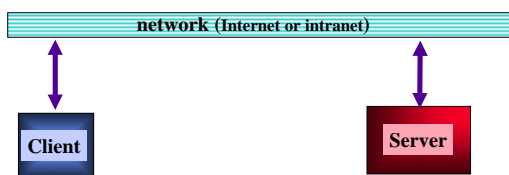


Technical basics

Client-server using TCP/IP
HTTPD
CGI
HTTP protocol

Clients and servers



Web browser or other
client program run on
user's machine

Program running on host
machine

Client-server basics

- o Brief overview, revision for some students; see also
 - CSCI213 java.net examples
 - More detailed information, and C++ code for networking, if you take CSCI214 or its replacement CSCI319
 - o CSCI319 emphasises a theoretical rather than a practical approach, so you may not get any experience with coding

TCP/IP library code

- o Communications between client and server use TCP/IP.
- o All details of network usage handled by code libraries linked to client and server applications.
- o Application programmer sees relatively simple interface
 - Maybe three functions used to get IP addresses (via DNS)
 - About three functions to set up server side
 - Two more functions for client to set up and connect to server
 - Ordinary read and write operations
- o Linked library code sorts out details of packets and error handling etc.

IP layer

- o Rules for transferring single packets between machines (identified by IP addresses) on network
 - Never concerned with this
 - It is all built into operating systems of client and server machines, and into the code of the intermediary switches and routers

TCP layer

- TCP
 - Hides the network and conceals "packet" transfer aspects
 - Makes it look as if client and server are connected by a bidirectional channel
 - Channel established when client requests TCP/IP connection
 - Channel stays open
 - Either end can write at any time (subject to buffering limits)
 - Blocking read requests
 - Channel closed on disconnect

Application layer

- Application defined protocol
 - Set of requests that a client can make
 - Typically
 - Request identifier
 - Argument data
 - Set of responses permissible for each request
- Application traffic often uses ordinary text messages

Servers

- FTP
 - Get file, put file services
- Telnet
 - Open a terminal connection to a program (usually program is "shell" – enter commands that run on remote host)
- HTTP
 - Fetch hypertext document (and more)
- Applications ...

Passive server Active client

- Server
 - Starts up when host OS restarted
 - Prepares "well known port"
 - **Waits for client(s)**
 - Responds to clients when they connect
- Client
 - Starts on user command
 - Establishes connection to server
 - **Transmits command** entered by user through to server program
 - Displays server response

Typical client structure

Pseudo-code

Get identity of server (IP address, port number)

IP address could be dotted decimal (easy to convert to IP number) or hostname (have to invoke DNS)

Build a "socket" structure containing server info

Open connection via socket

Get input/output streams from socket, wrap in classes with more useful I/O functions

Loop

user enters command

write to socket

read response

display response to user

*ftp and telnet clients have this structure;
http client (web browser) has similar code
invoked whenever a link must be followed*

Server

- Server side is more complex
- Port
 - Used when clients need to first establish connection
 - Used to transfer data.
- Server program and operating system have to keep these different kinds of use separate.

“Server socket” part

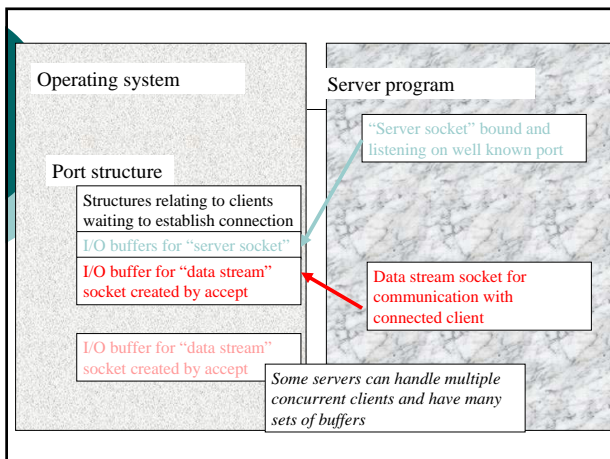
- Construct socket with description of server’s “well known port”
- Bind to port
- “Listen”
 - informs OS that server program is hoping that clients will come seeking connection to this port
- Accept
 - Blocking request to OS, causes server program to wait until a client does connect;
 - Returns a second socket connection (data stream socket) to same port, this has different I/O buffers

Server’s “data socket” part

Repeat

read next command from client via data socket;
process command
generate response
write response to data socket;

Until client disconnects;



Core of server

- Pseudo-code
 - Construct socket for well known port
 - Bind to port
 - Listen
 - Forever {
 - newdatasocket = accept
 - ...
 - }

Multiple clients

- Servers generally have to deal with multiple clients
 - Many people using telnet for remote access
 - Several people using ftp to download files
 - Numerous clients accessing HTTP server for web browsing
- Number of strategies for dealing with multiple clients

Server strategies for multiple clients

Serial server

- Don’t allow multiple clients; make them wait. Finish dealing with first client before take next client from head of a short queue maintained by OS.

Forking server (“fork” – Unix speak for start a new process)

- Create new process to handle client (costly in computing resources but easy to code)

Threaded server

- Have separate thread for each client

Polling or “select” loop server

- Hardest to implement, juggle work for all clients!

Forking server

server (listener)

create socket
bind to "well known" port
listen (i.e. activate service)
forever()
accept client
fork subprocess to deal with client

Different processes!

server (child process)

forever()
read next request
if "bye" then quit
handle request
send response

*new socket created by
accept is passed to child*

Threaded server

All the one process (independent clients)

main thread
create socket
bind to port
activate
forever()
accept client
create new
thread to
deal with this
client
start thread in
"handle_client"

handle_client
forever()
read next request
if "bye" then break
handle request
send response
thread die

handle_client
forever()
read next request
if "bye" then break
handle request
send response
thread die

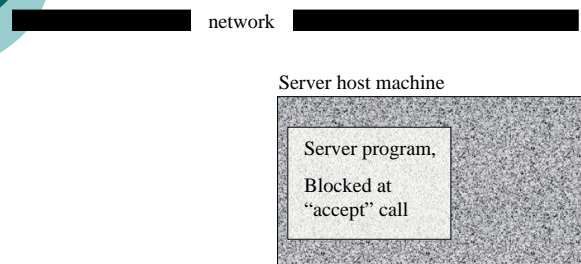
handle_client
forever()
read next request
if "bye" then break
handle request
send response
thread die

handle_client
forever()
read next request
if "bye" then break
handle request
send response
thread die

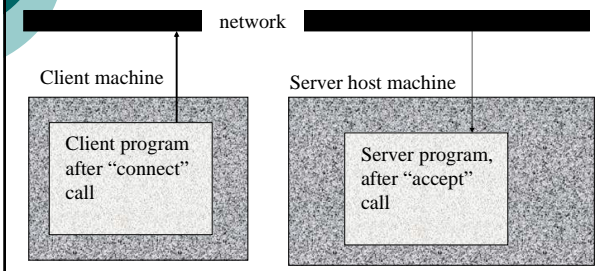
Internet services

- Traditionally, "forking server" architecture has been used
 - Very easy to implement on Unix and similar operating systems
 - Thread packages weren't standardized until mid-1990s; most Internet services defined and running much earlier.
- Gradual shift to threaded servers
 - Apache 1.3.39 most common Web server is sophisticated version of forking server
 - Apache 2, new version coming into use, is a threaded server

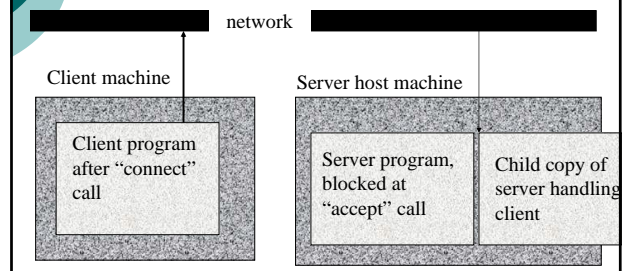
Traditional forking server operation : 1: server startup



Traditional forking server operation : 2: client connect



Traditional forking server operation : 3: after fork



Forking server code

```
Build socket structure describing well known port
Bind to port
Listen
Forever {
  newdataSocket = accept(...);
  "fork"
  if (this is parent process) close connection to
    newdataSocket
  else { // child process
    close connection to main server socket;
    repeat { handle client request }
    until client disconnects;
    kill this process;
  }
}
```

Forking server

- After “fork” (Unix) or equivalent Windows call, have two processes running
 - Same code (shared code segment, or two copies of code depending on sophistication of OS)
 - Independent stack and heap segments
 - Initially some shared “system” data – file descriptors
 - Some mechanism for process to “know” whether it is parent or child

Forking server

- Often that is all you want
 - Web-server (will discuss this more in Apache lecture)
 - “Chief” process
 - Launched automatically
 - Waits for clients to connect
 - Launches children
 - “Child” process
 - Runs the rest of the web-server code on behalf of specific connected client
 - Gets files etc

Exec

- But sometimes want more
- Web-server to start form-processing program.
- Fork
 - Child doesn’t want to run copy of web-server code, it wants to run a form-processing program.
 - So
- Exec
 - Change the code segment, loading in code of desired program
 - When loading complete, invoke main() of program
 - (This work done by OS on behalf of process)

Fork:exec:load-script

- Often the program that is “exec-ed” is a script interpreter, e.g. the Perl interpreter.
- Need another step to load the script that is to be run
 - Script would be the Perl code of the form-processing program

Costs

- Fork-exec
 - Relatively costly calls to operating system
 - Both slow
 - Fork is slow
 - Have to allocate new stack and heap space and copy some information on stack
 - May have to copy all the code segment
 - Exec is slower
 - Always have to go to disk to get executable that is to be run
 - Then have to create new code, stack, and heap
 - May have to run complex initialization code
- Many of “hacks” encountered later in CSCI399 represent attempts to reduce these costs and speed up response

ftp server as example

- ftp has been around since 1973 if not earlier
- Reworked version typical of main Internet TCP/IP based servers from ~1982
 - Connect
 - Login (*either user account in which case get direct access to that user's filespace and bidirectional transfer of files, or "anonymous" and get access to limited set of files for free download*)
 - Commands
 - Change directory, list directory, get file, put file, set file transfer mode (text/binary) ...
 - Main TCP/IP connection kept open until log out
 - Secondary TCP/IP connection used for actual file transfers
 - Server maintains state information (login id, current directory, current transfer mode, ...)

ftp – application protocol

- Application protocol
 - Commands that client can send server
 - Responses that server can send client
- Client commands
 - Simple text lines such as "cd ../Examples", "binary", "get image1.gif"
- Server responses
 - One line status reports (numeric code, text explanation)
 - Info needed to establish separate TCP/IP connection for a file transfer, or text data like listing of a directory

HTTPD

The web-server process

Overview Server Processes

Server machine



**"http" daemon
server listening to
port 80**

Server machine



**"http" daemon
server listening to
port 80**

Incoming connect
request

Server machine



**"http" daemon
server listening to
port 80**

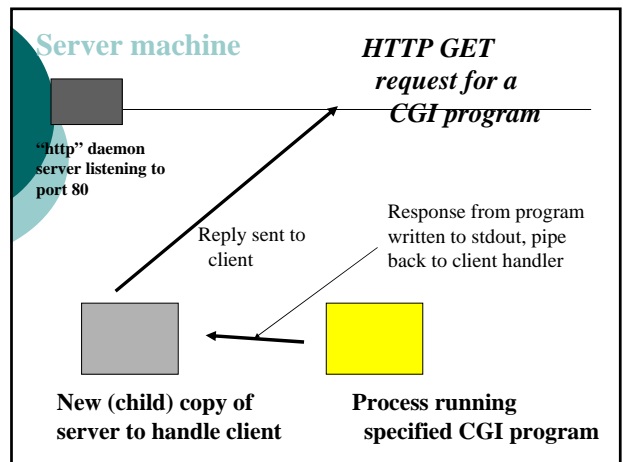
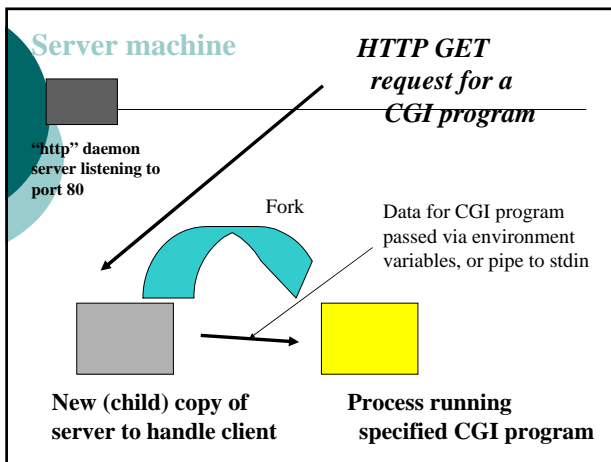
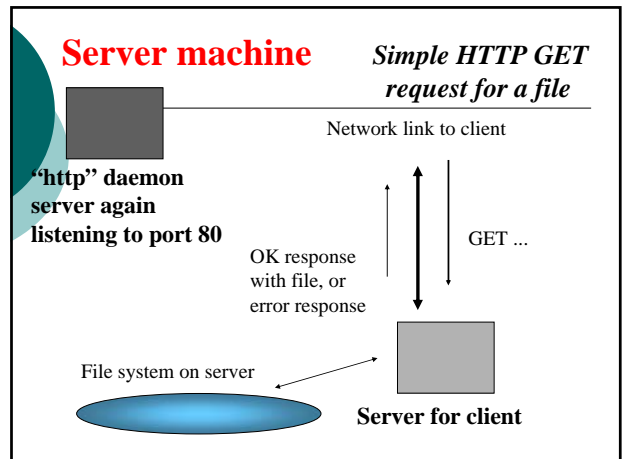
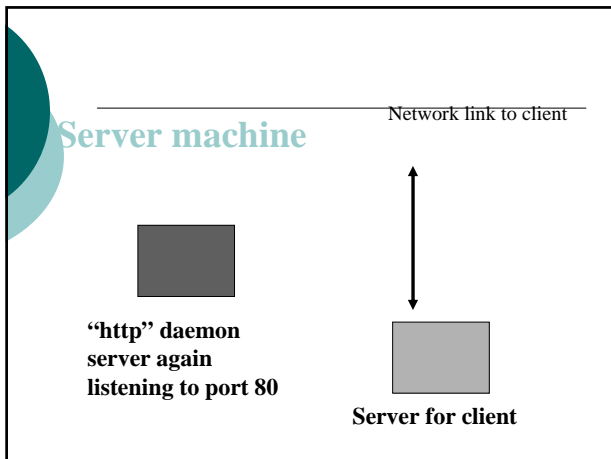
Incoming connect
request



Fork



**New (child) copy of
server to handle client**



Server processes

Each client connection creates at least one process, and may create many more on server--

- example: fetch page with 6 embedded gifs, a trivial applet, an embedded audio track:
 - connect, fork, interact with own server, get HTML page
 - repeat entire connect, fork, interact process
 - once for each gif
 - once for applet (actually, once for each class defined for applet)
 - once for audio track
 - (that is at least 9 processes so far)
- (that is a bad case scenario – HTTP 1.0 etc – since 1999 it has been more usual for server to “keep alive” a connection allowing more than one file to be downloaded)

Server processes

- If request is not a simple request for a data file, but is instead a request for server to run a program:
 - additional fork & exec to start CGI program
 - cost of setting up environment variables etc
 - pipe to get output back from CGI program

Server processes

- Web and HTTP protocol are costly for server machines.
- **HTTP1.1 protocol** (1999 standard) allows same connection to be used for more than one file
 - connect
 - get data file
 - repeat up to n - times (n specified by control file on server)

Other server features

- Server like Apache, even if configured simply to return static files, is quite elaborate
 - Can impose some security constraints
 - Certain files available only to authorized clients
 - Can deal with requests for alternate language versions of pages

Server processes

- Servers can have "plugins"
 - these plugins add functionality
 - Functionality to handle CGI request built into server itself
 - save cost of that second fork-exec in the call sequence, since it will be compiled code it will run faster than an interpretive script

Server processes

- Like a plug-in, a server side extension has to be written to match 'host' program
 - component must provide certain functions that will be called by server
 - component relies on specific functions in server
- Servers define such functionality through an "Application Programmer Interface" (hence NSAPI and ISAPI)
Apache has rules for "module writers"

Server processes

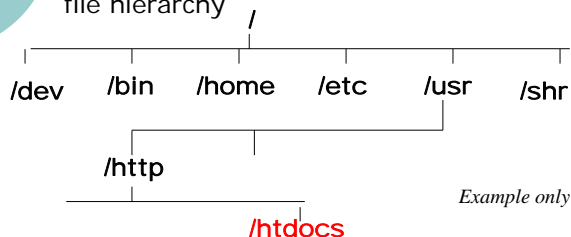
- Generally, a server that receives a request for a HTML file can simply fetch it from disk and send it back.
- There is however an option for "**server side includes**" (SSI)
 - requires server to check the text of each HTML file before it is returned
 - can have requests in the HTML for the server to insert extra information (e.g. time file was last modified)

Server processes

- Too costly to check every HTML file; systems that support *server side includes* use various hacks to tag those files that must be processed in this way.
- SSI
 - More on this later;
 - SSI itself is limited, but it inspired technologies like Microsoft's "Active Server Pages" and PHP

Servers and file access

- HTTP server should be set up so that it works within one restricted subtree of file hierarchy



Servers and file access

- If set up this way, Web site administrator has (should have) some control
 - only directories below /usr/http/htdocs accessible
 - Web administrator owns directory and is responsible for putting files there
- But
 - watch out for directory navigation
 - users on site will probably demand to control their own pages

Navigating the file hierarchy

- If you type in the URL for a directory rather than a file (get a real URL, leave off filename), you may get:
 - the home.html or index.html page in that directory
 - a selected (edited) list of directory contents
 - **a simple listing of the directory**

Navigating the file hierarchy

- Inquisitive visitors can use Unix style “../” directory escapes to ascend hierarchy
- So if Webmaster not careful:
 - GET ../../../../home/staff
 - and someone has now a list of user-ids of your staff (getting user-ids is a starting point for some break in strategies)
- Webmaster must arrange that web server runs with a “chroot” setting so that htdocs seems to be root of file system (“/”).

Web server: whose process?

- Every process running (on Unix) has a “user id” associated - the person who is “running the program”.
- This “user id” determines file access privileges - execute, read, write.
- Want Web server to be run for a token user who has very few privileges
 - (typically, something like “nobody” in group “outsider”)

User directories

- From security perspective, better if Webmaster controls all web pages, but users do like editing their own pages.
- Servers can be configured so that URLs of form:
/servername/~username/file.html
- will be looked for in subdirectory (public_html) of user’s home directory (rather than within /htdocs hierarchy)

Controls on IP address of clients

- Web sites can be configured so that have greater variety of access
- Configuration file for server, coupled with control files in appropriate directories, can be used to specify restrictions.
- Common one is to restrict certain subdirectories so that can only be fetched by clients with same IP net address as server

Logging

- Optionally, a Web server will log every client request
 - log record identifies client machine (as IP number, but you can request DNS lookup to get names)
 - HTTP protocol allows for attempt at identifying actual user on client machine (but most machines don't support protocol for this, it is expensive, it won't work if there is a "firewall" at either end, and won't work if dynamically allocated IP address)
 - other data are time, resource requested etc

Access log

- 130.130.189.103 - - [28/May/2001:14:37:17 +1000] "GET /-yz13/links.htm HTTP/1.0" 200 1011
- 208.219.77.29 - - [28/May/2001:14:37:26 +1000] "GET /robots.txt HTTP/1.1" 404 216
- 130.130.189.103 - - [28/May/2001:14:38:18 +1000] "GET /-yz13/image/tb.gif HTTP/1.0" 200 94496
- 130.130.64.188 - yag [25/May/2001:11:39:49 +1000] "GET /controlled/printenv.pl HTTP/1.1" 401 486

CGI programming

CGI programming

- "Common Gateway Interface"
- Essentially a simple protocol (set of rules) that define how Web servers are to pass data to sub-launched data processing programs.

CGI programming

- Basic idea:
 - user fills in form in an HTML page
 - form has a number of named input fields (radio buttons, checkboxes, input text, textarea, select)
 - form processing code in browser builds a request as a set of **name=value** pairs (one for each input field in form)
 - form has URL (**web server name**, path to **data processing program**), request sent to specified **web server**
 - ...

CGI programming

- Basic idea (continued):
 - ...
 - request sent to specified **web server**
 - web server starts new process which is made ready to run specified **data processing program**
 - **name=value** data pairs passed in some way to data processing program
 - data processing program works out information required in reply
 - ...

CGI programming

- Basic idea (continued):
 - ...
 - Data processing program works out information required in reply
 - reply information made up into a HTML page (just copy standard HTML layout code so have `<head> ... </head> <body> ...</body>` framing)
 - processing program passes generated page to web server (on same computer)
 - web server returns page to client browser

CGI programming

- Processing program can be written in any language that you want
 - simply have an "executable" file in the **cgi-bin** directory used by your web server
 - executable file?
 - Compiled and linked code from C or C++.
 - A "script" file for a scripting language
 - sh, Unix's standard shell script
 - perl, a scripting language very popular for these applications

Text files from editor, with executable bit set by chmod

CGI programming

- "x-www-form-urlencoded"
 - dealing with encoding of strings
- http request
- "environment variables"
- GET
- POST
- replying with HTML & HTTP

CGI programming "x-www...encoded"

- HTTP protocol has restrictions on characters that can appear in things like URLs
- If a browser has a string that contains an unacceptable character, it is supposed to substitute an acceptable combination
 - letters and digits are OK
 - spaces get changed to '+' signs
 - everything else ...

CGI programming "x-www...encoded"

- Non alphanumerics translated -
 - % sign, two hex digits (01234..9AB..E)
- Encoding is applied to all text entered in any input area of a form.
- Your CGI program will have to perform reverse translation for all "values" that it processes.

HTTP request

- Text message (you can telnet to a web server and type in requests)
- Message type
 - GET/PUT/HEAD *resource protocol-level*
- Some parameters, name: value [,value]
 - things like make of browser, (MIME) data types that you can accept, preferred language, size of data portion of message
- Blank separator line
- Data as appropriate

Environment variables

- General feature of Unix system.
- Name=Value pairs**
- Used to define preferences -
 - what editor to sub-launch from mail reader
 - which printer
 - which font for xterm sessions
 - where to look for programs (other than current directory and /bin and /usr/bin)
 - ...

Environment variables

- Environment variables set
 - in your .profile file in home directory
 - by explicit shell command
- Values of environment variables can be changed, and new ones added as part of the "exec()" call that is used to start a different program.

Environment variables

- Unix system calls allow an executing program to find values for named environment variables
 - char* getenv(char* env_var_name)
- CGI relies on environment variables as part of its mechanism for the web server to pass data to the data processing program.

Can also get at a char [] with all the environment variables, loop through until get a null pointer.*

Environment variables Web-server-> CGI program

- Three types of data transferred using environment variables
 - details of server
 - details sent as part of the HTTP header that is sent by the client's web browser
 - some or all (depends on style of request) of the data from the form filled in by the user

Environment variables Web-server-> CGI program

- Server info
 - SERVER_SOFTWARE,
 - SERVER_PORT
 - ...

```
char *info = getenv("SERVER_SOFTWARE");
cout << "Server is " << info << endl;
info = getenv("SERVER_PORT");
int portnum = atoi(info);
if(portnum != 80) cout << "Using non standard port " <<
portnum << endl;
```

Environment variables Web-server-> CGI program

- General details about client
 - client and client's machine:
 - REMOTE_HOST hostname from DNS or null
 - REMOTE_ADDR its IP address
 - REMOTE_USER usually undefined, could be user id
 - USER_AGENT web browser used
 - REMOTE_ADDR its IP address
 - HTTP_ACCEPT MIME types client handles

Environment variables Web-server-> CGI program

- Details of request
 - REQUEST_METHOD get or post
 - CONTENT_TYPE should be x-www-form-urlencoded
 - PATH_TRANSLATED full pathname of CGI program

- PATH_INFO
- QUERY_STRING
- CONTENT_LENGTH ***Depend on POST/GET method being used***

Form requests for GET & POST

- On web client side, browser program builds request with name=value pairs
 - values are x-www-...urlencoded
 - Pizza example
 - (size, regular) (delivery, express), (customer, dennis smith), (address, 22 fairoaks avenue/nclayton)
- ```
size=regular&delivery=express&customer=dennis+smith&address=22+fairoaks+avenue%0dclayton
```

## GET

- If HTTP Get protocol being used, the generated query string is appended to the url  
(/cgi-bin/fred\_cook )
- ```
GET /cgi-bin/fred_cook?size=regular&delivery=express&customer=dennis+smith&address=22+fairoaks+avenue%0dclayton HTTP/1.0
```
- Web server splits URL at '?', places everything in QUERY_STRING environment variable

POST

- If HTTP POST protocol being used, the encoded string will appear in data area of HTTP message

```
POST /cgi-bin/fred_cook HTTP/1.0
User-agent: Mozilla 2.0
Accept: test/plain, text/html
...
Content-type: x-www-...urlencoded

size=regular&delivery=express&customer=dennis+smith&address=22+fairoaks+avenue%0dclayton
```

CGI program

- GET
 - program gets a large string by asking for QUERY_STRING environment variable
- POST
 - program reads a long string character by character from stdin (can't rely on end-of-file as reading from a pipe connected to web server, so first use value of CONTENT_LENGTH environment variable to control reading loop)

CGI program

- Program wants the name=value combinations
- Strategy
 - use & to break input into sections
 - change + signs back to spaces
 - reverse encoding of non alphanumerics
 - split at = sign
- This should give a set of name value string pairs (use atoi() or similar function to convert numeric strings to values)

CGI program

- Utility functions exist for C (and for perl, and ...) that help perform the steps
 - plusospace() (substitute for + signs)
 - unescape_url() (%hex escape sequence converted to character)

CGI program

Typically CGI program creates an HTML page as response

- can create plain text, gif files etc
- CGI program outputs to stdout
 - last line of HTTP header (the one defining content type, Content-type: text/html or image/gif)
 - blank separator line
 - content (encoded as appropriate for MIME type)
- Web server adds rest of HTTP stuff at front.

Replying with HTML

- Page made up by just outputting standard HTML text

```
cout << "<html>" << endl << "<head>" << endl;
cout << "<title>Your pizza order from Fred's"
    << "</title>" << endl << "</head>" << endl;
cout << "<body>" << endl;
cout << "Order for " ;
...
cout << "</body>" << endl << "</html>" << endl;
```

HTTP

HyperText Transfer Protocol

http: from the source

- *The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems.*
- *It is a generic, stateless, protocol.*
- *It can be used for many tasks beyond its use for hypertext, eg name servers and distributed object management systems.*

http: from the source

- A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

http versions

- HTTP/0.9
 - a simple protocol for raw data transfer across the Internet.
- HTTP/1.0
 - Added MIME-like features with information about data transferred and modifiers on request/response semantics.
- HTTP/1.0 limitations
 - does not sufficiently take into consideration the effects of hierarchical proxies, caching, the need for persistent connections, or virtual hosts.
 - Also, many incompletely-implemented versions of HTTP/1.0

http versions

- http 1.1 (~1999)
 - Better support for two communicating applications to determine each other's true capabilities.
 - Much more detailed specification of caching rules
 - Defined support for persistent connections.

http

- Conceptually
 - Client --- network --- Origin server
 - Request and response travel between two endpoints
- Really
 - Client --- network --- Intermediary --- net --- another intermediary --- etc --- Origin
 - Any of intermediaries may have a cached response to query and respond on behalf of origin server

http: definitions – request, response

- HTTP-message = Request | Response
- Both Request and Response:
 - start-line
 - *(message-header CRLF)
 - CRLF
 - [message-body]

http: Request

- Request = Request-Line
 - *((general-header
 - | request-header
 - | entity-header) CRLF)
 - CRLF
 - [message-body]
- (A request is a request line followed by some headers, each on a separate line, followed by a blank line, and an optional message body)

http: request line

- Request-Line = Method SP Request-URI SP HTTP-Version CRLF
- Request-URI = "*" | absoluteURI | abs_path | authority
 - "*" request is about server
 - Absolute URI (protocol, host, path etc)

http: request line

- Currently absoluteURI used when client browser communicates via a proxy
- If abspath used, then client should include a "host" header.

```
GET http://www.acme.com/home.htm HTTP/1.1
Or
GET /home.htm HTTP/1.1
HOST: www.acme.com
```

One host machine,
with one IP address
192.193.194.195

Registry of names IP
addresses:
www.WonderDiet.com
= 192.193.194.195
www.HealthFoods.com
= 192.193.194.195
www.TheGym.com
= 192.193.194.195
...

When get request for "index.html" need
to determine whether Gym, HealthFoods, ...

http: Request types:

- Request-Line = Method SP Request-URI SP HTTP-Version CRLF
- Methods:
 - "OPTIONS"
 - "GET"
 - "HEAD"
 - "POST"
 - "PUT"
 - "DELETE"
 - "TRACE"
 - "CONNECT"

http: requests

- GET
 - Request for resource
 - Other header info. can modify request
 - Conditional get (if info. contains something like IF_MODIFIED_SINCE)
 - Partial get (if info. contains Range specification)
- HEAD
 - Request for information about resource
 - Server replies with same header data describing resource as in GET, but message not appended

http: requests

- POST and PUT
 - Both specify a URI
 - Both supply content
 - The URI in a POST request identifies the resource that will handle the enclosed entity (e.g. a data-accepting process, a gateway to some other protocol, or a separate entity that accepts annotations).
 - The URI in a PUT request identifies the entity enclosed with the request

http: requests

- POST
 - Submit an entity (content part of message) as a new subordinate of the resource identified by the Request-URI in the Request-Line.
 - Annotation of existing resources;
 - Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
 - Providing a block of data, such as the result of submitting a form, to a data-handling process;
 - Extending a database through an append operation.

Subordinate to that URI: e.g. a file to a directory, a news article to a newsgroup, or a record to a database.

http: requests (POST)

- Function performed by the POST method is determined by the server and is usually dependent on the Request-URI.
- If creates a new identifiable resource:
 - Respond 201 (Created) with a Location header for resource
- If doesn't create a new resource:
 - Respond 200 (OK) with entity that describes result
 - 204 (No content)

http: requests

- PUT
 - Requests that the enclosed entity be stored under the supplied Request-URI.
 - If the Request-URI refers to an existing resource, the enclosed entity taken as a modified version.
 - If the Request-URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI.

http: requests

- TRACE
- OPTIONS
- CONNECT

http: responses

- Response = Status-Line
*((general-header
| response-header
| entity-header) CRLF)
CRLF
[message-body]
- Response is a status line, followed by some number of headers, a blank line, and an optional message body.
- Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

http: response status codes

- Five groups of status codes
 - 1xx request received, processing continues
 - 2xx "success" (resource retrieved, put, posted, whatever)
 - 3xx Further action needed
 - 4xx Client error, request invalid
 - 5xx Server failure

http: response status codes

- Status-Code =
 - "100" ; Continue
 - "101" ; Switching Protocols
 - (Client can submit partial request then ask if it should continue, client can ask to switch protocols)

http: response status codes

- Status-Code =
 - "200" ; OK
 - "201" ; Created
 - "202" ; Accepted
 - "203" ; Non-Authoritative Information
 - "204" ; No Content
 - "205" ; Reset Content
 - "206" ; Partial Content

http: response status codes

- Status-Code =
 - "300" ; Multiple Choices
 - "301" ; Moved Permanently
 - "302" ; Found
 - "303" ; See Other
 - "304" ; Not Modified
 - "305" ; Use Proxy
 - "307" ; Temporary Redirect

http: response status codes

- Status-Code =
 - "400" ; Bad Request
 - "401" ; Unauthorized
 - "402" ; Payment Required
 - "403" ; Forbidden
 - "404" ; Not Found
 - "405" ; Method Not Allowed
 - "406" ; Not Acceptable
 - "407" ; Proxy Authentication Required
 - "408" ; Request Time-out
 - "409" ; Conflict
 - "410" ; Gone
 - "411" ; Length Required
 - "412" ; Precondition Failed
 - "413" ; Request Entity Too Large
 - "414" ; Request-URI Too Large
 - "415" ; Unsupported Media Type
 - "416" ; Requested range not satisfiable
 - "417" ; Expectation Failed

http: response status codes

- Status-Code =
 - "500" ; Internal Server Error
 - "501" ; Not Implemented
 - "502" ; Bad Gateway
 - "503" ; Service Unavailable
 - "504" ; Gateway Time-out
 - "505" ; HTTP Version not supported

http: Message headers

- HTTP header fields include
 - general-header,
 - request-header,
 - response-header,
 - entity-header.
- Each header field:
 - name : value

Field names are case insensitive

http: general header fields

- Used with both request and response messages.
 - Cache-Control
 - Connection
 - Date
 - Pragma
 - Trailer
 - Transfer-Encoding
 - Upgrade
 - Via
 - Warning

Explain a few of these to illustrate capabilities of http

http: general controls: cache

- Allow requestor or responder to ask to override default caching rules in proxies on connection route.
- Cache-Control: directive
- Requestor can specify:
 - No-cache, no-store, max-age=..., max-stale=...,
 - No-cache (can't use response to request without confirming no changes),
 - no-store (don't let this entry get stored permanently on tape),
 - max-age, will accept cached entry if age < ...

http: cache controls

- Responder can specify:
 - Public, private, no-cache, no-store, no-transform, must-revalidate, proxy-revalidate, max-age=... etc
 - Public cache: this response can be cached and used to satisfy others requesting same resource
 - Private cache: can be cached, but only for this requestor
 - Max-age ...

http: general headers: date

- With a few limited exceptions, date (in standard format) must be included in responses:
- Date: Tue, 1 May 2001 09:15:30 GMT
- Dates can be included with "PUT" and "POST" requests from clients

http: general headers: update

- Client can ask to change protocol with an upgrade header eg secure http
- Upgrade: SHTTP/1.3
- Server includes Upgrade header if indicating that it is changing protocol

http: Transfer Encoding

- Can do things like apply gzip (or Compress) compression algorithm to body of message
- Transfer-Encoding heading specifies any transform applied.
- (Note, intermediaries in chain between client and server – such as proxy servers- can chose to change encoding on a message)

http: request headers

- request-header =
 - Accept ;
 - Accept-Charset ;
 - Accept-Encoding ;
 - Accept-Language ;
 - Authorization ;
 - Expect ;
 - From ;
 - Host ;
 - If-Match ;
 - If-Modified-Since ;
 - If-None-Match ;
 - If-Range ;
 - If-Unmodified-Since ;
 - Max-Forwards ;
 - Proxy-Authorization ;
 - Range ;
 - Referer ;
 - TE ;
 - User-Agent ;

http: request headers

- Client can specify things like
 - media types (*I prefer "wav" audio but will accept "au" audio*),
 - character sets (*ISO-ASCII, Unicode, ...*),
 - and languages (*I would like British-English but will accept American-English*)
- Specifications can be quite complex as can include "quality" indicators indicating how strong are your preferences!
Accept: audio/*; q=0.2, audio/basic

http: request headers

- **From**: email address of client's user
 - Required in requests from robots
 - Browsers etc shouldn't send this by default, should be explicitly selected user option
- **Host**: either absolute URI or include host header
- **User-agent**: your browser details
- **Referer**: if following a link from a page, then that page's URI

Request example

```
GET http://www.salesforce.com/SpecialOffer.html HTTP/1.1
Connection: Keep-Alive
Accept: image/gif, image/png, text/html, text/plain
Accept-Charset: iso-8859-1
Accept-Encoding: gzip
Accept-Language: en-GB, en
User-Agent: Mozilla/4.7 ...
```

http: response headers

- response-header =
 - Accept-Ranges ;
 - Age ;
 - ETag ;
 - Location ;
 - Proxy-Authenticate ;
 - Retry-After ;
 - Server ;
 - Vary ;
 - WWW-Authenticate ;

http: response headers

- Accept-Ranges
 - If client sends an options request asking to use ranges, server will reply with message that has Accept-Ranges tag specifying its capabilities
- Age
 - If data being returned by a cache, Age header is estimate of how many seconds ago the original came from origin server

http: response headers

- Location
 - In a redirect response
 - If a resource created
- Retry-After
 - In a "service not available" response, tells you an estimate of when it will be available
- Server
 - Identifier of web server product

Response example

```
HTTP/1.1 206 Partial content
Date: Wed, 15 Nov 1995 06:25:24 GMT
Last-Modified: Wed, 15 Nov 1995
 04:58:08 GMT
Content-Range: bytes 21010-47021/47022
Content-Length: 26012
Content-Type: image/gif
```

Authorization

- Server side controls apply to directories.
- Can have controls
 - IP based,
 - domain-name based,
 - User/password
 - Plus a few exotics ..
- Combinations
 - *E.G. can be read by those in domain ... or by any of the following people if they give their passwords ..., ..., ...*

In some cases, control may apply to a URI or to a file

Authorization: *caution*

- Controls are of limited efficacy.
- You aren't really protecting data, you are marking some data *"private, please don't read this unless you are supposed to"*
- Relatively easy for unauthorized persons to break in and read the data.
- Better now – properly configured web-server should stop access to controlled areas.

IP/Domain control

- Checks IP address or domain name
- Response:
 - 200 plus contents
 - 403 Forbidden

Forbidden: *canned text response*

```
<HTML><HEAD>
<TITLE>403 Forbidden</TITLE>
</HEAD><BODY>
<H1>Forbidden</H1>
You don't have permission to access ...
on this server.<P>
<HR>
<ADDRESS>Apache/1.3.17 Server at ... 8080</ADDRESS>
</BODY></HTML>
```

Password control

- Directory password controlled.
- Request for a file in that directory produces a "401 Unauthorized" response.
 - Response includes details of how password control being applied and an identifier ('realm') for the controlled resource.
- TCP/IP link kept open by server, to allow browser to respond to challenge.

Password control

- Browser puts up dialog allowing entry of user-name and password.
- User-name and password returned in a revised GET request (*now with an **Authorization** header*)
 - Name/password sent with trivial encryption (base64 encoding) so insecure
- Browser validates name and password and returns page.

Password control

- Browser keeps *realm/name/password* triple
- Browser will respond automatically to other 401 Unauthorized challenges relating to same realm by submitting authorized requests.

Password control

- Authorization identifier is typically stored in the logs maintained by the web server.

Password control: *canned response*

```
HTTP/1.1 401 Authorization Required
Date: Thu, 24 May 2001 08:09:45 GMT
Server: Apache/1.3.17 (Unix)
WWW-Authenticate: Basic realm="Controlled space"
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1

1da
<!DOCTYPE HTML PUBLIC "-//IETF/DTD HTML 2.0//EN">
<HTML><HEAD><TITLE>401 Authorization Required</TITLE>
</HEAD><BODY>
<H1>Authorization Required</H1>
This server could not verify that you are authorized to access the document
requested. Either you supplied the wrong credentials (e.g., bad password), or
Your browser doesn't understand how to supply the credentials required.<P>
<HR><ADDRESS>Apache/1.3.17 Server at ....</ADDRESS>
</BODY></HTML>
```

Negotiated content: browser

- *Many browsers don't provide adequate interface so some aspects of negotiated content are hard to use.*
- Browser preferences:
 - IE: Tools/Internet/Languages
 - Netscape: Edit/Preferences/Navigator/Languages
- Can specify languages accepted and preferred order

Negotiated content: server

- For choice of document in different human language:
 - Server has table that maps language preference code (included in browser request) to special file extension:
 - en-us .yank
 - en-aus .oz
 - en-gb .lime
 - Server looks for variant of requested file with extension
 - Greetings.html.yank
 - Greetings.html.oz
 - ...

Negotiated content: server

- Other mechanisms use "variant" files associated with resources
 - These define the files that contain different versions, e.g. jpeg, gif, ... pictures
 - Can have supplementary data such as "quality" ratings.
- Server compares client preferences with data in var file and finds best match

http: GET and POST and forms

- Data from forms sent as (encoded) set of name=value pairs

```
GET /cgi-bin/process1.cgi?name=me&address=123+main+street HTTP1.1
```

```
POST /cgi-bin/process1.cgi HTTP1.1
```

```
...
```

```
Name=me&address=123+main+street
```

http: Get and Post

- Note:
 - Get request URI can be book-marked and then resubmitted
 - Caution: don't use Get for anything that changes records at origin server (as user can submit same request repeatedly)

http: problem of "state"

- Request: Response
 - TCP/IP connection established
 - Possibly server process created
 - One or more files copied
 - TCP/IP connection closed
 - Possibly server process terminated
 - Apart from entry in log files, the server has no memory of this exchange

http: state

- Next request from client appears completely separate
 - IP addresses can't be used to track clients unambiguously
 - All users of time-share system get same IP address
 - Proxy based system sending IP address of proxy
 - Your IP address may be dynamically reassigned between requests!

http: state

- So problems in situations like
 - Form 1 : enter name and password
 - CGI program 1: validate name and password; this guy is ok
 - Form 2: request data ...
 - CGI program 2: data access requires name, password to be entered
 - **Server has no record that these already checked**

http: state

- Other problems are "shopping carts":
 - Form (or forms) where users request items, need to build up collection of requested items
 - Form where users enter remaining order details.
 - Users expect a single composite order to be handled.
 - Problem: *remembering the contents of the collection of requested items*

http Hacking state maintenance

- Solutions:
 - Hidden fields
 - Cookies
 - URL rewriting
 - Hidden fields – all state data transferred back to client, then returned to server later
 - Cookies, URL rewriting – *usually* a magic key sent to client, keyed state data on server
- All these illustrated in later examples