portal.\n<p>The answer is: %d + %d = %d\n<p>Thanks for visiting!\n",
          n1, n2, n1+n2);

  /* generate the dynamic content */
  printf("Content-length: %d\n", strlen(content));
  printf("Content-type: text/html\n");
  printf("\r\n");
  printf("%s", content);
  fflush(stdout);
  exit(0);
}
```


Tiny sources

The complete Tiny hierarchy is available from the course Web page.

- follow the "Documents" link.

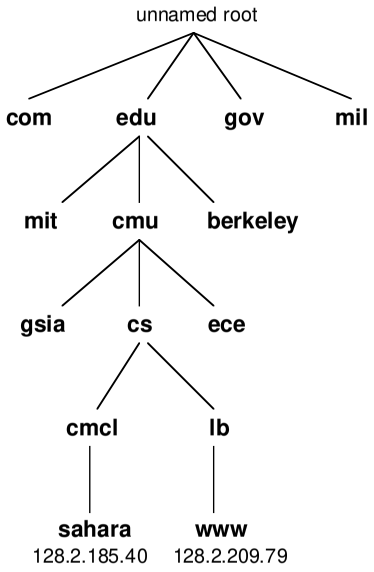
Hierarchical domain name space

Until 198x, domain name/IP address mapping maintained in HOSTS.TXT file at SRI.

Each new host manually entered and copied to backbone routers.

Explosive growth rendered HOSTS.TXT approach impractical.

Replaced by Domain Name System in 198x.



DNS

Worldwide distributed system for mapping domain names to IP addresses (and vice versa).

Implemented as a collection of cooperating servers called *name servers*.

Name servers perform lookups for DNS clients

- user programs
  - `gethostbyname(), gethostbyaddr()`
- `nslookup`
  - stand-alone client with command line interface

```
kittyhawk> nslookup bass.cmcl
Server:  localhost
Address:  127.0.0.1

Non-authoritative answer:
Name:    bass.cmcl.cs.cmu.edu
Address:  128.2.222.85
```

Zones

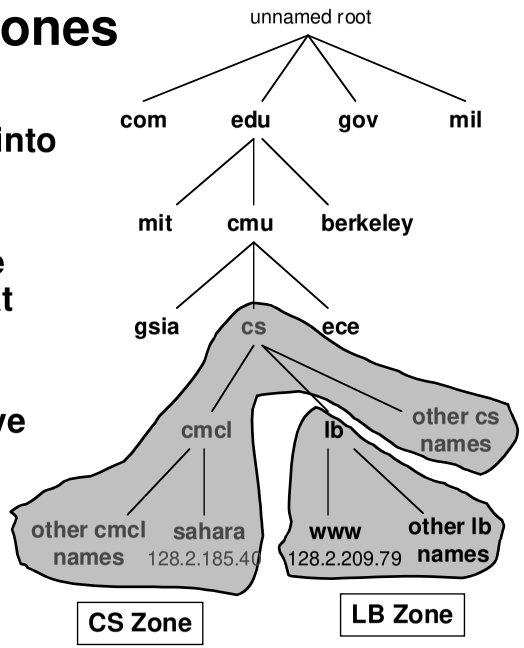
Domains are partitioned into zones.

Each zone has multiple name servers that store info about names in that zone.

- CS zone has 4 servers

One server is authoritative

- the others get copies of the authoritative server's data



Zone databases

Each name server keeps a database with information about each name in its zone.

Examples of info (`type`: description)

- `A`: IP address
- `NS`: name servers for zone
- `SOA`: "start of authority" indicates authoritative server
- `WKS`: well known services running on that host
- `HINFO`: host info (OS and machine type)
- `PTR`: domain name ptr (if this subdomain has its own server)

Zone transfers

Clients can inspect the contents of a zone database via a copy operation called a *zone transfer*.

- all info of a particular type or types (`A`, `NS`, etc) of info for each domain name in the entire zone is copied from server to client.

Servers can control which client machines are allowed to perform zone transfers

Example: zone transfer of `cs.cmu.edu` (Types `A` & `PTR`)
(note: this is the default for `nslookup`)

```
...
SAHARA.CMCL       128.2.185.40
...
LB                server = ALMOND.SRV.CS.CMU.EDU
LB                server = PECAN.SRV.CS.CMU.EDU
...
POSTOFFICE        128.2.181.62
...
```


Zone transfers (cont)

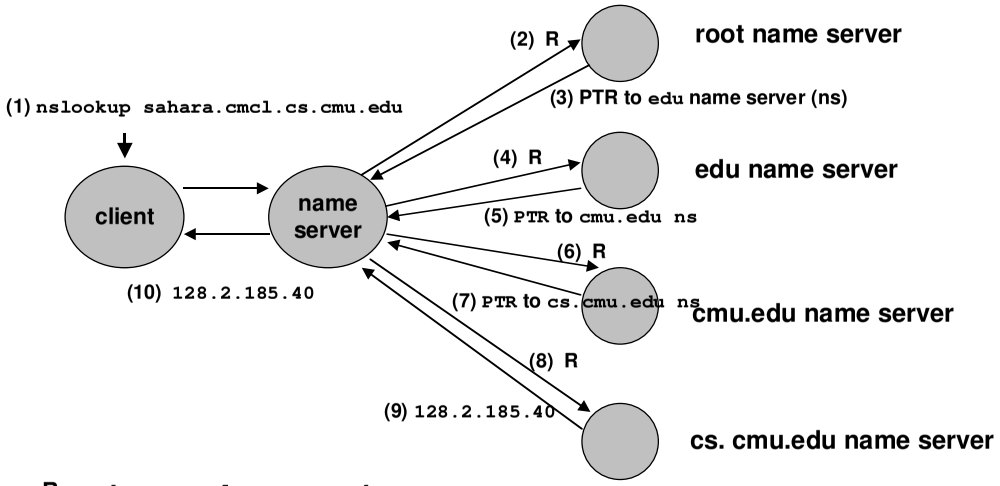
Example: zone transfer of `cs.cmu.edu` (Type `HINFO`)

```
...
SAHARA.CMCL       DEC-600-5/333 UNIX
...
AMEFS.SRV         INTEL-486 UNIX
...
```

Note: no `HINFO` for `POSTOFFICE` or `LB`

Mapping domain names to IP addrs

Used by `gethostbyname()` and `nslookup`



R = `sahara.cmcl.cs.cmu.edu`

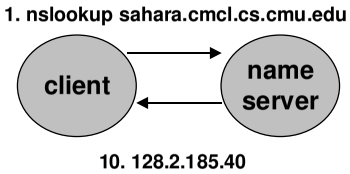
DNS Caching

Servers cache (keep a copy of) of information they receive from other servers as part of the name resolution process.

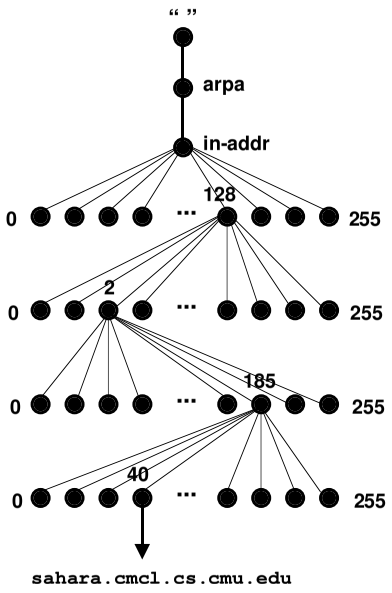
This greatly reduces the number of queries.

Example

- In our previous example, the next query for `sahara.cmcl` can be answered immediately because the server kept a copy of the address.



Mapping IP addrs to domain names



A separate hierarchy exists in the `in-addr.arpa` domain that maps IP addresses to domain names.

Used by `gethostbyaddr()` and `nslookup`

Example:

- **IP address**: 128.2.185.40
- Corresponding domain name `sahara.cmcl.cs.cmu.edu` stored at `40.185.2.128.in-addr.arpa`