15-213

Internet Services II

April 27, 2000

Topics

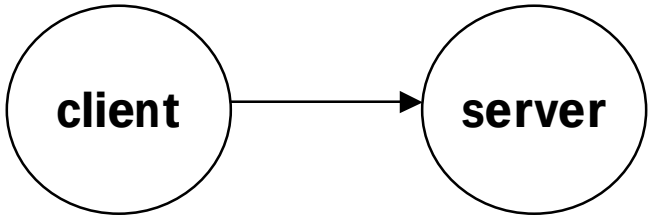
- dynamic content
- Tiny Web server tour

Serving dynamic content

Client sends request to server.

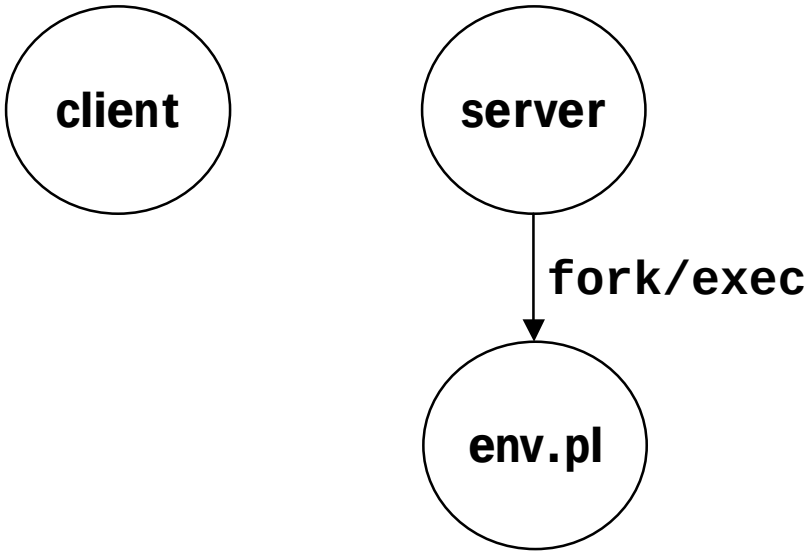
If request URI contains the string "/cgi-bin", then the server assumes that the request is for dynamic content.

```
GET /cgi-bin/env.pl HTTP/1.1
```



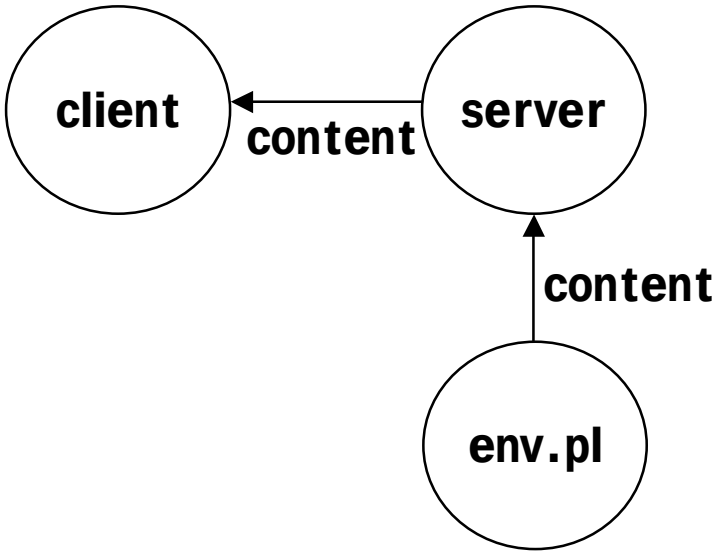
Serving dynamic content

The server creates a child process and runs the program identified by the URI in that process



Serving dynamic content

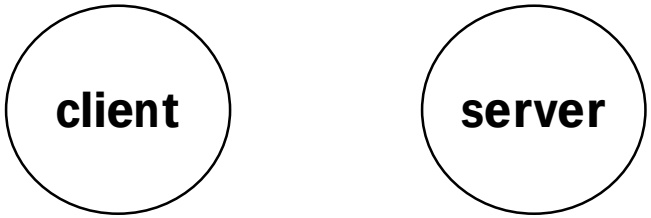
- The child runs and generates the dynamic content.
- The server captures the content of the child and forwards it without modification to the client



Serving dynamic content

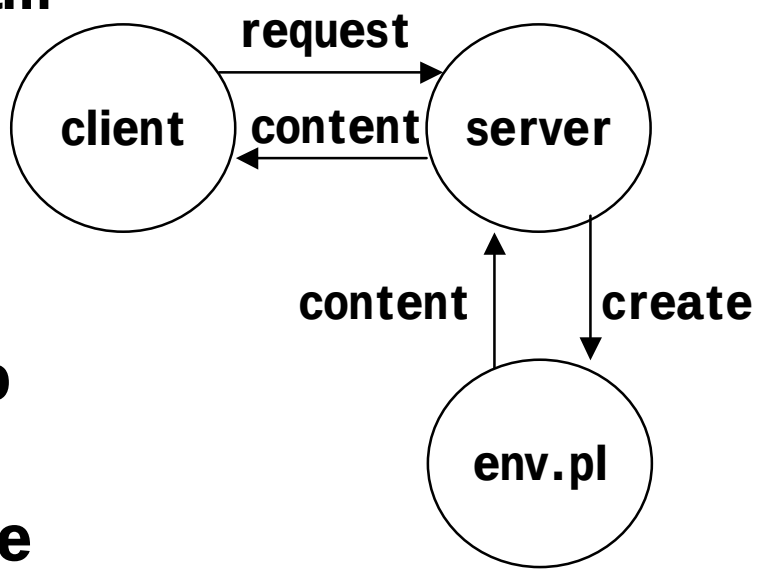
The child terminates.

Server waits for the next client request.



Issues in serving dynamic content

- How does the client pass program arguments to the server?
- How does the server pass these arguments to the child?
- How does the server pass other info relevant to the request to the child?
- How does the server capture the content produced by the child?
- These issues are addressed by the Common Gateway Interface (CGI) specification.



CGI

Because the children are written according to the CGI spec, they are often called CGI programs.

Because many CGI programs are written in Perl, they are often called CGI scripts.

However, CGI really defines a simple standard for transferring information between the client (browser), the server, and the child process.

add.com: THE Internet addition portal!

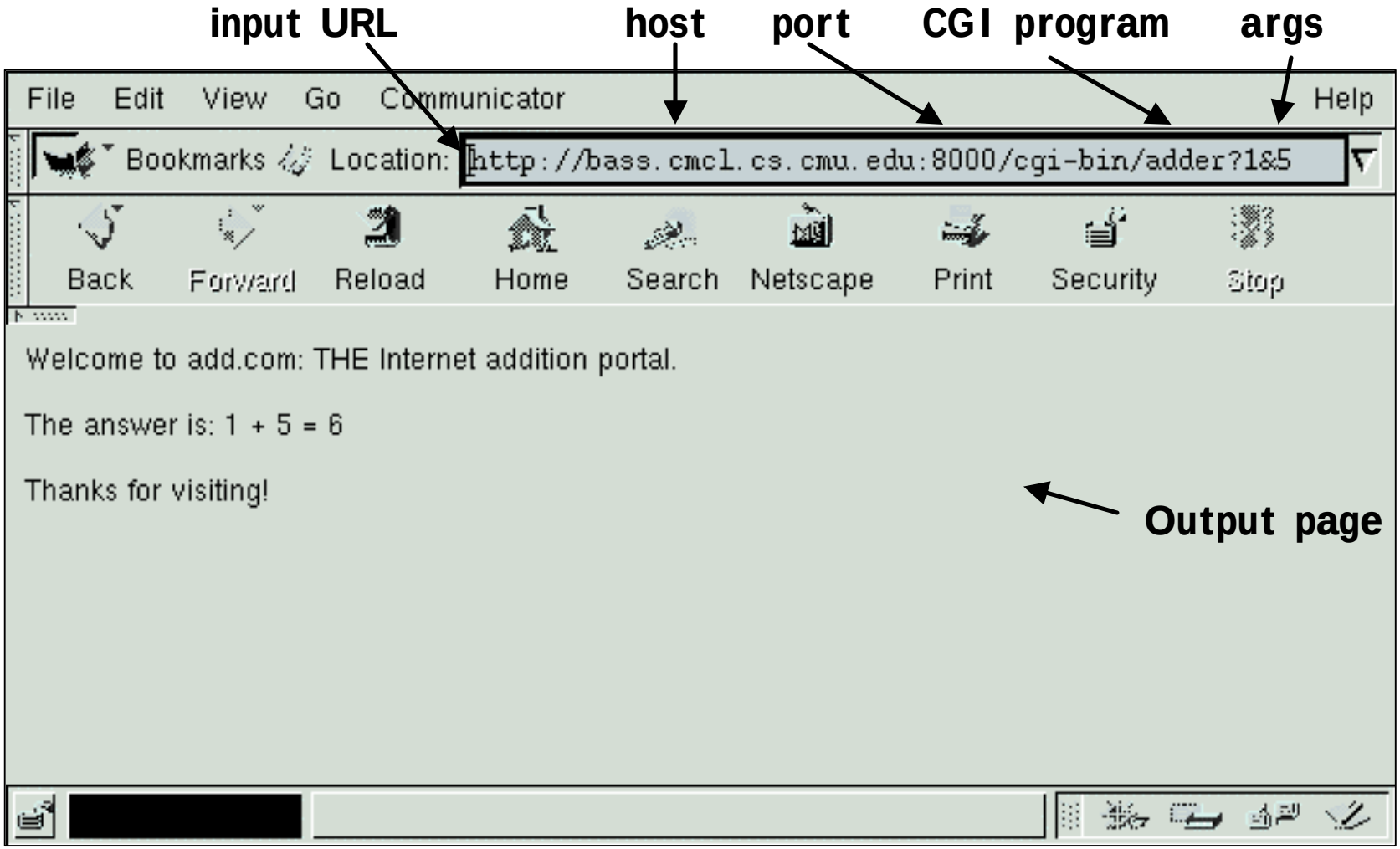
Ever need to add two numbers together and you just can't find your calculator?

Try Dr. Dave's addition service at add.com: THE Internet addition portal!

- Takes as input the two numbers you want to add together.
- Returns their sum in a tasteful personalized message.

After the IPO we'll expand to multiplication!

The add.com experience



Serving dynamic content with GET

Question: How does the client pass arguments to the server?

Answer: The arguments are appended to the URI

Can be encoded directly in a URL typed to a browser or a URL in an HTML link

- `http://add.com/cgi-bin/adder?1&2`
- `adder` is the CGI program on the server that will do the addition.
- argument list starts with "?"
- arguments separated by "&"
- spaces represented by "+" or "%20"

Can also be generated by an HTML form

```
<form method=get action="http://add.com/cgi-bin/postadder">
```

Serving dynamic content with GET

URL:

- **http://add.com/cgi-bin/adder?1&2**

Result displayed on browser:

Welcome to add.com: THE Internet addition portal.

The answer is: 1 + 2 = 3

Thanks for visiting! Tell your friends.

Serving dynamic content with GET

Question: How does the server pass these arguments to the child?

Answer: In environment variable QUERY_STRING

- a single string containing everything after the "?"
- for add.com: QUERY_STRING = "1&2"

```
/* child code that accesses the argument list */
if ((buf = getenv("QUERY_STRING")) == NULL) {
  exit(1);
}

/* extract arg1 and arg2 from buf and convert */
...
n1 = atoi(arg1);
n2 = atoi(arg2);
```

Serving dynamic content with GET

Question: How does the server pass other info relevant to the request to the child?

Answer: in a collection of environment variables defined by the CGI spec.

Some CGI environment variables

General

- SERVER_SOFTWARE
- SERVER_NAME
- GATEWAY_INTERFACE (CGI version)

Request-specific

- SERVER_PORT
- REQUEST_METHOD (GET, POST, etc)
- QUERY_STRING (contains GET args)
- REMOTE_HOST (domain name of client)
- REMOTE_ADDR (IP address of client)
- CONTENT_TYPE (for POST, type of data in message body, e.g., text/html)
- CONTENT_LENGTH (length in bytes)

Some CGI environment variables

In addition, the value of each header of type *type* received from the client is placed in environment variable `HTTP_`*type*

- **Examples:**
  - **`HTTP_ACCEPT`**
  - **`HTTP_HOST`**
  - **`HTTP_USER_AGENT` (any "`-`" is changed to "`_`")**

Serving dynamic content with GET

Question: How does the server capture the content produced by the child?

Answer: The child writes its headers and content to stdout.

- Server maps socket descriptor to stdout (more on this later).
- Notice that only the child knows the type and size of the content. Thus the child (not the server) must generate the corresponding headers.

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

/* child generates the headers and dynamic content */
printf("Content-length: %d\n", strlen(content));
printf("Content-type: text/html\n");
printf("\r\n");
printf("%s", content);
```

Serving dynamic content with GET

```
bass> tiny 8000
GET /cgi-bin/adder?1&2 HTTP/1.1
Host: bass.cmcl.cs.cmu.edu:8000
<CRLF>
```

HTTP request received by server

```
kittyhawk> telnet bass 8000
Trying 128.2.222.85...
Connected to BASS.CMCL.CS.CMU.EDU.
Escape character is '^]'.
GET /cgi-bin/adder?1&2 HTTP/1.1
Host: bass.cmcl.cs.cmu.edu:8000
<CRLF>
```

HTTP request sent by client

```
HTTP/1.1 200 OK
Server: Tiny Web Server
```

HTTP response generated by the server

```
Content-length: 102
Content-type: text/html
<CRLF>
```

HTTP response generated by the CGI program

```
Welcome to add.com: THE Internet addition portal.
<p>The answer is: 1 + 2 = 3
<p>Thanks for visiting!
```

```
Connection closed by foreign host.
kittyhawk>
```

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.**
- **support 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)**

Tiny: cerror

cerror() returns HTML error messages to the client.

- stream is the childfd 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 FTP 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: read HTTP request

```
    /* open the child socket descriptor as a stream */
    if ((stream = fdopen(childfd, "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);
      close(childfd);
      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);
      close(childfd);
      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.

```
    ...

    /* print arbitrary sized response body */
    fd = open(filename, O_RDONLY);
    p = mmap(0, sbuf.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    fwrite(p, 1, sbuf.st_size, stream);
    munmap(p, sbuf.st_size);
  }
```

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);
      close(childfd);
      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(childfd, buf, strlen(buf));
sprintf(buf, "Server: Tiny Web Server\n");
write(childfd, buf, strlen(buf));

...
```

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(childfd, 1); /* map socket to stdout */
      dup2(childfd, 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.

Tiny sources

The complete sources for the Tiny server are available from the course Web page.

- **follow the "Lectures" link.**