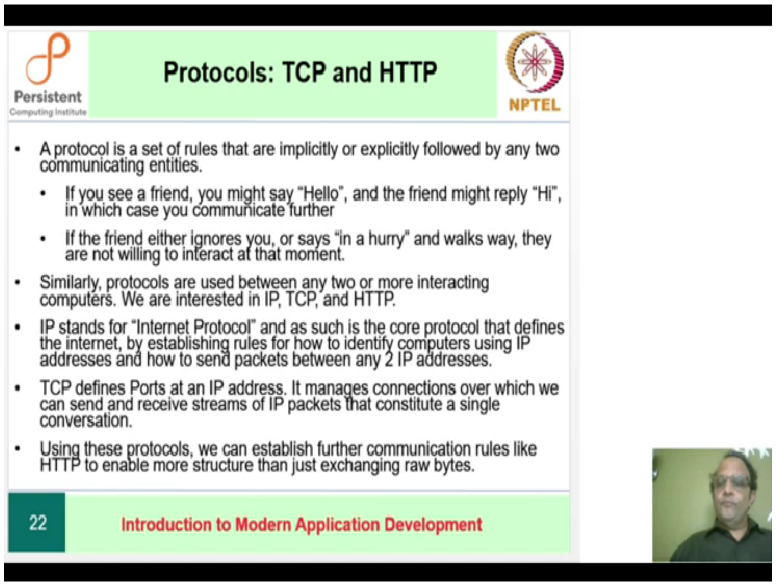
Introduction to Modern Application Development

Prof. Aamod Sane
FLAME University and Persistent Computing Institute
Abhijat Vichare
Persistent Computing Institute
Madhavan Mukund
Chennai Mathematical Institute

Lecture-15
Session 2-Part2

(Refer Slide Time: 00:02)



Protocols: TCP and HTTP

- A protocol is a set of rules that are implicitly or explicitly followed by any two communicating entities.
  - If you see a friend, you might say "Hello", and the friend might reply "Hi", in which case you communicate further
  - If the friend either ignores you, or says "in a hurry" and walks way, they are not willing to interact at that moment.
- Similarly, protocols are used between any two or more interacting computers. We are interested in IP, TCP, and HTTP.
- IP stands for "Internet Protocol" and as such is the core protocol that defines the internet, by establishing rules for how to identify computers using IP addresses and how to send packets between any 2 IP addresses.
- TCP defines Ports at an IP address. It manages connections over which we can send and receive streams of IP packets that constitute a single conversation.
- Using these protocols, we can establish further communication rules like HTTP to enable more structure than just exchanging raw bytes.

22 Introduction to Modern Application Development

A protocol (or communication protocol) is a set of rules that are implicitly or explicitly followed by any two entities that wish to communicate or transmit some information. For example, suppose you want to talk to a friend. When you see your friend, you might say something like, hello. And then the friend replies Hi. And in that case, it is clear that both of you wish to communicate with each other at this time. On the other hand, if the friend happens to be busy, they might say something like, "Oh, I am in a hurry" and walk away.

This kind of implicit social rules governs all communication, at least amongst people. And then there are more complicated versions of these, for example, how one should address a dignitary like the president of a country, their parents…etc. We all know and are so used to these rules that many of use never realize how arbitrary these rules are.

The issue replicates itself in the case of computers also. For example, suppose two computers wish to talk to each other, how does one computer will initiate a communication? If one of them is busy, then what are they supposed to do? These are the reason why computers also have ***communication protocols***.

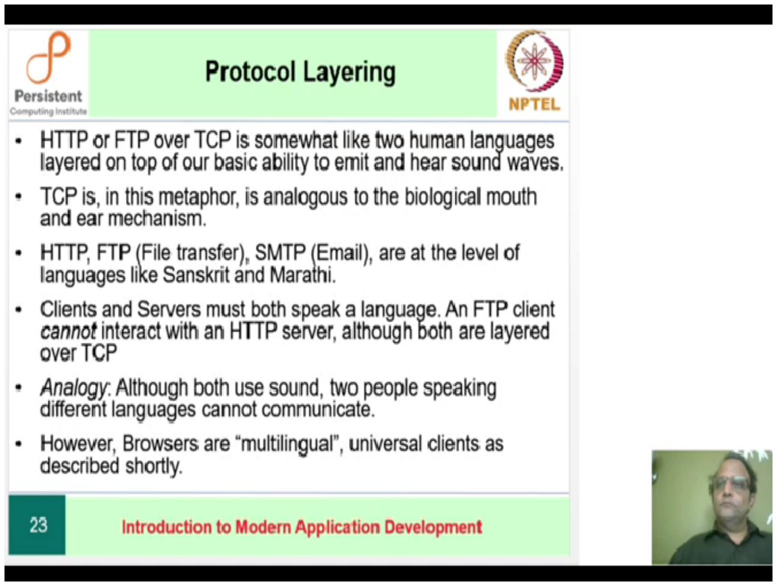
The particular protocols that we are interested in are **IP, TCP** and **HTTP.** *IP stands for Internet Protocol*. And as such, it is the core protocol that defines pretty much what it means to be the internet. As an example, the core functionality of what we call the internet is to be able to identify a computer and somehow reach it.

This job is done by IP. Here "reach" has a very specific meaning. Reach means something like: Can I send the envelope containing certain data, which is known as a packet, to another computer, specified address, somehow and some machinery should exist which takes this packet from this computer to that computer. And so, we can say that IP is the protocol that defines or establishes the internet.

IP is what is called a ***Datagram protocol*** or a **packet protocol** because all it does is it sends chunks of data. Now on top of this, we use another protocol TCP. TCP stands for ***Transmission Control Protocol***, and it define something called ports which we have seen before just now. So, a port is a service that can exist on the machine that represents a service that can exist on a machine and a TCP or entity which wishes to speak with a TCP port on one side will be able to establish a connection with another entity with a port on another side.

And then once the connection is established, you can send packets of information that to the user of the connection feels like a continuous data stream. So, using these protocols, we can further establish communication rules, such as HTTP and FTP and many others that put additional constraints on just what kinds of messages can be exchanged. This sort of set is considered to be a protocol.

(Refer Slide Time: 03:33)



Protocol Layering

- HTTP or FTP over TCP is somewhat like two human languages layered on top of our basic ability to emit and hear sound waves.
- TCP is, in this metaphor, is analogous to the biological mouth and ear mechanism.
- HTTP, FTP (File transfer), SMTP (Email), are at the level of languages like Sanskrit and Marathi.
- Clients and Servers must both speak a language. An FTP client *cannot* interact with an HTTP server, although both are layered over TCP
- *Analogy*: Although both use sound, two people speaking different languages cannot communicate.
- However, Browsers are "multilingual", universal clients as described shortly.

23 Introduction to Modern Application Development

When multiple protocols cooperate in some fashion, they are said to be layered. Here, layering has a very strong analogy with how any communication facility must work. So, for example, HTTP or FTP over TCP is a bit like two different human languages which are layered on top of our core ability to emit and hear sound waves. So, in this metaphor, TCP is equivalent to the mouth and ear mechanism, which allows us to speak and listen different kind of sound.

The next layer up over this core biological functionality is things like languages like Sanskrit or Marathi, which invent their own rules by which people can talk to each other. So, the analog is to HTTP, FTP, SMTP… etc.

- HTTP: **Hypertext Transfer Protocol**
- FTP: **File Transfer Protocol**
- SMTP: **Simple Mail Transfer Protocol**.

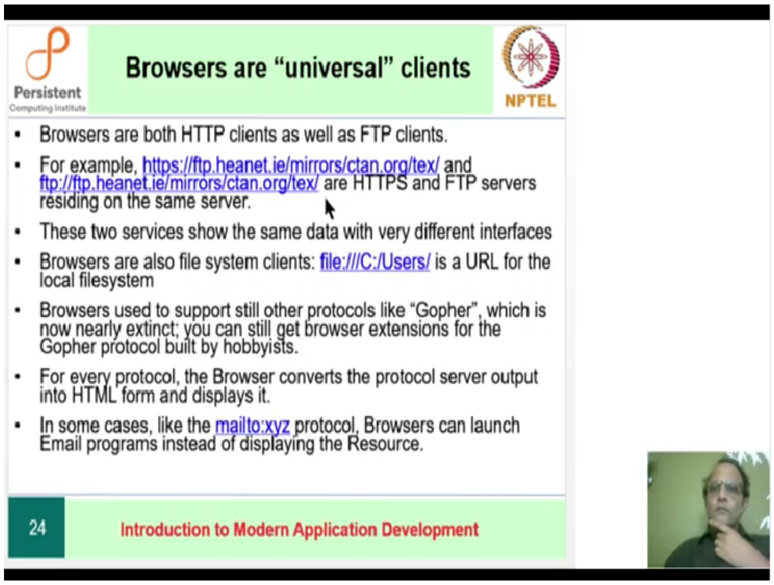
So, these protocols are like higher level languages on top of basic ability to make and here song.

One characteristic of such a mechanism is that it is very much like the way we structure programs using functions. There are general purpose functions which are used by special purpose functions. Similarly, the special purpose protocol TCP uses a more general protocol called IP. And in turn HTTP, which is a still more specialized protocol uses the general protocol called TCP.

The next part is to notice one other thing about this kind of layering of languages, which is that clients and servers must both speak the same language in order to be comprehensible to each other. So, the analogy here is that even if two people who can speak and hear sounds but speak different language, cannot properly communicate with each other.

So, in this sense, FTP client cannot talk to HTTP server. You might expect that say, strictly speaking, but HTTP client cannot talk to FTP server either. This is true. On the other hand, browsers can talk both HTTP and FTP. And this is because browsers are multilingual, universal clients, as we will discuss shortly.

(Refer Slide Time: 06:30)



Browsers are "universal" clients

- Browsers are both HTTP clients as well as FTP clients.
- For example, https://ftp.heanet.ie/mirrors/ctan.org/tex/ and ftp://ftp.heanet.ie/mirrors/ctan.org/tex/ are HTTPS and FTP servers residing on the same server.
- These two services show the same data with very different interfaces
- Browsers are also file system clients: file:///C:/Users/ is a URL for the local filesystem
- Browsers used to support still other protocols like "Gopher", which is now nearly extinct; you can still get browser extensions for the Gopher protocol built by hobbyists.
- For every protocol, the Browser converts the protocol server output into HTML form and displays it.
- In some cases, like the mailto:xyz protocol, Browsers can launch Email programs instead of displaying the Resource.

24 Introduction to Modern Application Development

Browsers in the sense that we have just discussed can speak multiple languages. Here are a few examples. I have shown 2 sites for a service called CTAN. CTAN is short for comprehensive **tex** archive network, which was one of the earliest successful users of FTP and HTTP servers since the early days. So, it stands to reason that that kind of service with its long duration offers still offers both FTP and HTTP access to the information. These two services show the same data with two very different interfaces.

As another example, browsers can also look at the local file system. So, we say that browsers or file system clients, something like `file:///C:/users/` on windows is a URL for the local file system.

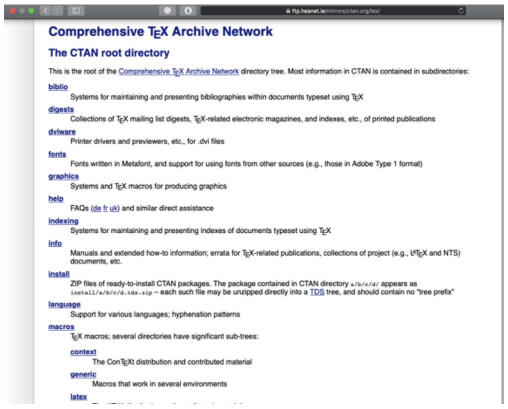
Browsers used to support still other protocols. An old protocol one of the early protocols on the internet was something called **Gopher**. And for a long-time browser supported this protocol as well. So, what browsers did was they became a gateway from any protocol to, in effect, the world of HTML and the web in general, which is why we make a distinction between the internet and the web.

The net is the net, which is defined by things like TCP and IP, whereas the web specifically is largely about browsers, HTTP and HTTP servers. Although there are ways to adapt to the older world and different services like FTP. Gopher, for instance, is now nearly extinct, except for a few servers and browser extensions, which are maintained largely by hobbyists.

What the browser does is for every kind of data that it gets from a server, it converts that effectively into HTML, and then displays it. But it is not the case that the browser can display everything. For example, the older versions of browsers used to launch separate programs on the local desktop for opening a PDF. However, nowadays browsers have PDF reader built-in. Still, that is not true for every kind of URL. Even now, the URL `mailto:xyz` is such a URL that if you click on it, the browser will launch email program instead of displaying the resource because there is nothing to be done. Thus far at least browsers have not evolved to become email clients in the sense that the `mailto` protocol wants you to, even though as we know browsers make excellent mail clients by using websites like Gmail, Yahoo Mail, Proton Mail… etc. However, it must be noticed that modern browsers allow the user to select the default action to handle a `mailto` protocol request. For example, we can go to Settings in Chrome and set Gmail to be the default email client, or use a chrome extension to manage `mailto` requests.

(Video Starts: 09:45)

Let us take a look at the examples we just discussed. Here, for instance is the website `https://ftp.heanet.ie/mirrors/ctan.org/tex` which is accessed as an HTTPS service. And as a result, we get a normal HTML page, which shows you what the contents of this thing are. Now, the same information is available on also from `ftp://ftp.heanet.ie/mirrors/ctan.org/tex`. But with the FTP protocol, we observer that the presentation here looks much more like ordinary file system compared with the nice HTML layout that we have seen in the HTTPS case.



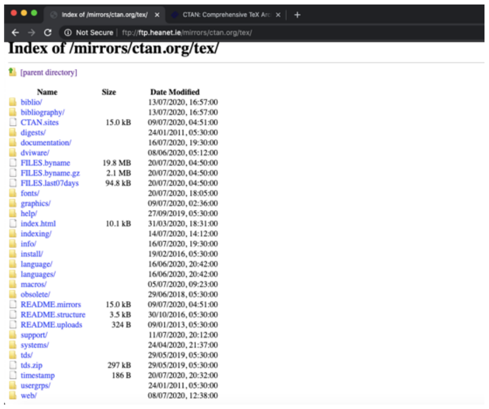
Comprehensive TEX Archive Network

The CTAN root directory

This is the root of the Comprehensive TEX Archive Network directory tree. Most information in CTAN is contained in subdirectories:

biblio
Systems for maintaining and presenting bibliographies within documents typeset using TEX
digests
Collections of TEX mailing list digests, TEX-related electronic magazines, and indexes, etc., of printed publications
dviware
Printer drivers and previewers, etc., for .dvi files
fonts
Fonts written in Metafont, and support for using fonts from other sources (e.g., those in Adobe Type 1 format)
graphics
Systems and TEX macros for producing graphics
help
FAQs (de fr uk) and similar direct assistance
indexing
Systems for maintaining and presenting indexes of documents typeset using TEX
info
Manuals and extended how-to information; errata for TEX-related publications, collections of project (e.g., LATEX and NTS) documents, etc.
install
ZIP files of ready-to-install CTAN packages. The package contained in CTAN directory `a/b/c/d/` appears as `install/a/b/c/d.tds.zip` – each such file may be unzipped directly into a TDS tree, and should contain no "tree prefix"
language
Support for various languages; hyphenation patterns
macros
TEX macros; several directories have significant sub-trees:
context
The ConTEXt distribution and contributed material
generic
Macros that work in several environments
latex

Figure 1 View of https://heanet.ie



Index of /mirrors/ctan.org/tex/
CTAN: Comprehensive TeX Arc
Not Secure
ftp://ftp.heanet.ie/mirrors/ctan.org/tex/

Index of /mirrors/ctan.org/tex/

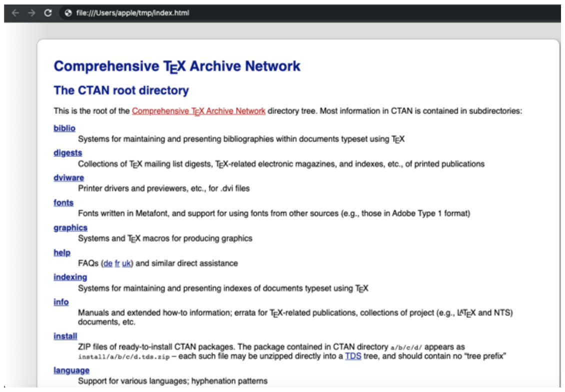
[parent directory]

Name	Size	Date Modified
biblio/		13/07/2020, 16:57:00
bibliography/		13/07/2020, 16:57:00
CTAN.sites	15.0 kB	09/07/2020, 04:51:00
digests/		24/01/2011, 05:30:00
documentation/		16/07/2020, 19:30:00
dviware/		08/06/2020, 05:12:00
FILES.byname	19.8 MB	20/07/2020, 04:50:00
FILES.byname.gz	2.1 MB	20/07/2020, 04:50:00
FILES.last07days	94.8 kB	20/07/2020, 04:50:00
fonts/		20/07/2020, 18:05:00
graphics/		09/07/2020, 02:36:00
help/		27/09/2019, 05:30:00
index.html	10.1 kB	31/03/2020, 18:31:00
indexing/		14/07/2020, 14:12:00
info/		16/07/2020, 19:30:00
install/		19/02/2016, 05:30:00
language/		16/06/2020, 20:42:00
languages/		16/06/2020, 20:42:00
macros/		05/07/2020, 09:23:00
obsolete/		29/06/2018, 05:30:00
README.mirrors	15.0 kB	09/07/2020, 04:51:00
README.structure	3.5 kB	30/10/2016, 05:30:00
README.uploads	324 B	09/01/2013, 05:30:00
support/		11/07/2020, 20:12:00
systems/		24/04/2020, 21:37:00
tds/		29/05/2019, 05:30:00
tds.zip	297 kB	29/05/2019, 05:30:00
timestamp	186 B	20/07/2020, 20:32:00
usergrps/		24/01/2011, 05:30:00
web/		08/07/2020, 12:38:00

Figure 2 View of ftp://ftp.heanet.ie

But here is something interesting. The HTML layer that we have seen is usually available in a file called `index.html`. Indeed, when we look at HTTPS for this resource, implicitly, this thing is fetching a file called `index.html` which is sent by the server. We do not have to say that is one of the things that is automatically done by servers.

When we try to open the index.html file using the FTP protocol, as far as Firefox knows, this is FTP service, hence it does not know immediately know what to do with that file. Here though, Firefox does something interesting. It says I am trying to open `index.html`. I do not know what to do with it since I got it from an FTP server, and the indication is not there. But then I do know that `.html` files can be opened by Firefox. So, we get prompt which asks us whether this file should be opened with Firefox. It might seem a bit strange, but this is quite logical when we consider the case that we could have requested any type of file from the FTP server like plain text, an application file…etc. And if you say yes, by all means, go ahead and open it, then it gets a local file here on my machine, and that file is identical to the file that to the web page because it is the same HTML file. This is the interesting example which shows you a distinction between a HTTP server and the FTP server.



file:///Users/apple/tmp/index.html

Comprehensive TEX Archive Network

The CTAN root directory

This is the root of the Comprehensive TEX Archive Network directory tree. Most information in CTAN is contained in subdirectories:

biblio
Systems for maintaining and presenting bibliographies within documents typeset using TEX

digests
Collections of TEX mailing list digests, TEX-related electronic magazines, and indexes, etc., of printed publications

dviware
Printer drivers and previewers, etc., for .dvi files

fonts
Fonts written in Metafont, and support for using fonts from other sources (e.g., those in Adobe Type 1 format)

graphics
Systems and TEX macros for producing graphics

help
FAQs (de fr uk) and similar direct assistance

indexing
Systems for maintaining and presenting indexes of documents typeset using TEX

info
Manuals and extended how-to information; errata for TEX-related publications, collections of project (e.g., LATEX and NTS) documents, etc.

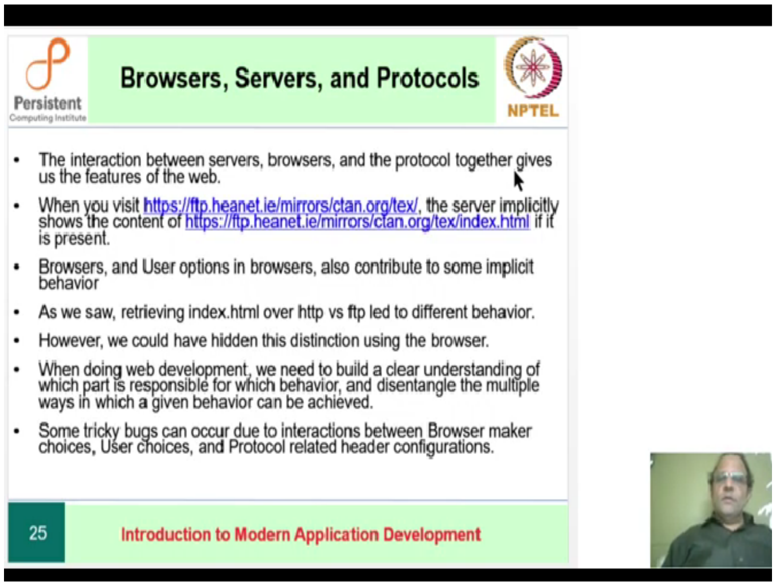
install
ZIP files of ready-to-install CTAN packages. The package contained in CTAN directory `a/b/c/d/` appears as `install/a/b/c/d.tds.zip` – each such file may be unzipped directly into a TDS tree, and should contain no "tree prefix"

language
Support for various languages; hyphenation patterns

Figure 3 Opening a local file in a browser

Implicitly, we have also seen the browser work like a file system information retrieval while doing this work. For example, we can see in the screenshot above (Figure 3) that the URL is the actual file protocol request to get from the `file:///Users/apple/tmp/index.html`, but because it happens to be an HTML file, firefox knows what to do with it now for display purposes, because it is retrieved from the *file server* rather than from an FTP server.

(Refer Slide Time: 13:34)



Browsers, Servers, and Protocols

- The interaction between servers, browsers, and the protocol together gives us the features of the web.
- When you visit https://ftp.heanet.ie/mirrors/ctan.org/tex/, the server implicitly shows the content of https://ftp.heanet.ie/mirrors/ctan.org/tex/index.html if it is present.
- Browsers, and User options in browsers, also contribute to some implicit behavior
- As we saw, retrieving index.html over http vs ftp led to different behavior.
- However, we could have hidden this distinction using the browser.
- When doing web development, we need to build a clear understanding of which part is responsible for which behavior, and disentangle the multiple ways in which a given behavior can be achieved.
- Some tricky bugs can occur due to interactions between Browser maker choices, User choices, and Protocol related header configurations.

25 Introduction to Modern Application Development

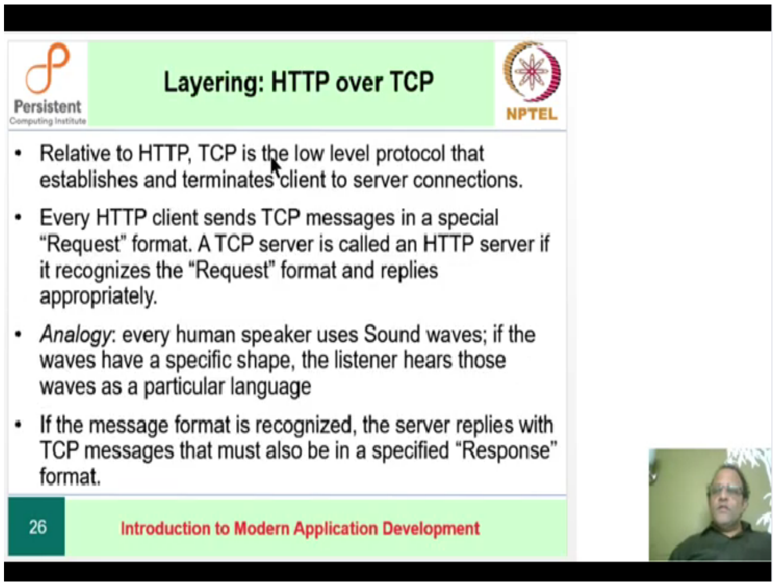
So, this is how a browser works as a universal plan. These examples show us that the features we experienced as users of the web are because of interactions between server features browser features and protocol features. Our first example was that when you visit a URL that is basically just a directory URL. The server implicitly shows the contents of the file index.html at that URL if it happens to be present.

Just like this, browsers will also have features and user options chosen by the users among the browser settings will also contribute to some implicit behavior. For instance, retrieving index.html over HTTP versus FTP led to a different behavior. The browser could have chosen to hide it. But either Firefox does not do it or I have myself set some behavior, which prevents that from happening. I no longer remember.

But since FTP so rarely use this particular behavior does not matter much. So that gives us another choice. This choice happens to be done in the browser. But as far as the user is concerned, they do not care. When doing web development, we need to build within ourselves a very clear understanding of all the different ways in which a particular feature could be achieved. Here to be aware of what kind of server you are getting it from, what are the settings the server is telling us about. How is the browser interacting and how is the user interacting with it.

This adds to the difficulty of programming. But again, as I said before, after a while, you get used to it and the culture around us along with websites like StackOverflow help us figure out what is really going on. Tricky bugs can come into existence due to browser makers choices, user choices and protocols, related header configurations, server configurations and an endless number of other things. It is useful to remember that these distinctions exist, and over time build up a clear understanding of what is the root cause for a particular effect.

(Refer Slide Time: 15:55)



Layering: HTTP over TCP

- Relative to HTTP, TCP is the low level protocol that establishes and terminates client to server connections.
- Every HTTP client sends TCP messages in a special "Request" format. A TCP server is called an HTTP server if it recognizes the "Request" format and replies appropriately.
- *Analogy*: every human speaker uses Sound waves; if the waves have a specific shape, the listener hears those waves as a particular language
- If the message format is recognized, the server replies with TCP messages that must also be in a specified "Response" format.

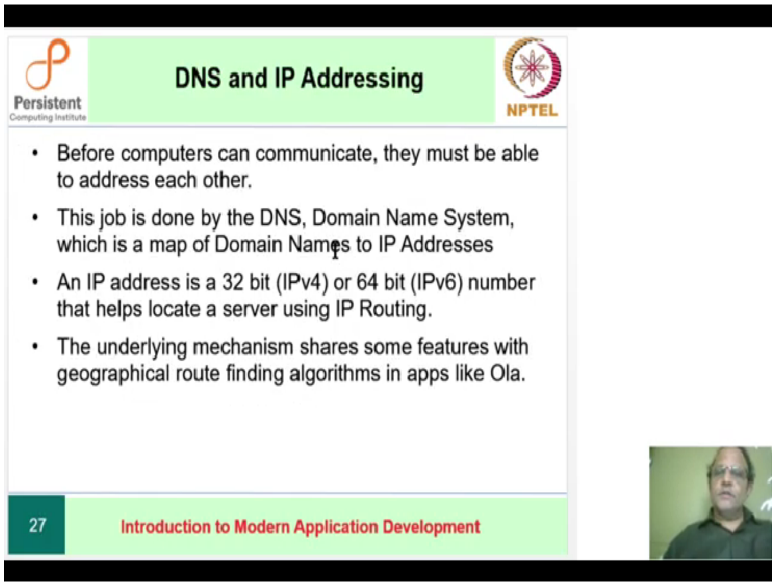
26 Introduction to Modern Application Development

We have seen that protocols exist in order to allow communication between different entities and that they are layer so that we can divide the responsibility of different parts of communication among different protocols. But the specific layering we care about most is the layering of ***HTTP over TCP***. Relative to HTTP, TCP is the low-level protocol that establishes and terminates client to server connections. Once TCP establishes a connection, a client that can speak HTTP will send specific TCP messages that bear a format which is known as the request format.

We say that a TCP server is a HTTP server, if it recognizes the HTTP request format, and replies appropriately with the HTTP response. We can use our earlier analogy of sound and language to understand what is going on here, every human speaker uses sound waves. But if the waves have a specific shape then we are capable of recognizing it as a particular language. So, it is with TCP and HTTP.

If the content of a TCP packet is of a particular kind, then the HTTP server can recognize it as an HTTP message and respond appropriately. So, the response format in turn tells the client that yes, it is succeeded in talking to a HTTP server. And the response it gets is an HTTP response. Protocols are all well and good of course, but before computers can communicate, they need to be able to address each other.

(Refer Slide Time: 17:50)



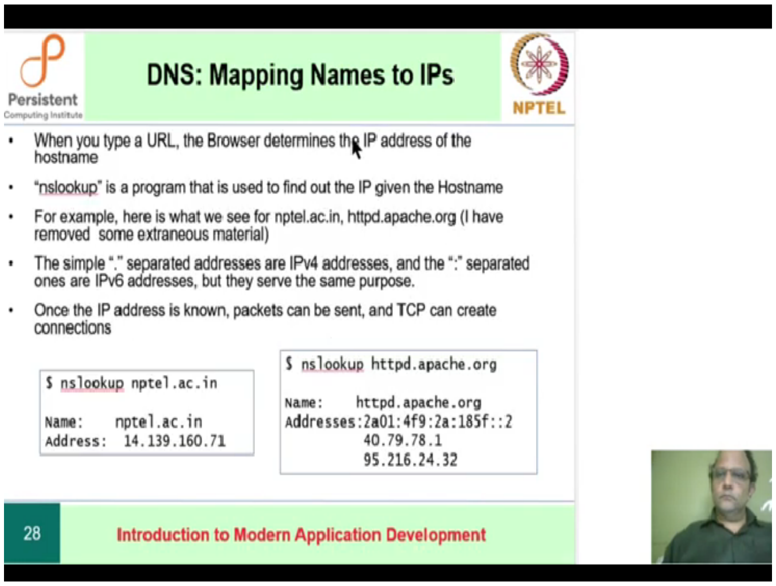
DNS and IP Addressing

- Before computers can communicate, they must be able to address each other.
- This job is done by the DNS, Domain Name System, which is a map of Domain Names to IP Addresses
- An IP address is a 32 bit (IPv4) or 64 bit (IPv6) number that helps locate a server using IP Routing.
- The underlying mechanism shares some features with geographical route finding algorithms in apps like Ola.

27 Introduction to Modern Application Development

This job is done by the system called domain name system, and the IP addresses which constitute the names of the machines that need to talk to each other The job of Domain Name System is to maintain a map between domain names and IP addresses. An IP addresses could be of 32 bit, in the case of IPv4, or 64 bit in the case of IPv6. The address helps locate a server using IP routing. The routing mechanism actually shares some features with geographical route-finding algorithms. For example, just as Google Maps tells how to go from an origin to a destination while keeping track of things like how much congestion is there and so forth, so does the route-finding mechanism used in internet which tells how to transfer a packet between various nodes so that it reaches its destination from the origin.

(Refer Slide Time: 18:45)



DNS: Mapping Names to IPs

- When you type a URL, the Browser determines the IP address of the hostname
- "nslookup" is a program that is used to find out the IP given the Hostname
- For example, here is what we see for nptel.ac.in, httpd.apache.org (I have removed some extraneous material)
- The simple "." separated addresses are IPv4 addresses, and the ":" separated ones are IPv6 addresses, but they serve the same purpose.
- Once the IP address is known, packets can be sent, and TCP can create connections

```
$ nslookup nptel.ac.in

Name:    nptel.ac.in
Address:  14.139.160.71
```

```
$ nslookup httpd.apache.org

Name:     httpd.apache.org
Addresses:2a01:4f9:2a:185f::2
          40.79.78.1
          95.216.24.32
```

28 Introduction to Modern Application Development

When you type a URL to browser determines the IP address of a hostname. You do not actually need a browser to find the IP address associated with a domain name/hostname; `nslookup` is a small program that is used to find out the IP given the hostname. You should try this out on your own, still, below I give you some examples below and indicate what it is that you are seeing.

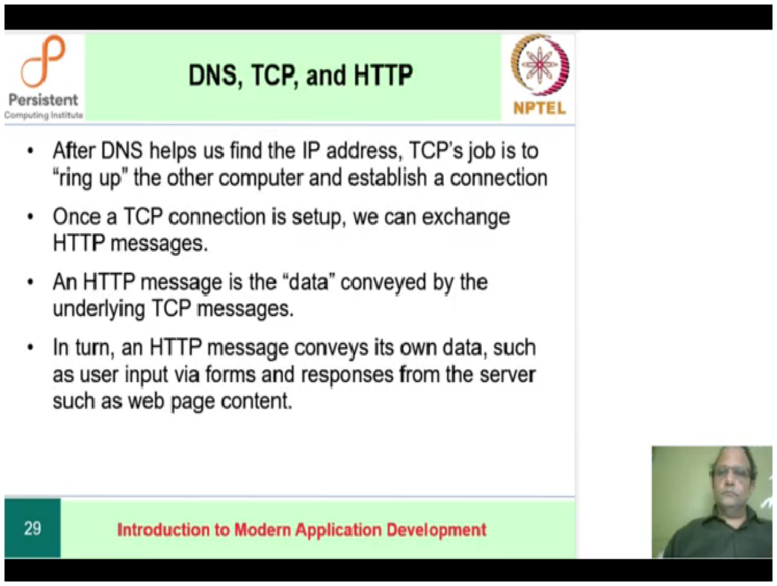
```
1. (base) HitLap:~ apple$ nslookup nptel.ac.in
2. Server:     172.20.10.1
3. Address:    172.20.10.1#53
4.
5. Non-authoritative answer:
6. Name:   nptel.ac.in
7. Address: 14.139.160.71
```

On the 7th line, we see the IP address of the site.

```
1. (base) HitLap:~ apple$ nslookup httpd.apache.org
2. Server:     172.20.10.1
3. Address:    172.20.10.1#53
4.
5. Non-authoritative answer:
6. Name:   httpd.apache.org
7. Address: 95.216.24.32
8. Name:   httpd.apache.org
9. Address: 40.79.78.1
```

You will not always get the same IP addresses there are mapping mechanisms which distinguish between a domain as approach from India, for example, versus a domain as approached from say, Japan. Nonetheless, the bottom line is that as far as the user is concerned you see the domain name, you find the IP address. And your expectation is that whatever content we are supposed to see at that domain name is also the content that you will see. As long as that happens, it does not matter to us, as the user, how this mapping is done. Once that is established, TCP can now establish a connection and packets can be sent.

(Refer Slide Time: 20:44)



DNS, TCP, and HTTP

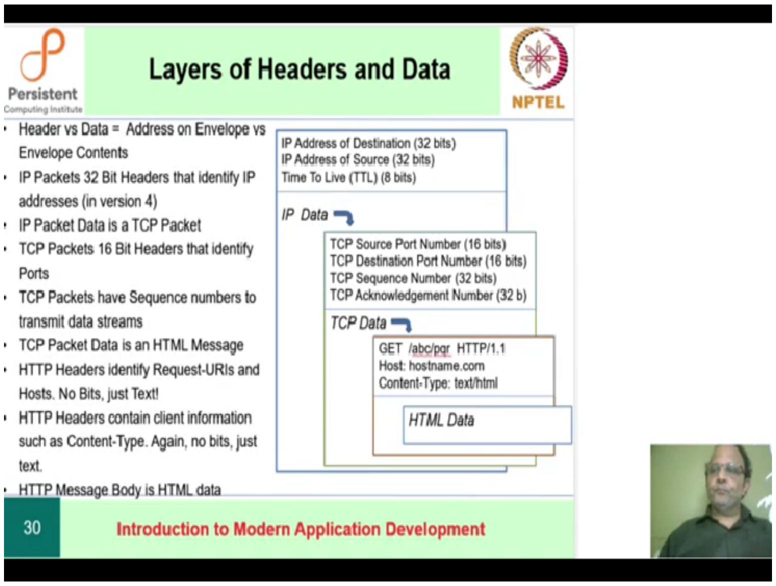
- After DNS helps us find the IP address, TCP's job is to "ring up" the other computer and establish a connection
- Once a TCP connection is setup, we can exchange HTTP messages.
- An HTTP message is the "data" conveyed by the underlying TCP messages.
- In turn, an HTTP message conveys its own data, such as user input via forms and responses from the server such as web page content.

29 Introduction to Modern Application Development

After DNS helps us find the IP address, TCPs job is to ring up the other computer and establish a connection. Once a connection is set up, then we can exchange these HTTP messages over TCP. An HTTP message is the data conveyed by the underlying TCP messages. This is much like having putting some content inside an envelope and mailing that using the postal service. In turn an HTTP message conveys its own data, such as user input from forms, responses from servers and so forth.

And it itself has a similar distinction between the information that is used for the purpose of the protocol versus the information that is used for the purposes of the content. We will see that in a little bit. This is known as the header versus data distinction.

(Refer Slide Time: 21:43)



Layers of Headers and Data

- Header vs Data = Address on Envelope vs Envelope Contents
- IP Packets 32 Bit Headers that identify IP addresses (in version 4)
- IP Packet Data is a TCP Packet
- TCP Packets 16 Bit Headers that identify Ports
- TCP Packets have Sequence numbers to transmit data streams
- TCP Packet Data is an HTML Message
- HTTP Headers identify Request-URIs and Hosts. No Bits, just Text!
- HTTP Headers contain client information such as Content-Type. Again, no bits, just text.
- HTTP Message Body is HTML data

IP Address of Destination (32 bits)
IP Address of Source (32 bits)
Time To Live (TTL) (8 bits)
IP Data
TCP Source Port Number (16 bits)
TCP Destination Port Number (16 bits)
TCP Sequence Number (32 bits)
TCP Acknowledgement Number (32 b)
TCP Data
GET /abc/pqr HTTP/1.1
Host: hostname.com
Content-Type: text/html
HTML Data

30 Introduction to Modern Application Development

When we begin speaking with one another we might say some word like "Hello" with a response such as "Hi". These two words, "Hello" and "Hi", strictly speaking, are fillers; they are not relevant to the communication or the data that is going to happen next, their role is simply to establish that the preconditions for talking about something or make that the relevant amount of interest is there. In the context of Internet communication, this information is known as **header** information. And the actual thing you want to talk about is the data. Another analogy here could be that of address on envelope versus the contents inside the envelope.

Since protocols are layered on top of each other, we call the end result of **protocol stack.** So, our protocol stack has **IP** at the bottom, **TCP** inside IP, and **HTTP** inside TCP, and finally, using HTTP we sent HTML data we want our users to see.

The details look a bit like as shown in the slide given above. For IPv4 at least, this header says that the first 32 bits are the IP address of the destination, which is used by the routing mechanism to take the packet from your computer to the destination computer, or so to speak from the browser computer to the server computer.

It carries the IP address of the source which is the originator of the communication, so that the other side can reply back. There is a third header called “Time To Live”. Tough it is an interesting header, we would not be spending any time describing this header for now.

Inside the packet there is the data part. This data part for the case of TCP, contains a TCP message. In turn the TCP message has a TCP header and a TCP data distinction. In TCP, you have the source port number, you have the destination port number, you have a sequence number. Since TCP is a stream of data, it sends packet in a sequence and this sequence numbers help tell us the order in which packets should arrive at the other end. If packets become out of order TCP is capable of waiting and reassembling your stream in order necessary.

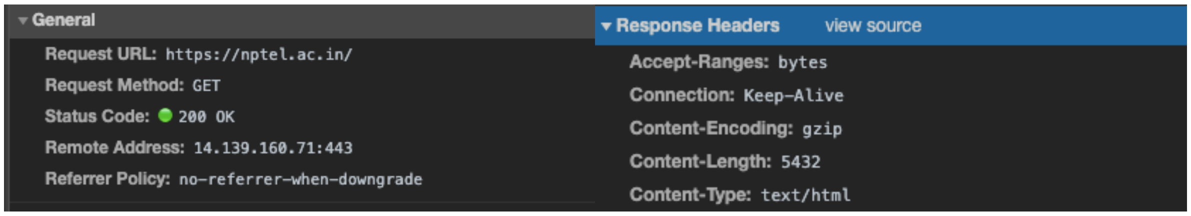
Now, since we are only interested in using TCP and HTTP combination, we will say that, as far as our messages are concerned, the data part of TCP consists of HTTP packets. What is an HTTP packet? It is simply a chunk of information containing header and some data. Now in this header, in contrast with the headers of IP and TCP, you will notice that the earlier two headers are defined in terms of bits. But the HTTP header is defined in terms of text.

So, this switch from bits at the low-level protocols, which are all largely about efficiency versus text at the higher-level protocols is because it is simpler for the developers of the higher-level protocol to develop these kinds of systems rather than complicated systems involving bits and so forth. Not that really prevents us from getting to bits if we really want to, and nowadays people do. But let us leave that aside.

Let us just say that our HTTP packet has this kind of header. Now the word GET here, is exactly like saying “Hello”, at least in the sense of what it achieves. It tells the receiver that the sender wants to get this particular URL using the protocol HTTP1.1. So, when I say GET is like “Hello , what I am saying is, the packet is prefixed with the word GET. And that prefix tells the receiver that this is HTTP packet. That is not the only thing.

You also need to end the first line with this line HTTP 1.1 or 1.0 as the case may be. Just like there are addresses in the IP and TCP packets, there is an address here, which is the hostname.com. And there is a third header I have shown, there are many more headers and we will talk about it in greater detail later, but for now I have shown one more header called content type. So, if the content type is text HTML, then the data part should be HTML.

We show some screenshots of various headers taken from Chrome's DevTools.

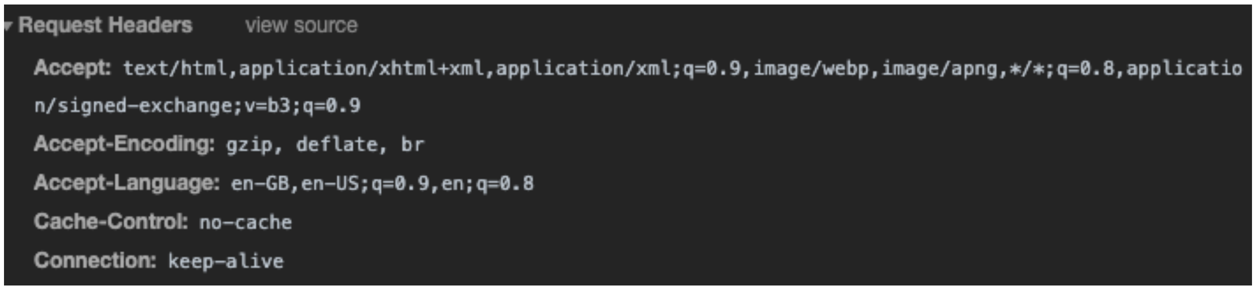


General

Request URL: https://nptel.ac.in/
Request Method: GET
Status Code: ● 200 OK
Remote Address: 14.139.160.71:443
Referrer Policy: no-referrer-when-downgrade

Response Headers view source

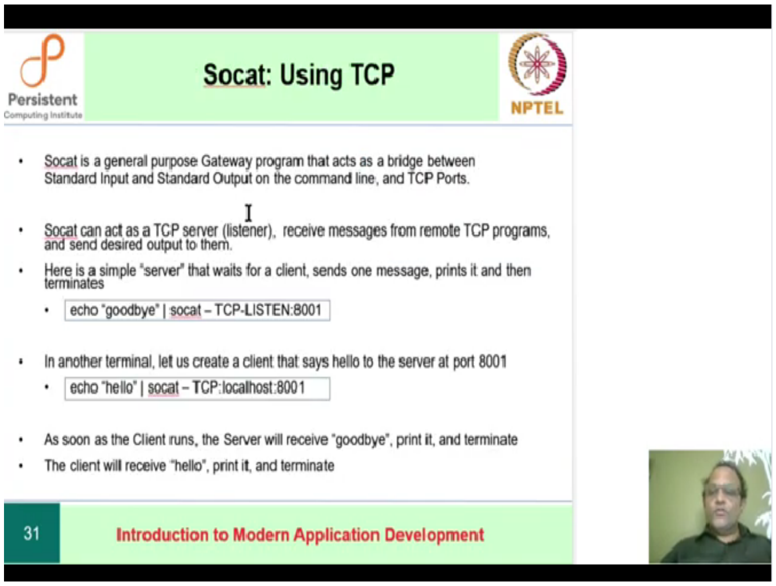
Accept-Ranges: bytes
Connection: Keep-Alive
Content-Encoding: gzip
Content-Length: 5432
Content-Type: text/html



Request Headers view source

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Cache-Control: no-cache
Connection: keep-alive

(Refer Slide Time: 27:42)



Socat: Using TCP

- Socat is a general purpose Gateway program that acts as a bridge between Standard Input and Standard Output on the command line, and TCP Ports.
- Socat can act as a TCP server (listener), receive messages from remote TCP programs, and send desired output to them.
- Here is a simple "server" that waits for a client, sends one message, prints it and then terminates
  - echo "goodbye" | socat – TCP-LISTEN:8001
- In another terminal, let us create a client that says hello to the server at port 8001
  - echo "hello" | socat – TCP:localhost:8001
- As soon as the Client runs, the Server will receive "goodbye", print it, and terminate
- The client will receive "hello", print it, and terminate

31 Introduction to Modern Application Development

Now that we have some idea of what protocols and messages that does not look like, let us use this understanding to get a greater grip on how HTTP works. So, our first step here will be to say that we are not interested in the details of IP and TCP. But we are interested in the details of how HTTP messages are built up using the baseline of TCP and IP. Therefore, we will use a readymade program called ***socat***, which already knows how to speak TCP. And we will use this program to build HTTP clients and servers. Of course, our clients and servers are just enough to illustrate what is going on. So that you have a good grip on what it means for some interaction to use HTTP protocol. So, let us take a look at socat. Socat is a general-purpose gateway program that acts as a bridge between normal machine standard input and standard output and TCP ports.

We will look at these examples and after we are done with these examples, we will see a demo in which you will see how this overall setup works. Socat can act as a TCP server. So, we say that the server has to sit and listen. It listens for requests and whenever there is a request, it responds. It receives messages from remote TCP programs and depending on what the server is designed to do it will further interpret the data that it receives from the TCP programs.

And then will send whatever output we want back to the program which interrogated it. So, let us take a look at some basic examples here.

1. The lines below show you a simple server that waits for a client sends one message prints it and then terminates.

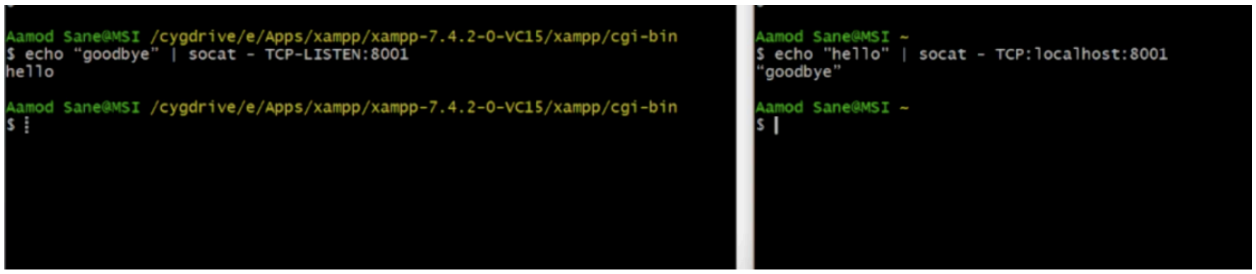
   ```
   $ echo "goodbye" | socat – TCP-LISTEN:8001
   ```

   What is happening here is that the program echo sent the word "`goodbye`" to the input of this particular socat instance. The dash here means to listen to the standard input. And this command `TCP-Listen:8001` says that listen for any TCP communication on port number 8001.
2. Now in another terminal we will create the client which also uses socat, the job of the client is to say hello to the server at port 8001. So here we say echo "hello" and send that as input to socat. This socat is also configured to listen to its standard in. But this time, what it says is talk to TCP on localhost at port 8001.

   ```
   $ echo "hello" | socat – TCP:localhost:8001
   ```

3. As soon as the client runs, the server will receive the message "hello" from the client. And it will terminate because this is just a one line and does nothing else to do once the message is received. In turn the client will receive goodbye from the server because that is what is available to the server on its standard input. And again, because there is nothing more to do, the server will terminate.

In the screenshot given below, we show the client side on the left, and the server side on the right. Notice that we start the server prior to sending the client message for this communication to work. The server must be sitting while waiting for client connection when the client sends a message to the server.



```
Aamod Sane@MSI /cygdrive/e/Apps/xampp/xampp-7.4.2-0-VC15/xampp/cgi-bin
$ echo "goodbye" | socat - TCP-LISTEN:8001
hello

Aamod Sane@MSI /cygdrive/e/Apps/xampp/xampp-7.4.2-0-VC15/xampp/cgi-bin
$
```

```
Aamod Sane@MSI ~
$ echo "hello" | socat - TCP:localhost:8001
"goodbye"

Aamod Sane@MSI ~
$
```

What is happening is that the client is first sending the message, then completing the reception of the message from the server. And then the system together is coming to a halt. The goodbye message came from the server to the client. And the hello message went from the client to the server. Note that this is on a single machine, but it does not have to be wherever if you have access to multiple machines, then by all means, try this out on multiple machines where you can open ports.

And you will find that this program will work just fine. Now that you have seen the basic behavior of socat, going forward we will focus on HTTP, we will take TCP and IP for granted. And in the following we will see how socat can be used to build toy HTTP clients and servers. And later we will study other more built for HTTP tools like curl and firefox web tools. The transition from using simple tools like socat to build HTTP clients and servers will show you that how the layering actually works in practice.

But a normal web programmer will rarely need to use things like socat, except on very special occasions, like proxying and tunneling and so forth. But aside from those unusual TCP related uses the usual tools that we use will be curl and Firefox, like Firefox web tools type things. So, let us take the next step and see what happens.

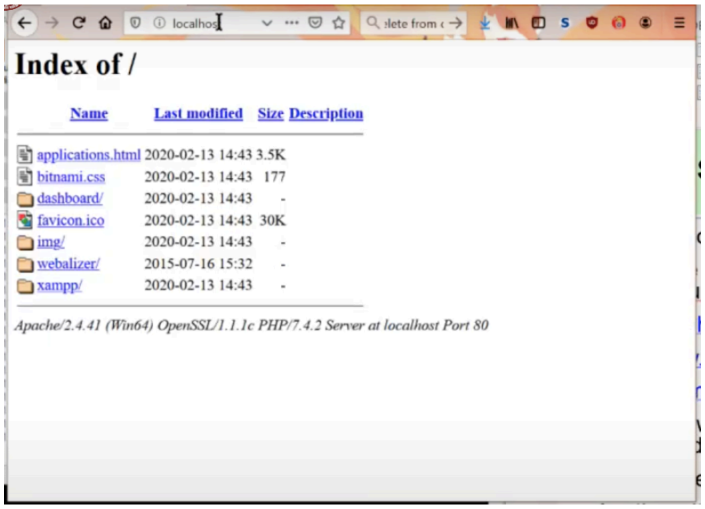
Let us get started with HTTP protocol. The oldest simplest form of HTTP protocol was version 1.0, which had a very straightforward implementation. A client, the browser opens a connection to the server sends its request, and once it gets the reply which it wanted, it terminates the connection. This kind of simple implementation made it very easy to get started with the web. The software was dead simple and long ago when machines were not so powerful and people were not so used to dealing with massive amounts of software, IDEs and great debugging and so on.

It was important that the software be simple. As the need for that the web became more popular, it was then at that point easy to realise that we also want better performance. And so, we should not be establishing connection so often. So, modernization HTTP protocols from version 1.1 and above maintain a single underlying TCP connection, they can exchange multiple HTTP messages over a single connection.

Just as we do not really stop a phone call, let us say after a single sentence, but we will wait for some acknowledgment from our listener, and then continue on to the next. Something similar happens in the case of multiple HTTP messages over a single TCP connection. Here is a technical terminology piece that you should know about. Whenever a single connection is shared for multiple HTTP interactions it is called a ***persistent connection*.**

And remember what I said about starting simple in general, it has been the experience that when you start with a simple system wait for some amount of popularity and then increase the complexity. Such designs turn out to be more fit for survival, let us just say, as different requirements get put on these systems.

We have XAMPP running apache at port 80 and 443. We open up the browser for illustrating what you see when you access the localhost domain. And like I said, browsers for whatever reason has started dropping the HTTP prefix. So, we do not see that anymore. But nonetheless, this is what we see.



Index of /

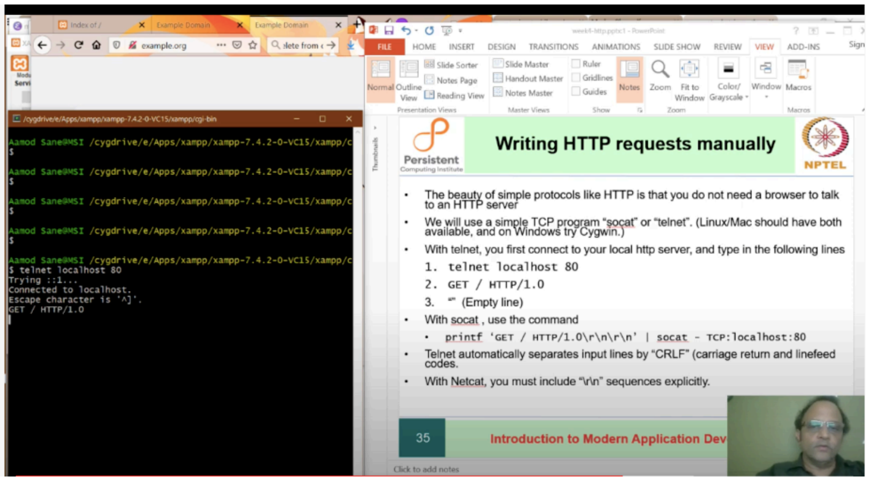
Name	Last modified	Size	Description
applications.html	2020-02-13 14:43	3.5K	
bitnami.css	2020-02-13 14:43	177	
dashboard/	2020-02-13 14:43	-	
favicon.ico	2020-02-13 14:43	30K	
img/	2020-02-13 14:43	-	
webalizer/	2015-07-16 15:32	-	
xampp/	2020-02-13 14:43	-	

Apache/2.4.41 (Win64) OpenSSL/1.1.1c PHP/7.4.2 Server at localhost Port 80

I've made some changes to the default XAMPP setup. There is a file called `index.php` in this directory in default XAMPP, which I just completely removed which resulted in this native behavior of apache, which is to simply show the underlying file system. Similarly, you can see the results when you look at a domain like www.example.org, and the result is exactly the same, which is what we expect.

Now what we are going to do is we will see how to get similar results without an HTTP client of any kind. We will use TCP clients like `telnet` and socat to talk HTTP. Another alternative `netcat` is available. But there are too many incompatible versions of `netcat` on different operating systems. Therefore, we use `socat` which is more predictable across different systems. So, let us write our first HTTP request manually.

(Refer Slide Time: 38:33)



```
Aamod Sane@MSI /cygdrive/e/Apps/xampp/xampp-7.4.2-0-VC15/xampp/c
$

Aamod Sane@MSI /cygdrive/e/Apps/xampp/xampp-7.4.2-0-VC15/xampp/c
$

Aamod Sane@MSI /cygdrive/e/Apps/xampp/xampp-7.4.2-0-VC15/xampp/c
$

Aamod Sane@MSI /cygdrive/e/Apps/xampp/xampp-7.4.2-0-VC15/xampp/c
$

Aamod Sane@MSI /cygdrive/e/Apps/xampp/xampp-7.4.2-0-VC15/xampp/c
$ telnet localhost 80
Trying ::1...
Connected to localhost.
Escape character is '^]'.
GET / HTTP/1.0
```

Writing HTTP requests manually

- The beauty of simple protocols like HTTP is that you do not need a browser to talk to an HTTP server
- We will use a simple TCP program "socat" or "telnet". (Linux/Mac should have both available, and on Windows try Cygwin.)
- With telnet, you first connect to your local http server, and type in the following lines
  1. `telnet localhost 80`
  2. `GET / HTTP/1.0`
  3. "" (Empty line)
- With socat , use the command
  - `printf 'GET / HTTP/1.0\r\n\r\n' | socat - TCP:localhost:80`
- Telnet automatically separates input lines by "CRLF" (carriage return and linefeed codes.
- With Netcat, you must include "\r\n" sequences explicitly.

35 Introduction to Modern Application Dev

The beauty here is that you hardly need any software because the protocol is defined as text protocol. Because the protocol is defined as a text protocol, it is very easy for us to just type in and get the results we want. So, we use the following command to connected to the localhost.

`$telnet localhost 80`

Now we write `GET/HTTP/1.0` followed by an empty line. And now you see a large amount of HTML that the system sends you.

This content is the same as what we see when use the browser to access this URL. Since it is written as So, all the content is the same. It is just that there is a bunch of HTML that visible, which the browser nicely renders into this kind of a list. So that is our first example of HTTP protocol. Now let us try this one, here what we will do is we will use `socat`.

```
(base) HitLap:~ apple$ printf "GET / HTTP/1.0\r\n\r\n" | socat - tcp:localhost:80
HTTP/1.1 200 OK
Date: Tue, 21 Jul 2020 19:07:57 GMT
Server: Apache/2.4.41 (Unix)
Content-Location: index.html.en
Vary: negotiate
TCN: choice
Last-Modified: Thu, 29 Aug 2019 05:05:59 GMT
ETag: "2d-5913a76187bc0"
Accept-Ranges: bytes
Content-Length: 45
Connection: close
Content-Type: text/html

<html><body><h1>It works!</h1></body></html>
```

So, in `printf 'GET / HTTP/1.0\r\n\r\n'`. The first "`\r\n` " terminates the HTTP 1.0 line. The second "`\r\n`" terminates the empty line that is supposed to occur after the first line. And now we have `socat -TCP:localhost:80`. There we go, the same thing. When working with these kinds of systems, it is important to know exactly what data you are going to be sending.

So, now let us see what the difference is between `printf`, `echo`, and `echo - e`. We use the `od` command for this.

1. `printf:`

```
$ printf "GET / HTTP/1.0\r\n\r\n" | od -c
0000000   G   E   T       /       H   T   T   P   /   1   .   0  \r  \n
0000020  \r  \n
0000022
```

2. `echo:`

```
$ echo "GET / HTTP/1.0\r\n\r\n" | od -c
0000000   G   E   T       /       H   T   T   P   /   1   .   0   \   r
0000020   \   n   \   r   \   n  \n
0000027
```

3. `echo - e:`

```
$ echo -e "GET / HTTP/1.0\r\n\r\n" | od -c
0000000   G   E   T       /       H   T   T   P   /   1   .   0  \r  \n
0000020  \r  \n  \n
0000023
```

As we can see that both `echo`, and `echo - e` are adding additional characters. Hence, we use `printf` as it outputs the data as it was given to it; nothing more, nothing less.

Sometimes instead of "`od - c`", which shows you the characters, it is useful to have a `hexdump`. This is sometimes easier to understand them the dumps produced by "od".

With this we have our first HTTP client built using nothing but the underlying TCP systems. So just to clarify, let us take a look at this again. And as we can see, the same kind of thing is showing up here as you can see here, except of course with the browser, it looks a lot prettier. The reader is encouraged to perform a similar analysis for HTTP 1.1. We have shown this in the lecture video.

The main distinction between HTTP 1.0 and HTTP 1.1 is in what lines should be allowed as the header. One normal thing is that the host must be mentioned otherwise it is not a well-formed packet. The processing of HTTP 1.1 is some more strict. So, if you make a mistake in the header command, say you did not include the host line while using HTTP/1.1, you will see is this kind of message, “Your browser or proxy, sent a request that the server could not understand”. So, in this sense HTTP/1.1 is stricter. One other thing we will get to later is that there is another form of error which shows up in the protocol itself. It says HTTP 1.1 400 bad request. This means that you made a mistake somewhere in the request. If we fix the mistake in the request header, we shall get the HTTP response with 200 OK status code.

The last distinction between HTTP/1.0 and HTTP/1.1 that we will like to mention is that persistent connections are taken to be standard. So, we have to add the header `connection:close`. And if we do not, then what happens is that the connection remains open for a while until it timeouts. Again, the reader can see the video lecture at 45:39 for demonstration.

So, in this sense, persistent connections have a timeout at the server end. So that empty connections are not unnecessarily maintained. But the main point is that unlike 1.0 connections explicitly have to be closed either that or they will be closed because of time okay.

Now let us look at the next part. Now we can do something more interesting. We are going to use a socat command to create a server. But this time our server will listen on port 80, 90 or 8001, or whatever it is you want. Because port 80 is already occupied by apache, we have to use some other port. So, let us see what it is like to write server using *socat*.
The server side script is:

```
$ for i in {1..10};
> do
> (printf "HTTP/1.1 200 OK\r\n\r\n";
>  printf "Hello Client: Your request reached server at time $(date)\r\n") | socat -
  TCP->LISTEN:8010 done
```

On the client side (for example, write this in another terminal tab), the command used and the output are shown below.

```
$ echo "Hi Server" | socat - tcp:localhost:8010
Hello Client: Your request reached server at time Wed Jul 22 01:47:49 IST 2020
```

So, on the client side we have `echo "hello" socat - TCP localhost 8010`. Our "hi" went to the server which printed it out, and our server sent back HTTP 1.1 200 OK, followed by this hello client message with a date.

If we used a browser for this request, the headers received by the server we just created will be quite different.

```
HTTP/1.1 200 OK

GET / HTTP/1.1
Host: localhost:8011
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-
Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_3) AppleWebKit/605.1.15 (KHTML, lik
e Gecko) Version/13.0.5 Safari/605.1.15
Accept-Language: en-gb
Accept-Encoding: gzip, deflate
Connection: keep-alive

```

Here, user agent tells you which kind of browser is asking; the bunch of other headers, which we will look at later, and so on.

When we use the browser, there is one more request that happens to get the *fav icon*, which is one of those little icons, like for example, this one, which decorates these, the tabs in the web browsers. And that request gets automatically made. It turns out that there is not of fav icon because we do not serve one. And so next time the request is not repeated. So, as you can see, as far as the browser, as well as our this client is concerned, the server appears to be a valid HTTP server. Now, there are limitations here, of course. One thing is that our server replies with the correct HTTP response format. But it does not verify that the input is an HTTP compatible message.

So, although the browser center correctly configured HTTP message, our other client, the one over here, did not. And that was perfectly fine with this particular server. Nonetheless, you can see what the basic idea is, a server listens to a message and then send some formatted text in response and with not much effort with the aid of a TCP program we have a working HTTP toy server and HTTP toy client as well.

So, the key issue here is to consider the shear simplicity of HTTP. The basic idea is dead simple. On top of TCP, you layer a standard way to send requests and responses. And so, given some way to handle TCP connections themselves, doing HTTP on top is very simple. Let us correct this to HTTP, okay. Web servers like apache and tomcat share the same underlying structure as our toy servers.

They listen to messages in HTTP format, and they respond. Now we are going to look at what each of those lines in the protocol headers actually means. This is part IV, HTTP details and web tools. We will begin by studying HTTP headers followed by a study of curl and Firefox WebDev tools in the next lectures.