Skye Nygaard
Luke Parkhurst
Debra Cooperman

We did our final project on creating a world wide web like system for sending and receiving files, as well as resolving meaningful domain names, in order to facilitate the communication between multiple machines.

Client

For the client the main goal was to be able to request a URL, pass in some parameters, and open in a way that displayed the web page accurately to the user. This was primarily accomplished through the combination of using a command-line interface for the initial request, similar to wget and other command line HTTP interfaces, as well as the default browser as an HTML, CSS, and Javascript engine. Because it would have been out of scope to try and create a rendering engine from scratch, it was determined that the best way to get the files to display out to the user was whatever the default browser on the computer was. This is accomplished through the Java line `Desktop.getDesktop().open(new File(filePath))`.

The file transfer is accomplished through TCP and port level programming. In order to get server information on how to establish connection, the DNS client built is used. This acts similar to the real internet, routing to the appropriate location and giving the client program the information it needs to proceed, based on human understandable input. Serving the file is a cooperative effort between the client and the server, where the server must wait with open ports for the client to try and connect. Then the client will receive back the file and write it out. In order to have the asset for the browser to open, the file is first written to disk. This cache location is available to the user through the save=*path* command line argument. It will default to ~/web-cache with no user input, but the user can provide a different location if they want to cache their web pages somewhere else, which could be desirable if you wanted to access the page quicker in the future. This permanent caching allows you to view the website later, even if you are offline.

However, plain HTML is very rarely used in the modern web. There is javascript to add functionality, and CSS to add color and aesthetics. These are not represented in the initial call,

so just like in a real web system subsequent calls are made to get these assets. This is done in the client code through an HTML parser. While a full HTML format parser is not necessary, as all displaying is done through the browser, for all relative links we needed to download content from the server. Because of the emulated nature of our websystem, relative links such as ./folder/file.css doesn't mean anything to the browser. Normally this would be resolved and these related links would be downloaded. In our code, they are downloaded and placed in that relative folder within the web-cache. Regex was used to download files for all ./ in a <link>, a <script>, or <img> to cover all css, javascript and image assets that were included in the page. This resulted in an appearance that is similar to the real application, though due to the async nature of modern websites this is not always perfect. This is the same for any offline compatible system on the modern internet, as many API calls are made constantly while browsing the internet, for data as well as precompiled assets.

The other command-line argument that could be passed was header=language:-s-size:-20000. This would ask for the Spanish version of the website at the link, and the size limits the size of assets, which can prevent overuse of disk space in a simple internet compatible system. It could also be used to just get the top of a page. These arguments are sent along with the URL/link path to the server, and parsed to obtain the appropriate data from the database. This gives the user a greater degree of autonomy, and could be used automatically by a more advanced web system that would develop over time.

Some of the biggest challenges with making the client were related to opening many connections, saving out many files, and making sure as many of the dependant assets were downloaded as possible in order to get the most accurate experience of the web page. The parser is a core component, and it took some trial and error and research in order to get most every important relation in an HTML file. Additionally, getting the nested folders to write out to cache, and opening the file with the dependent assets properly posed some challenges. Creating a parsable and consistent interface to communicate from client to server and back also posed some challenges, as does negotiating any web standard. Ultimately, we felt the interfaces we agreed on provided a good mix of usability, simplicity, and expandability through depth.

Web Server:

The web server is primarily in charge of opening a socket and listening for incoming connection requests from clients. Upon receiving a request, the web server creates a dedicated TCP connection thread for the client to send a request through to the server. The server receives a string get request header encoded in UTF-8 from the client and decodes it before handing it off to a handler.

The handler takes the get request and parses it for information about it. This information includes what language the client wishes the get request to be in, and how much information to send back. The request also holds the URL of the file they are requesting. After parsing the data, the handler holds onto the information in preparation for sending back the file. The URL requested is handed off to a Get Header handler.

The Get Header handler takes the URL request from the web server handler and parses the URL string into smaller bits to pull out the specific file path. It first parses to see if the site visiting is a COM, ORG, or EDU. Then the parser further tokenizes the string and creates another string of the folder path to the file specifically requested. A final variable is saved on the specific file requested, located at the end of the URL token chain. The Get Header handler can then return the folder path, and specific file when needed.

Once the get request is parsed, the web server moves onto the next phase of finding the specific file, packaging it, and then sending it back to the client. It first checks the the server can reach the type of site first, ie. COM, ORG, EDU, etc. If the web server can reach the type of site, it then builds a file path, asking what language was requested, if none then english by default, and then concatenating the folder path and specific file to it. With the file path completed into a string, a new File object is created with the server attempting to reach the specific file. If the file can not be reached, a 404 error integer flag is marked and a 404 error message will be sent back to the client. If the file is found, then a byte array of the full file is sent back to the client. If, however, the request only wants part of a file by dictating the size of the file to send back, then a byte array of requested size is created and as much information of the file that can be sent, will be sent to the client.

Some of the challenges faced making the web server was the original design. The web server went through three versions, the first being a pure http server, the second using an imported Maven Apache library for JSON, then a final implementation of using a basic string encoded in UTF-8. Other challenges included creating a parser that allowed for multiple folder paths to help pull assets for a .html request. A final challenge was ensuring multiple clients

could connect to the web server at the same time. This required some thought process behind the multi threading, and went from trying to use a ThreadExecutioner to creating a server that was created and listened for connections before handing the connect clients off to their own mini web server process.

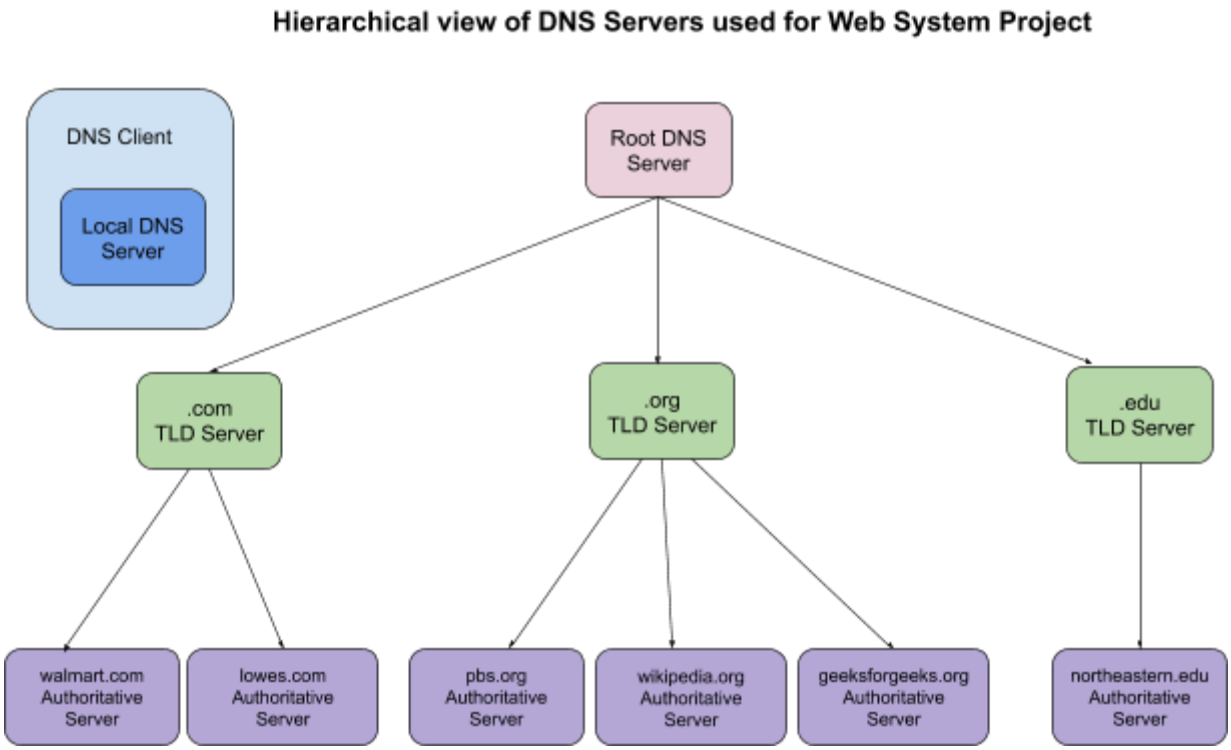
Emulation:

Wanting to try and show how we could simulate a network environment, we opted to create a working topology in GNS3, where 10 virtual machines were used, 2 being masters, 8 being clones. This allowed for a setup of 3 hosts and 1 dns server on one master and 3 clones, while 6 web servers were set up on the other master and 5 clones. This would allow the possibility to show the network working in a simulated environment. Some issues arose though, through the GNS3 virtual machine itself, and the number of virtual machines using a larger amount of RAM on the overall host machine. As a backup and precaution, the team made a project version that worked entirely on local host, in case the GNS3 project portrayal did not work as planned.

DNS

DNS Hierarchy. The DNSServer implementation is ultimately responsible for taking a domain name as input and finding and returning the corresponding IP address, but it is more complicated behind the scenes. To mimic the DNS structure of the overall Internet, we created a hierarchy of DNS servers starting with a Root Server, that in turn had references to 3 top-level domain (TLD) Servers (for .com, .org., and .edu), which in turn had references to Authoritative DNS servers for our 6 domains: www.walmart.com, www.lowes.com, www.geeksforgeeks.org, www.pbs.org, www.wikipedia.org, and www.northeastern.edu. Because we were not able to get the emulation fully working, each of these servers were set up on different ports on the same machine. The browser client then talks to a DNS client, which first checks the cache of its local DNS server. Each DNS Server has a cache containing 0 or more resource records, which contain references to the servers, domains, etc., and include a "time-to-live" property that indicates how long a record can stay in the cache. Each DNS server periodically cleans up its cache and removes expired records. To set up this structure, we created a Registrar class that acts like a registrar/ISP to create and insert the appropriate DNS records into each DNS

Server's cache, and a DNSReceiver class that sets up, registers, and launches the servers to ready them to be queried.



DNS Queries. If a reference to the IP address does not exist in the cache, the local DNS server will communicate with other DNS servers over UDP, constructing a query segment that consists of a 12-byte header that has a query id, flags that indicate whether it is a query or response message, whether recursive or iterative server querying is desired, the number of queries being submitted (limited to 1 for the purposes of this discussion), and a payload that includes the query string itself and the "question type". There are two main question types, one which indicates that the query is looking for an NS (Name server) record, which only includes the name of the authoritative server for a given domain and not the IP address for that server, vs. an A (Authoritative) record, which not only has the name of the authoritative server that houses that IP address but also contains a direct reference to the IP address being asked for. We also included a few other question types that are not formally included as question types in the header. One is a "G" (Glue) record, which are records that in practice are set up ahead of time by an ISP, which provide references to the IP addresses of the nameservers that are owned by

a given ISP, so that if both the domain IP address and the name server IP address are unknown, there is a way to contact the nameserver. We have also created a separate record/query type for TLD servers, which in the case of these references being cleared from the various DNS server caches, are persistently referenced by the Root dns server. As mentioned above, both iterative and recursive queries are supported. An iterative query means that the client executing the original query to a given DNS server only gets back the information that that queried DNS Server directly references. So for example, if the client is looking for "www.pbs.org", and the server being queried only has a reference to a .org TLD server instead of the IP address of the authoritative server, the client will get back only the reference to the .org TLD server, and must issue another query to the .org server to get further information. In recursive querying, if the original DNS Server that was queried doesn't have the answer, it must continue to query other DNS servers until it gets the specific IP address for that domain or sends an error back to the querying client. When a DNS Server receives information back from another DNS Server that is not in its cache, it adds that information to its cache so it can be there for the next query. The response message uses the same DNS Header format as the query message, and returns one or more Resource Records, including an "Error" Resource Record that sends back a "DOMAIN NOT FOUND" message and the domain/query String that was not found. This way the DNS servers can keep track of frequent queries that do not have answers, to help avoid DNS flooding.

DNS Load Balancing. In the real world, DNS Servers can both have multiple IP addresses/servers associated with the same hostname, and multiple hostnames can be mapped to the same IP address (this is accomplished using a CNAME (Canonical Name) record). In the interest of simplicity, we did not address either CNAMEs or MXRecords (for email aliases), but we did want to implement a simple round robin load balancing scheme for the case of multiple IP addresses. Because we were not able to use emulation to scale up the number of web servers and separate IP addresses we could have running at the same time (we have only two IP addresses per authoritative web server), the load balancing is not very useful to the overall program, but our implementation supports the selection of one IP address through a round robin/taking turns strategy in the case where multiple are returned, so that no one IP address (i.e., web server) is used the majority of the time.

Applications:

A web system has multiple applications in the archival of websites and networking retrieval of sites. The web system we created serves as a foundation setting for a possible web rendering engine and finalization for a proper web browser. This can be used to retrieve multiple sites from located web servers, and cached or simply viewed. Because everything has to be offline compatible, it is a much better system for archiving, since the version of the website received will always be constant. The system also can be formatted for additional changes that developers may prefer, such as but not including, allowing the hiding of ip addresses, accessing sites not allowed in certain countries, or circumventing troublesome ads.

There is a very real threat of the traditional internet being censored and things being shut down, so alternative methods for delivering content through a centralized traditional architecture that is easily extensible can provide great value. By simply adding a folder and running the server, a client is able to connect, and as long as the content sticks to traditional web proccessible formats, it will be able to be displayed out through the existing rendering technology. A system like ours provides a robust, extensible, and consistent way of viewing content across networks.

Although in a traditional web system DNS is used for discovering new computers on the Internet, current research in DNS applications points to new uses for service discovery, in containerized environments, and as a way to provide unique identifiers for any device.