Lecture 21 - “the network is the computer”

This was “sun” micro systems moniker about 20 years ago. Today this is increasingly the case. Lots of applications and systems exist only in the “cloud” and are not on a persons computer. Lot’s of computes lack the capability to even have local programs run on them. They are just a platform for running a “client” like Google Chrome. The 2nd larges use of IO in modern computers is via the “net”. The question is how it all works.

Questions:

What is a domain name?

What is a URL?

What are the parts of an URL?

How is DNS resolved?

What is an IP Address?

What is IP v.4 , what is IP v.6?

How will a message from a client make it to a server?

How is can messages be interrupted in the middle?

How do servers work?

What is a proxy?

How do proxies work?

What is a “router” and how do they work?

This is all about answering questions like this. At the low level a server sees an incoming message, deciphers it, if the message is for that system then it acts on it and sends back a response.

This is how most “system” software is built today.

Most languages like Java and Python were designed for a model of read in some data on a single processor, process it, produce some output. That is not a really relevant model for today. First most systems have multiple processors (like 48 * 4 in a single box) and most software is written to run across the web.

Blochain is a good example. The front end is a web application that formats a message that gets sent to the back end - the "server" or "network of nodes" to get processed.

Gmail and Google docs are perfect examples too. A front end has some local data that it is displaying. Then messages are sent to the back end to change and update data (you decide to view an email for example).

This is the big picture - of how messages are sent / received / routed / processed and data is returned to the client.

All of you have used a web browser - that is one kind of client. Lots of other clients exist like "curl" and "wget" it is easy to build custom clients.

Flow - how it all comes together

URL - what is it:

```
http://www.2c-why.com/index.html
                      ^^^^^^^^^^--- "path" to the resouce (communication endpoing)
           ^^^^^^^^^^-------------- domain name
       ^^^------------------------- sub-domain
^^^^------------------------------- Protocal (https, http, sftp etc)
```

This is a default port of 80 for "http" and port 443 for "https". So

```
https://www.2c-why.com:80/index.html
```

is the same address and port.

Domains get resolved into IP addresses. (Or you can use the IP address itself)

Special IP addresses

IP Version 4

Address	Description
127.0.0.1	Local host
192.168.0.xxx	Class C
172.16.xxx.xxx	Class B
10.xxx.xxx.xxx	Class A

IP Version 4 ran out of addresses - so now we have IP version 6.

IP 6 - looks like: 2001:db8:85a3:8d3:1319:8a2e:370:7348

`Localhost` for IP 6 looks like `::1`

So to reference your local system for a server `http://[::1]/`.

Port numbers are from 1 to 65536 on most systems.

www.2c-why.com and 2c-why.com can have different addresses.

Also you can have

test1.2c-why.com and mike.test1.2c-why.com etc. Each with it's own address.

An Average page will have 90 or more domain name lookups. If you can bring you domain lookup local then pages will load much faster.

What happens when somebody "spoofs" a domain lookup - and sends the wrong address to you - then can then intercept the traffic that would normally go to that address.

You can do this too. For example I might want to spoof `a.doubclick.com` - one of the major add serving sites and sent it off to some other address, for example 127.0.0.1 - my local computer. This is how lots of add blocking works.

DNS over HTTPS is designed to prevent this. Using a domain name server that you trust like 1.1.1.1 or 1.1.1.2 (Cloudfont's) or 8.8.8.8 or 8.8.4.4 (google's) is also to prevent this kind of spoofing.

Do you have a laptop - you connect to a wireless network and part of that connection is DHCP - this allows your computer to receive its connection and IP address from the local server - it also gets its DNS setup from that server. Let's say you are in DIA and you connect to AT&T's WiFi hotspot. AT&T can now tell you where to get 'a.doublclick.com' adds from - and instead of sending this to doublclick might want to direct it to it's own add server - and get the revenue from the adds for itself - instead of letting google get the revenue. This actually happens.

Or how about spoofing your bank - This is what https:// is all about - preventing this spoofing. So - your computer has a "signature" that it can check through a chain of trust that the destination is the correct destination for the request - and that via encryption that no body in the middle can look at what was transmitted. This prevents man in the middle attacks - provided that nobody looses the "private" keys that protect the chain of trust. For example the government "Georga" requires that you turn over you private keys to the government. Dell managed to "loose" a set of signing private keys etc.

So DNS turns a "name" into an address.

Then routing occurs.

But most organizations (including in my home) want to intercept your request before it leaves the local site. Both for DNS and for tracking and security reasons. This is the first level of proxy. So your request runs through something like “squid” - and squid can cache pages that you visit - and track you.

Finally you enter the routing and frame cloud. Through some magic with routing tables your packed of data gets directed to the correct computer. Usually on the receiving end it will again pass through a proxy like HA-Proxy - a proxy designed to grantee high availability of the resources. Again it may pass through some caching layer like Cloudfront or a proxy like squid that will cache requests.

Then it will probably reach a reverse-proxy like NgInx - where it gets directed to a final server - some chunk of code that runs on your box or AWS S3 that serves the request.

What happens at the server

http started out serving files. Then the capability for the server to do something replaced some of the files. In our example:

```
http://www.2c-why.com/index.html
```

The “destination” or communication end point is ‘index.html’. If you put in:

```
http://www.2c-why.com/
```

Then the server will usually replace this with `http://www.2c-why.com/index.html` . Some weird servers use something else like `default.htm` - but this is the exception not the rule.

Often ‘index.html’ is a file in some directory on the server. This is how most single page web applications work. So a request for `http://www.2c-why.com/sample/demo1.txt` will probably look in some directory for a sub-directory called `sample` then in that directory for `demo1.txt` . There is no grantee of this behavior - I use a web server that takes `demo.html` and generates the `demo.html` dynamically in code and returns it.

What about ‘https’?

So we could have used `https://www.2c-why.com/` - with the HTTPS protocol. It would default to port 443 - but you can use other ports. This runs on top of a system called “transport layer security" or TLS. Currently TLS is at version 1.3 with 1.0 and 1.1 being deprecated versions that have security flaws that should not be used.

TLS uses encryption to send data back and forth. It also uses a web-of-trust to establish that you are talking to your intended destination. The most common encryption used is ECDSA - the same as the encryption and signatures in the Ethereum and Bitcoin blockchain.

HTTPS uses a table of known good public keys on your system to validate that who you are talking to is the correct destination. If this set of signatures on your system is corrupted - then the validation is useless. So if you have some malicious software gain access to your system and it installs a new "Root Certificate" in your local "Certificate Cache" -then it can spoof anything that is signed by that "Root Certificate". For example as a developer I find it really useful to be able to `https://localhost:9494/index.html` so that I can test code with https locally. This is one of my known test steps before deploying code to a server. Browsers behave differently with 'https' v.s. 'http'. For example the use of cookies for authentication is different. I have to test that before code gets sent to production. To do this I have a way to setup a signing authority (Root Certificate) so that I can sign for the domain `localhost` to be `127.0.0.1` or `[::1]` - on my system. This means that with that root certificate any server that runs on my computer can now used `https` and spoof things like my bank.

How dangerous is that - now suppose that some malicious software did the same thing - fixed a signing authority in the local cache on your system so that it can replace some domain name that it wants to watch. Then it steps in the middle as a proxy -but- it now has the ability to take your browser encrypted HTTPS messages - decrypt them - save them - then re-encrypt the messages and pass them on to the destination. You see the same page that you normally see - but somebody is in the middle catching your login and emptying your bank account.

What about "dynamic" content.

Lots of stuff over the web relies on some dynamic content. When you use gmail and delete an email - this is not deleting a file - this is updating a database at Google. So there is some sort of "API" application programming interface that gets called as the communication end point.

Inside the server there will be some method for routing these communication end points to some chunk of code. Basically this is a table that says for `/api/v1/delete-email` - call the function `HandleDeleteEmail` with the request from the client and a way to send back the results / success/fail to the server.

In Go it looks like:

```
    http.HandleFunc("/status", respHandlerStatus)
```

This is using the standard Go ‘http’ library - it will see the endpoint of ‘/status’ and call the function respHandlerStatus for it.

So if you entered `http://www.2c-why.com/status` you will get back some sort of a message that shows the status of the server.

This means that you can use the http/https system to make calls to a server and do something. In the case of Google and gmail - this “do something” will be to delete the email from that users email.

So when you look at a set of server code and you want to know what messages it will respond to - the first thing to look for is this “routing in table”. Some languages like C# (Vs Code / .NET) go to extremes to “hide” this routing process from the developers. This makes it very difficult to understand where the routing takes place and manage the routing process.

Python and it’s django project are an example of a language that just shows the routing as a table:

```
from django.urls import path

from . import views

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<int:year>/', views.year_archive),
    path('articles/<int:year>/<int:month>/', views.month_archive),
    path('articles/<int:year>/<int:month>/<slug:slug>/', views.article_detail),
]
```

What is a proxy?

A proxy is a program that receives HTTP on one end and send a similar or modified version of the HTTP on the other end. Then when it gets a response it will send back the response to the person that made the request. Proxies are really useful for lots of stuff. They can cache responses . Then can enforce security rules - by reeking that a person login to the proxy before it will send stuff through. They can act as a man-in-the-middle for debugging code so that you can see what is being sent back and forth. Then can monitor load.

There are 2 flavors - a regular “proxy” that a business uses to enforce security and a “reverse proxy” that a server uses to route requests from a common interface to the correct inside server. Some reverse proxies are designed to do the “reverse proxy” and allow for fault tolerance. “HA Proxy” - the High Availability Proxy is an example. Some act as a reverse proxy just to do routing. NgInx is an example of this.

In most web traffic there are multiple proxies the middle.

REST - Representation State Transfer

On top of the - make a request - there is a set of request types that are used. The most common is `GET` that says send me back something - usually a file. All of the proxies and caches in between expect that `GET` is idempotent. That means that for a specific `GET` request they can cache the result for as long as the cache headers specify - and the next time that they see the same request they can just send back the result that they already have. If you are making a request to a chunk of code - and you want that to go all the way to the server - then this caching behavior is not what you want! Usually that requires the addition of a "cache-busting" value to be added so that the URL will be unique every time.

The besides "GET" other commonly used "Methods" are PUT, POST and DELETE. These are usually mapped as follows:

Method	database	Description
GET	select	Retrieve data from the server or database. Read a file or select from a database.
POST	insert	create new data
PUT	update	change some existing data. Often this is an insert/update operation so that if the data is new it will insert.
DELETE	delete	Remove existing data

The other modifier on the data is what format it is in. This is specified using a header. For example it could be text, or HTML, or JSON data that is send back.

The header tells the client how to display the data. You don't display a GIF image in the same way you display a text or html file.

In our example the "method" for display of the data is JSON - JavaScript Object Notion and the "header" for this is: `application/json`.

```
www.Header().Set("Content-Type", "application/json")
```

Parameters passed to the back end

There are a number of schemes for passing parameters from the client to the back end. Name based parameters are the easiest to understand and manager.

Our URL can have them:

```
http://www.2c-why.com/asm?code=Load&name=bob
                                        ^^^------- value
                                   ^^^^----------- name
                              ^^^^---------------- value
                         ^^^^--------------------- name
```

In this example there are 2 names, “code” and “name” and each has a value. It starts with a ‘?’ and is separated with ‘&’. To do this without blanks requires encoding of the parameters - so if you see `%20` that is a blank. There is a standard URL encoding scheme to allow you to encode lots of stuff on the URL.

The second name based scheme is used by POST and DELETE. This is called `x-url-encoded` and sends name based parameters as a part of the “body” of the message instead of the URL.

The Body is limited in length so there is a 3rd name based encoding that is used for files. This 3rd method encodes stuff in the same way that email attachments are encoded.

The forth method is to tell the server that the body of the message is JSON - so that we can send more complex name/value based data from the client to the server.

Another method is using a positional encoding of data in the URL itself. In our Python example :

```
from django.urls import path

from . import views

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<int:year>/', views.year_archive),
    path('articles/<int:year>/<int:month>/', views.month_archive),
    path('articles/<int:year>/<int:month>/<slug:slug>/', views.article_detail),
]
```

The `<int:year>` is telling the pattern matcher that at that location in the URL it needs to match an integer and return it as the named parameter ‘year’. This kind of positional encoding is very popular - but it has the disadvantage of making it much more difficult to analyze the “path” for what functions to call - and also it basically requires that a user of the URL use documentation to identify the lotions of all the parameters. Good research in language development and design has show that positional parameters only work with a small number of parameters and that above that small number it is much better to accept un-ordered name based values.

Sending data back.

So the front end client makes a request. As a developer we have gone and looked in the server or its documentation to find where the requests are processed - and what the end-points are for the communication - and we now know how to send a request. Let's try it with a simple server. We will use a command level client, 'curl' to send the request. We could type in the value in the URL box in the browser also - but in this case we will stick to a simple command line process.

```
$ curl 'http://0x7e3aFEc048bC7be745d0fA0F5af97D3978C40E9A:237801@192.168.0.44:9191/api/
```

This particular server will send back a list of accounts as JSON data.

If we put the same URL into the browser we get a similar result:

```
[
    "0x9d41e5938767466af28865e1c33071f1561d57a8",
    "0x31568cc92115a2ebe6eb37e9a7c7f6334b988196",
    "0x3b65b88e4256c8926358551072f17460efe5452b",
    "0x42f487a6d5c86962310d5ab5afe5cad7bc80805b",
    "0x5ae7b3cf64adc3d7fef099319a9be4acb8bd73ed",
    "0x4af64cd87a47aab7cffdbada6bfd6aef47036c03",
    "0xe7b8a518bf1b5c4f01b2a7ee39a2800a982e06ee",
    "0x7e3afec048bc7be745d0fa0f5af97d3978c40e9a",
    "0x885765a2fcfb72e68d82d656c6411b7d27bacdd7",
    "0xdb180da9a8982c7bb75ca40039f959cb959c62e8",
    "0x40681739b0ef568acce20f5575ad4cf24223926f",
    "0x6e06bf940bb57ade69cb03153d1c3842411bd3c1",
    "0x4e27d9c8ba3a6904f7a7cb31eae5ccce8bf33300",
    "0x0c34a1a3c5ae302cb41f9cfd999e7950b8ebf40f",
    "0xb0d533a8064ed180967aa4dafa453deab107961c"
]
```

In a front end application we may want to display this list in a more user-friendly way - like a HTML table - but the communication is the same. Somebody may have clicked a button in the page to say "Show All My Accounts" - a message was sent - then a response with a list is returned - and the front end then processes this result.

Feeding out the IP address of your computer.

Often your computer will be setup to use DHCP. This means that the address is not fixed. You may be able to use the localhost address - that stays the same. It is 127.0.0.1 - but you may also want to setup your system to use the non-local interface. On Mac and Linux you can use `ifconfig` to get the address. At the commandline:

```
$ ifconfig
```

output should look something like:

```
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
    options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
    inet 127.0.0.1 netmask 0xff000000
    inet6 ::1 prefixlen 128
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    nd6 options=201<PERFORMNUD,DAD>
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    ether f0:18:98:37:71:1f
    inet6 fe80::14c7:3cba:a20c:29e7%en0 prefixlen 64 secured scopeid 0xa
    inet 192.168.0.44 netmask 0xffffff00 broadcast 192.168.0.255
    inet6 fdc9:8c75:ae02::10b8:5dd2:e91:b73f prefixlen 64 autoconf secured
    inet6 fdc9:8c75:ae02::8e0 prefixlen 64 dynamic
    inet6 fdc9:8c75:ae02::9dd1:d612:d4db:1b6b prefixlen 64 deprecated autoconf temporar
    inet6 fdc9:8c75:ae02::f842:b64b:220:48f0 prefixlen 64 deprecated autoconf temporary
    inet6 fdc9:8c75:ae02::19c5:e05c:ad62:42d1 prefixlen 64 deprecated autoconf temporar
    inet6 fdc9:8c75:ae02::6139:73a6:dcba:6f3c prefixlen 64 deprecated autoconf temporar
    inet6 fdc9:8c75:ae02::c117:7f88:6346:eca4 prefixlen 64 deprecated autoconf temporar
    inet6 fdc9:8c75:ae02::8d42:3b5e:639:71be prefixlen 64 deprecated autoconf temporary
    inet6 fdc9:8c75:ae02::4078:1d3e:c848:7341 prefixlen 64 autoconf temporary
    nd6 options=201<PERFORMNUD,DAD>
    media: autoselect
    status: active
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
    ether 02:18:98:37:71:1f
    media: autoselect
    status: inactive
awdl0: flags=8943<UP,BROADCAST,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1484
    ether 7a:12:fb:34:f1:97
    inet6 fe80::7812:fbff:fe34:f197%awdl0 prefixlen 64 scopeid 0xc
    nd6 options=201<PERFORMNUD,DAD>
    media: autoselect
    status: active
```

I know for my system that I am looking in the section ‘en0’ - this will be true for most Mac’s. For Linux look for ‘eth0’ section. Somewhere in there there is a line that starts with ‘inet’ and after that there is an address. It may start with `192.` , `172.` or `10.` that is the address of your local computer on its network.

If you change networks this address will change.

On Windows you can bring up the PowerShell and run:

```
C:\> ipconfig
```

Windows uses a much simpler interface system. Windows fails to support multiple internet listeners on a single system.

Look at the output for the line that starts with “IPv4 Address”. This is the address of your system.

See: https://www.howtogeek.com/168896/10-useful-windows-commands-you-should-know/ for examples.

Other Protocols

HTTP 1.0 and 1.1 ruled for 20+ years.

Then HTTP 2.0 in the last 5 years has taken over. It is a purely https/tls based replacement for HTTP that has compressed headers (saving 10% to 25% of traffic) and batching of request - effectively doubling the throughput to web pages.

These all run on TCP/IP - a reliable communication layer on top of Unix “sockets”.

The latest protocol is QUIC. It skips TCP/IP and runs on UDP - an unreliable lower level communication system. Currently it is used by about 7% of all web traffic. Using QUIC v.s. HTTP 1.1 improves performance of an average page by a 3x factor.

QUIC will replace all the HTTP traffic in a few years. It is just a much better way of transferring data.

Other improvements and changes in the “web”.

1. BR based compression reduces the size of images by 20 to 30% while maintaining the presentation quality.
2. .webm video compression is 30% better than the best .mp4 compression.
3. “Progressive Web Applications” for mobile and desktop are easier to install and have a higher retention/usage rate.
4. WebAssembly - finally a compiled executable format for in the brewer operations. It runs 10x to 50x faster than JavaScript and you get to use other languages.
5. DOH - DNS over HTTP - so that you get security and variation of DNS addresses.
6. 5G - with all the other improvements bandwidth is not that important to mobile apps. 5G will have the biggest impact in Information of Things (IoT). The biggest impediment that Tesla and Goole Waymo have to getting self driving working is lack of data. Right now less than 1% of the data that the cars are collecting can be send back to the cloud. This is because of 4G

congestion. 5G will address this and allow the 30T of data that a Tesla car generates a day to all be pushed back to Tesla.

7. Low Power Processors - the latest ARM processors have 5% more compute power than x86-Intel processors with less than ½ the power consumption. New dedicated processors for AI and Neural networks are 100's of times faster for this kind of processing. The combination will drive all sorts of new and better applications.
8. Blockchain's use in providing reliable tracking, transfer of digital resources and distributed data will drive an new class of applications for years to come.