How the Web is Structured

1 Background

To be able to follow the material in this course, you will need to know SQL (which you do know, since you have taken 484), and also some basics of operating systems and networks. This section provides you with a brief look at the key concepts you will need.

1.1 Operating Systems

The web, ultimately, is all software. In other words, browsing the web or running a web-server involves executing some programs that have been written to perform the appropriate tasks. An engineer trying to understand the web as a software system must keep in mind that programs execute on computers as operating systems processes. We must be clear about which processes are running, about where in the process the current execution is, and about the forking of new processes.

The notion of a process is central to modern operating systems. This is the boundary that provides isolation to multiple users sharing a machine. But it is also heavy-weight, with a significant start up cost and a significant cost to send messages across process boundaries. For reasons of security and predictability, we will often wish to have components running in different processes. For reasons of efficiency, we will often wish the opposite. Making intelligent tradeoffs between these opposing desires is important.

The tradeoff mentioned above shows itself in many scenarios. An interesting tradeoff point that operating systems have begun to provide is the notion of a *thread*. You can think of a thread as a lightweight process. It is cheaper to start and to shut down than a full-blown process, but it also does not provide the security one gets from isolation of address space. Where parallelism is of value, threads can be particularly attractive.

1.2 Networking

There are seven layers in the networking stack. The top 3 are not necessarily ordered in the traditional way and are our focus.
Top 3 layers: Application (+ Presentation + Session). (Examples are HTTP, SMTP, FTP, nntp, telnet...)

Bottom 4 layers: Transport (E.g. TCP, UDP): Establishes reliable transmission of large amounts of information. (Retransmission and reordering of packets, etc.)
Network (IP): Routes a packet of information to the specified destination.
Data Link: Agree on zero/one encoding
Physical: Establish physical electrical connectivity

Once a data link layer connection has been established between two points, a sender can transmit bits to a recipient that it is directly connected to. However, in a large network, the sender and recipient are not likely to have a direct physical connection. A *circuit-switched* network, such as a telephone network, actually sets up a sequence of connections, through intermediaries, between sender and recipient, reserving suitable bandwidth on each data link along the way. The result is a good guarantee of bandwidth, and hence real-time communication quality. However, the reserved bandwidth is set aside whether or not actual communication is taking place. A *packet-switched* network, such as the internet, breaks up the data to be communicated into small packets, and then expects each packet to make its way independently from source to destination. The standard technique used to route packets in this manner is called IP or the internet protocol. It has the virtue of being very efficient, and of usually doing a good job though it offers no guarantees.

TCP, or the Transmission Control Protocol, is the standard protocol at the Transport layer for the internet. Since IP is a "best effort" protocol, there will be packets that get "dropped" along the way (e.g. because some queue overflowed). There will be other packets that take a very long time getting to the destination. Even if there is no inordinately long delay, packets will frequently arrive out of order at the destination by following slightly different routes. TCP manages the assembly of packets into a coherent message at the destination, reordering them into the source order. It also controls re-transmissions of packets that may have been lost.

Above this are the application, presentation, and session layers, and these are sometimes ordered. However, the operation of these layers really depends very much on what one is trying to do. So we will merge them all into a single notion of "application" in what follows.

The Internet, then, is a world wide network that allows communication between senders and recipients using TCP/IP. Many applications can be run on this network. For instance e-mail is a popular application – it runs using the Simple Mail Transfer Protocol (SMTP). Similarly, newsgroups work using the Network News Transfer Protocol (NNTP). You have probably

obtained access to remote files using the File Transfer Protocol (FTP). All of the above application layer protocols can be run on the Internet.

Our interest is in a particular application layer protocol known as the HyperText Transfer Protocol, or HTTP. We will have much more to say about HTTP below.

2 HTTP

HTTP is a *Request/response* protocol. Each action is a request from the client to the server, followed by a response from the server to the client. There is no way for the server to initiate communication, using HTTP. Also, basic HTTP is stateless – this means that the next time the same client speaks to the same server, there is nothing in the HTTP protocol that allows the client to identify itself as the same one from 5 seconds ago and no way within HTTP for the server to remember the context of the previous interaction.

Communication in each direction includes a header, with all sorts of useful fields, and an optional body, with the actual payload. The protocol itself specifies the fields in the header, and an implementation of the protocol will interpret these fields. The protocol implementation will do nothing about the body, which is just treated as a bucket of bits by HTTP.

There are several, progressively more extensive, versions of HTTP, with HTTP1.1 being the most current. There are a large number of header fields defined, of which we will look at a couple now, and several more later on. There are three primary methods of interest: GET, HEAD, and POST. Each of these is a request for a resource. Response provides the resource, if possible, and also includes, among other things, a status code field in its header. E.g. "200 OK" or "404 Not found".

The GET method is the one used most often by far. The client specifies the resource of interest to it, by placing the requisite resource identifier portion of the URL in the header. E.g.

```
GET /foo.html HTTP/1.1
Referer: http://www.google.com/search?q=web+databases
```

The server, in response, will send the file foo.html back, if indeed such a file exists. GET has some limitations, e.g. only characters in ISO-8859-1 may be used to identify the resource, and no more than 1024 bytes may be used. Note the referer field in the request, indicating that this request was obtained as a link from a google search page. The web-server may use

this information to appreciate the traffic that google is sending to its page foo.html. But note also, that the field is set by the client, and there is nothing to prevent the client from not sending this field at all, or populating it with an erroneous value.

The POST method is similar to GET, except that it has a body in addition to the request header. Whenever copious or complex information is to be sent, POST is the method used. The flexibility afforded by having a separate body part comes at a cost: storing POST requests is problematic, since just the header does not suffice. Similarly, POST requests can not be linked to or bookmarked.

The HEAD method is similar to the GET method, except that the resource requested is *not* sent by the server. Why would any one bother to request a resource that they will not get? Because, they still get the header back. Where the resource of interest is large, a client may just request the header to check whether the resource has been updated recently, for instance, and in this case would prefer to use the HEAD method rather than the GET method.

```
HTTP/1.1 200 OK
Server: Apache/1.3.12 (Unix)
Cache-Control: max-age=60
Content-Type: text/html
```

The server field indicates the type of software used in the web server. The content type field tells the recipient how to interpret the body, using MIME-standards, as in http://www.isi.edu/in-notes/iana/assignments/media-types/. In the example above, this is just plain html. The cache control field we will have more to say about when we get to caching.

3 Implementing HTTP

We understand HTTP in abstract terms. Now we need to think about how it is to be implemented. There is an HTTP client running on the client machine, typically as part of a web browser. There is an HTTP server running on the server machine, typically as part of a web server. The web browser is normally a process on the client machine, and runs for a long time, issuing many HTTP requests in this time. The best known browsers are Netscape and Microsoft's Internet Explorer, but there are dozens of other browser products.

The web server story is a little more complicated. There is nothing to do on the server machine until and HTTP request arrives. So the simplest design would be to initiate a server process when a request arrives, respond to the request, and then terminate the process. Due to the high overheads of starting and killing processes, this is not an acceptable solution. A better solution is to run a process on the server machine in a continuous loop, listening for HTTP requests, and responding to the request each time it gets one. Such a design is reasonable if requests are infrequent, with idle time between requests.

When the server is loaded, we would wish to process multiple requests in parallel, rather than queue them up for the single server process to manage. This can be accomplished by having multiple server processes run in parallel – each can pick up the next request in queue to process. (Note that we are relying upon the operating system to maintain a queue of HTTP requests – usually managed by means of a fixed size buffer allocated for this purpose). How many processes to run in parallel now becomes a question – running too many will needlessly overload the system and affect performance, running too few will not provide sufficient parallelism and hence affect performance. One popular technique is to adjust the number of processes dynamically – more processes are created as queue length grows, and processes are terminated if they are idle for too long. Note that the time constants for this dynamic adjustment must be set to be such that we are not perpetually adding and terminating processes each time a request comes and is serviced.

If one has a typical server machine with one CPU, why is there a benefit to serving multiple requests in parallel? After all, the CPU can only do one thing at a time, so the most effective thing to do is just to have everyone take their turn. This question can be answered in two ways. First, and most important, not all of the response time is CPU-based. There typically is a disk look-up involved, and there is little for the CPU to do in a sequential program once it has issued a read request to disk until that request is completed. Similar delays can be caused when requests have to be made to other back-end servers and databases, each of which may take time to respond. Operating systems are good at recognizing when processes get to such wait states, and are able to swap in a different process that can make progress while the first one is waiting. Having multiple processes with requests ready to go in parallel provides the opportunity for such a swap. A second reason to consider multiple requests in parallel is variability in response time. Suppose that most requests to a server are quick to handle, but some involve a great deal of computation, and so take time. While one of these expensive

requests is being served, hundreds of quick requests can pile up in queue and be delayed considerably. With multiple server processes, we can expect at least some servers to be able to process requests rapidly even if a few are stuck serving expensive requests.

More complex engineering schemes are possible. For instance, [PDZ99] has proposed that a single server process do initial processing of HTTP requests, and quickly take care of the ones that are very cheap, e.g. resource not found or resource found in main memory buffer cache. The remainder get farmed out to a pool of parallel processes that operate just as above, waiting for disk and back-end servers as needed. We will have more to say about schemes in this spirit when we get to our discussion of scalability.

Running the server as multiple threads instead of processes is a desirable option on systems that support threads. One gets all the benefits of parallelism that processes have to offer, with less overhead. The address space is shared between threads, but in this case that is actually an advantage, making it possible to communicate information cheaply from one thread to another.

In short, a web server is a complex piece of software that runs on the server machine. The web server's primary task to is execute the server end of the HTTP protocol, and to serve up files that have been requested as web pages. Popular web servers include Apache, IIS, and AOLServer. Note that a web server is not the same thing as a web site. The latter is a conceptual entity, representing content, while the former is software. In fact, many commercial hosting service providers will run dozens of web sites on a single web server.

4 Addresses on the Web

Resources one the web are typically identified by means of a Uniform Resource Locator or URL. A URL is of the form

```
http://server:port/path#fragment?search
```

Default port is 80 for http, 443 for https Frequent additional ports are 8000 and 8080.

A internet-wide *domain name service* (DNS) is used to translate a human understandable server name (such as www.umich.edu to a numeric IP address, which may look something like 135.22.87.1. We will, later on, study DNS and how it works. For now, all we need to know is that a web browser can take a URL, find the server specification part of it, can have that converted to an IP address using DNS. A TCP connection is now established

to the server and port specified. An HTTP GET request can be sent on this TCP connection, requesting the resource identified by the remainder of the URL (the path, fragment, and search string).

Note that a subsequent URL to the same server and port typically involves a fresh TCP/IP connection, and a brand new request to the web server. However, the DNS look-up of the IP address would typically have been cached in the browser, and so does not need to be looked up again.

Typically the path specifies a file name, and this is the file to be retrieved and returned. What if the URL specifies a directory? The server may have a convention in this case. Typically it would look for a file called index.htm or index.html, and return this file if it exists. If not, perhaps provide a directory listing. Exactly which file to look for, and what else to do is completely up to the way the web server has been configured: the protocol does not specify what should be done.

The fragment specification merely points to a portion of the resource being retrieved. In a typical implementation, with HTML, the entire resource is still obtained, the fragment of focus is just identified. In other words, the server does nothing differently with this specification. Due to this, browsers may frequently not send the fragment specification to the server. Rather, they may keep this local, and once the file has been retrieved, use the fragment specification to scroll down when displaying the file and position the cursor at the desired location.

The search specification provides a general purpose mechanism to send parameters to the web server that could be used to modify the request. There are many different ways in which these parameters are used, and we will see some of these when we get to dynamic web pages shortly.

Relative URLs allow skipping some of the earlier parts in the specification, but not in the access.

Content on the Web

We have learned how to transfer files across the web. But what do these files contain? How should we interpret their content? How do we get from fetching files to doing interesting things for the web user? These are the issues we consider here.

5 Text Encoding

The base-level standard for conveying information is a string of characters, or plain text. But what is a character? For English language speakers, the answer may be the twentysix characters of the alphabet, possibly in upper and lower case, the 10 numeric digits, and several punctuation symbols. But this set of characters is not adequate for many languages other than English, and does not include many symbols that even an English-speaker would use, such as mathematical symbols.

A *character set* (or repertoire) is a list of characters that may appear in a document. ASCII is the English language standard. "Latin-1" (ISO 8859-1) allows encoding most European languages, including accent marks, Greek, Arabic, Hebrew, and Cyrillic characters etc. Unicode is a standard, by the Unicode Consortium, which defines a character repertoire intended to be "universal". While it has been widely adopted, and is strongly supported by many in the computing industry, it has not found wide acceptance in the countries where definition of a standard character set is hard, as in the pictogram-based languages of Eastern Asia.

Given a character set, we have to agree on a way to store this on a computer. A *character encoding* is a way of storing characters on a computer as bits. For the same character set, there could be multiple encodings, so it is important to supply the character encoding for any document. For character sets such as ASCII and Latin-1, there is only one popular encoding, so that people frequently use the same name to speak of both character set and character encoding. In fact, the word *charset* is used to refer to an encoding, causing much confusion between character set and character encoding.

English-language personal computers used in the United States employ a 7-bit character code called American Standard Code for Information Interchange (ASCII), which allows for a character set of 128 (7 bits can code for $2^7 = 128$ values) items of upper and lower case Latin letters, Arabic numerals, signs, and control characters to be used. Since the usual unit of information is a byte with 8 bits, we have one character left over, and this

is typically used as a parity bit for error detection. Latin-1 is encoded using the full 8 bits in the ISO 8859-1 encoding.

The "native" Unicode encoding, UCS-2, presents each code number as two consecutive octets m and n so that the number equals $256m + n$. This means, to express it in computer jargon, that the code number is presented as a two-byte integer. This is a very obvious and simple encoding. However, it can be inefficient in terms of the number of octets needed. If we have normal English text or other text which contains ISO Latin-1 characters only, the length of the Unicode encoded octet sequence is twice the length of the string in ISO 8859-1 encoding. So a number of alternative, more efficient, encodings have been proposed.

One amongst these alternatives that is quite popular is UTF-8. In UTF-8, character codes less than 128 (effectively, the ASCII repertoire) are presented "as such", using one octet for each code (character). All other codes are presented, according to a relatively complicated method, so that one code (character) is presented as a sequence of two to six octets, each of which is in the range $128 - 255$. This means that in a sequence of octets, octets in the range $0 - 127$ ("bytes with most significant bit set to 0") directly represent ASCII characters, whereas octets in the range $128 - 255$ ("bytes with most significant bit set to 1") are to be interpreted as really encoded presentations of characters. The basic idea is to use less bytes for commonly occurring characters and more bytes for rare ones, to obtain a smaller number of bytes on average than the native UCS-2 encoding, which always requires 2 bytes per character.

6 Document Formats

The typical format in which a document is organized is HTML, the hypertext markup language. HTML is a markup language, which means that the document is a text document, but is organized (and interpreted) according to the rules of HTML. We will say more about HTML in the next section.

Note that HTML is completely separate from HTTP. In fact, HTTP can be used to transfer data in any format whatsoever. It is just that HTTP and HTML together define the basic infrastructure of the world wide web.

There are many popular data representation formats, including Microsoft office file formats, Adobe PDF (portable document format), postscript, a variety of graphics formats (including GIF and JPEG), and a large number of file and archive compression formats. There is nothing about HTTP that restricts the format of the data being transferred to HTML. Files encoded in

any of these other formats can, and indeed frequently are, transferred using the HTTP protocol. Web browsers could be engineered to display data in these formats, and for a few of the more popular ones, they do build in such facilities. However, for the most part, it should be assumed that the browser itself does not understand the format of the data file obtained. Rather, it maintains a list of "helper applications" associated with known formats, and calls upon the appropriate such application when needed. In the simplest form, this causes a new process to be started for the helper application. In a more efficient arrangement, when the helper application process is already running, the downloaded data is transferred to this process.

One other particularly important format for data representation and transfer is XML. We will have a great deal to say about XML later on.

7 Dynamic Web Pages

A dynamic web page is not stored as is. Rather, it is created on demand, which means that some programming is required to create it. There are two places where the program could run, at the client and at the server. We discuss these two options below, and then talk about tradeoffs.

7.1 Client Side Solutions

Since we are transferring arbitrary files across the web, there is no reason that we couldn't transfer programs. These programs could then execute on the client machine, and do whatever wonderful things the server would want them to do. Indeed, HTTP is sometimes used to download executable code explicitly, e.g. a shareware program that the web user wishes to obtain across the web, then install and run on the client machine.

However, there are issues when obtaining programs across the web. An immediate issue has to do with the operating system on the client machine – there are many possibilities, and typically a different executable is required for each. Indeed, your favorite executable download site very likely gives you a number of choices from which to choose the version of executable you need for your system configuration. Making this work transparently is hard. Even if the client is shielded from negotiation of system type by the download protocol, at the very least we require the creation of these multiple executable versions at the server end.

The Java virtual machine (JVM) concept helped to address this issue. Each client could define a JVM, which would be standard by definition. Executable code could then be compiled to run on this virtual machine, thereby

sidestepping the difficulty with supporting multiple operating systems. Web pages could then send executables, known as *applets* for execution at the client.

Given that we figured out how to make applets work, we are immediately faced with the question of preventing abuse. Since applets can execute arbitrary code on the client machine, it is important that they be trusted not to do any harm – trust that a web-client may be unwilling to extend to a web server that it knows little about. A second issue with applets as we have defined them here is that they are quite heavy-weight, requiring the creation of a new operating system process for each applet at the client.

Client-side scripting is used to overcome these. Web-pages include "scripts" in languages such as Javascript, VBScript, and Jscript. In HTML, tags are used to identify script boundaries. For instance:

```
<SCRIPT Language = "javascript"> document.writeln("Hello World"); </SCRIPT>
```

A browser is able to recognize the "SCRIPT" tag, and then check to see if it supports the specified scripting language. If it does, it executes the script specified. For instance, the simple script above will cause "Hello World" to be written to the screen.

Scripts are lightweight – they are executed by the browser, within the browser process, so it is reasonable to use scripts to do even very small and simple things. (E.g. make some object on the page change when the mouse is placed over it). Browsers are also able to impose limitations on what scripts are permitted to do – in particular, they can prevent scripts from looking at browser state, or communicating information from the client machine to the server, or any other action that the client wishes to prevent. Of course, this limitation has to be coded into the browser – so hackers can try to find new loopholes with scripts that browser vendors have not yet closed.

Client-side scripts should be used for simple actions that do not require interaction with the server, as well as for actions that require immediate response for a reasonable user experience. For most other things it is usually more appropriate to run the program at the server side rather than the client side, since the server can exercise much more control on the server side, as well as implement code much more efficiently. Furthermore, less reliance on clients means less clients that block actions the server wold like to take because of a lack of trust.

7.2 Server Side Solutions

The server can customize (or build) the web page for a specific request, by invoking a program on the web server, or a different machine on the server side of the internet. For example, the program may evaluate a query against a database using values that a user entered in a form and submitted as part of the HTTP request. The result of the database query is then packaged into an HTML page and sent back to the client.

It is often useful to think of these actions as comprising two main components – first there is the computation of result (such as the evaluation of database query) and second there is the generation of HTML. Both components could be specified in most programming languages. However, there is a difference in the ease with which this can be done. Server-side scripting languages focus on the HTML web page creation, with small computational components. Regular programming languages, such as C++ or Java, make it easy to write complex programs but provide no special support for generating the necessary HTML. We list here a few common techniques used for server side programming:

1. Server side includes. This is the lightest weight technique that is used on the server side. It comprises *directives* that are interpreted by the web server itself. The advantage is that there is no call to any external process, so it is very cheap. The disadvantage is that it is limited to the set of directives supported by the server. A very common use of this technique is a line at the bottom of the page that tells you when the page was last updated. The set of directives supported can vary based on the specific web server being used. The list of directives for Apache is available at http://httpd.apache.org/docs/howto/ssi.html.

2. CGI. The *Common Gateway Interface* is perhaps the oldest server side programming technique. It is also the most expensive. A separate process is forked, and this can execute code written in any programming language of choice. Parameters to this process are usually supplied by means of environmental variables.

3. Scripting languages. There are several popular languages that can be intermixed with HTML. The server has to be alerted to look for this intermixed code before sending the page to the client. It then scans the page, and invokes the script execution engine to run any code that it finds, between script begin and end tags. Some of the more popular scripting languages are PHP, usually run in conjunction with

the Apache web server; ASP (Active Server Pages) from Microsoft, usually run in conjunction with the Microsoft IIS web server; and JSP (Java Server Pages) which allows Java code to be intermixed with HTML.

7.3 PHP

```
<?php ... ?>

uninitialized variables produce no output

header must go before body
```