Lecture 3: The Domain Name System and Sockets

Anirudh Sivaraman

2020/10/11

In this lecture, we'll discuss three topics.

1. How hosts are identified on the Internet by computers (IP addresses) and by humans (domain names).
2. The Domain Name System (DNS), a service that maps domain names to IP addresses.
3. Sockets, the application programming interface (API) that programs use in order to communicate with other programs on the Internet. In the Sockets API, programs can identify other programs to communicate with using either domain names or IP addresses.

1 Identifying hosts on the Internet

1.1 IP addresses

An IP (Internet Protocol) address is an address that identifies a host or router on the Internet. In version 4 of the Internet Protocol, these addresses are 32-bit numbers. Version 6 modifies these addresses to be 128-bit numbers, but we'll only be dealing with version 4 addresses (also known as IPv4 addresses) in this course.

IP addresses can either be public (globally accessible) or private (locally accessible). A public IP address is typically used for a server that needs to be at a known IP address so that users/clients know where to connect to. An example is the server for www.nyu.edu. A public IP address is an IP address that routers on the global Internet understand and can route packets to (i.e., they know of a path that will get packets to a public IP address). A private IP address, on the other hand, makes sense only within the context of a private network, such as a campus network, a dorm network, or a home network. You cannot route to a private IP address unless you're on the same private network as that IP address. For instance, it is hard for you to run a server behind your home WiFi router and expect that clients will be able to send packets to it.

As an example of a private IP address, the IP address on my laptop when I am at home is 192.168.1.153. This address only makes sense in the context of my own home network, which is made up of my WiFi router and any other computers connected to my WiFi router. In my neighbor's house, my neighbor's laptop could also have the IP address 192.168.1.153, connected to my neighbor's router. Private IP addresses provide a way to reuse the same IP addresses across different independent networks, which was important as the number of hosts on the Internet grew.[1]

In many cases, computers on a private network may not ever need to communicate with the global Internet. For instance, you could imagine a group of machines within a cluster that communicate with each other to process a large amount of data, but never communicate with the global Internet. Many supercomputing clusters work this way. But occasionally, a host on a private network needs to communicate with the global Internet, e.g., when your laptop at home wants to talk to Google's server, which has a public IP address. For such purposes, a device called a *network-address translator* (NAT) maps one or more private IP addresses to a single public IP address for the purpose of communicating with the outside world.

A NAT's ability to map one or more private IP addresses to a single public IP significantly reduces the need for new public IP addresses. For instance, a home router could have a single public IP address that would allow

[1] Private IP addresses were introduced as a solution to the rapid exhaustion of IPv4 addresses before IPv6, with its larger addresses, was designed.

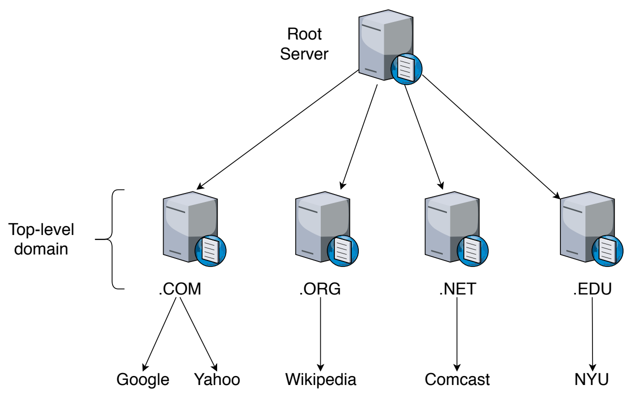

Figure 1: Hierarchy of DNS servers

it to communicate to the outside world on behalf of several clients with private IP addresses connected to that router. You can find out your host's router's public IP address by typing "IP address" into the Google search toolbar. Private IP addresses are like room numbers in a building. The room numbers only make sense within a building, but the building has a publicly advertised postal address for the entire building.

Private IP addresses are a good example of the classic computer systems trick of hierarchy. Private and public IP addresses form a two-level hierarchy: the first level consists of public IP addresses, while the second consists of several private IP addresses under each public IP address. When we get to the section on routing and forwarding, we'll see how IP addresses themselves are hierarchical and how this significantly improves the scalability of the routing problem, i.e., getting packets from one address to another.

The allotment of public IP address is handled by the Internet Assigned Numbers Authority (IANA), while private IP addresses can be self-assigned so long as they are within specific IP address ranges that are reserved for private IP addresses. The 127.0.0.1 address we see on all our computers is a good example of a private IP address.

1.2 Domain names

IP addresses are hard to remember. Imagine having to type https://172.34.56.123 every time you wanted to access a web site. Ideally, we would have a more memorable way to remember the web site. That's where the domain name comes in. An example of a domain name is www.cs.nyu.edu. Domain names are hierarchical: www.cs.nyu.edu belongs to the edu top-level domain (TLD), the nyu subdomain within the edu TLD, the cs subdomain within the nyu subdomain, and the www subdomain within the cs subdomain.

2 DNS: From domain names to IP addresses

Given a domain name, how do we retrieve its IP address? What's the dumbest way to do this? You could maintain a file that mapped every domain name to its IP address. As naive as this sounds, this is how the Internet worked before 1984. There was a single file called HOSTS.TXT that maintained this mapping and which people swapped with each other over the Internet. To add a new host, the host's owner had to call an operator at Stanford Research Institute! If you don't believe me, open up /etc/hosts on your UNIX/Mac machines. This is a remnant from the HOSTS.TXT era.

Clearly, at some point this was going to be unsustainable as the Internet grew in size. DNS evolved as an automated and decentralized solution to this problem. You can think of DNS as a global service that allows users anywhere to map from domain names to IP address. In fact, this mapping need not even be unique: www.google.com maps to a different public IP address, and hence physical server, depending on where you are accessing it from.

So, how does DNS work? DNS has a hierarchy of servers that mirrors the hierarchy found in domain names themselves (Figure **??**). At the top of the hierarchy are the root servers. One level below are the TLD servers, one for each TLD. Below this are the authoritative servers, typically one for each organization such as NYU, Google, or Facebook. In addition, there are also local servers owned and operated by service providers that are provided as a convenience to the end users.

When a user wants to retrieve the IP address for a host name, the user typically contacts the local server, whose IP address is available as part of the host's configuration (on my MacBook, this configuration is stored in /etc/resolv.conf). If the local server has the host name to IP address mapping available, it returns it right away. Otherwise, the local server does one of two things. It can punt on the problem and delegate the task of finding the IP address to another DNS server. It does this by returning the IP address of some other DNS server so that the user can contact that DNS server instead. Alternatively, the local server undertakes the task of finding the IP address on the client's behalf by contacting other DNS servers on its own.

The first case is called iterative DNS because every time the user contacts a DNS server, the server either returns the IP address corresponding to the domain name or the address of another DNS server. The client then iterates through these servers to get the IP address. The second case is called recursive DNS, because each DNS server contacted in the process of a looking up a domain name recursively takes on the responsbility of looking up the domain name.

In the worst case, this process can incur substantial latency. For instance, if the local server does not have the IP address available, the following sequence of events takes place. The local server first contacts the root of the hierarchy (the root servers), which then either directly contact or delegate the work to the TLD servers, which in turn might contact or delegate the work to the authoritative servers. Regardless of whether it was iterative or recursive, a name lookup would incur 8 messages worth of latency (two for each of the three levels of the hierarchy and two for the local server).

Instead, servers cache frequently looked up names, so that the higher levels of the hierarchy are contacted less frequently. This has two benefits: it reduces the latency of the lookup and the load on the higher levels of the hierarchy, which need to serve many clients by virtue of being higher up in the hierarchy. This is similar to the cache hierarchies found in processors: you ideally want data to be present in the lower levels of the cache to minimize the number of accesses to main memory and reduce the data access latency.

The task of creating new top-level domains is handled by the Internet Corporation for Assigned Names and Numbers (ICANN). ICANN decides who operates the root servers. ICANN also decides which organizations control the allocation of domain names in each TLD. These organizations are called *registries*. Registries are in charge of managing the allotment of subdomain names within specific TLDs to *registrars*. In turn, registrars can sell subsubdomain names to web site owners. Verisign is a registry in charge of the .com and .net TLDs,[2] while GoDaddy is a registrar that buys domain names from Verisign and sells them to web site owners. Domain names can range in cost from a few 10$ to a few million $ depending on how many casual web surfers are likely to stumble on the domain name.

One natural question is: why have two kinds of identifiers in the form of domain names and IP addresses? The case for domain names is clear: it is hard to remember IP addresses. The case for an IP address is that routers can't easily deal with variable-length English-readable names when looking up the destination identifier of a packet for packet forwarding. We'll look at packet forwarding later in the course when we discuss routing protocols.

3 The Sockets API

Now that we know how hosts on the Internet are identified by humans (domain names) and computers (IP addresses), we'll discuss how programs running on two hosts can communicate with each other. The predominant means for programs to interact with the network is an application programming interface (API) called Sockets.[3]

[2]Verisign also operates 2 out of 13 root servers.

[3]More precisely, it's called Berkeley Sockets because it originated in the Berkeley Software Distribution variant of UNIX in 1983. This API is another example of designing for generality; it continues to be used for unanticipated use cases, more than 30 years later.

Like other APIs, the Sockets API takes the form of several functions that can be called by a program wishing to communicate with another program. This API is available in many languages. For concreteness, we'll use Python, but equivalent APIs exist in other languages as well.

We'll discuss the main functions in the Python 3 Sockets API during the lecture. `https://docs.python.org/3/library/socket.html` is the authoritative reference for the Python 3 Sockets API and will come in handy when working on Assignment 1. We'll start with UDP (also known as datagram) sockets and then move on to TCP sockets.

3.1 UDP sockets

The first function is `socket()`, which creates an object called a socket. A socket is a number that a program can use to communicate with another program. In UNIX terms, it is no different from a file descriptor, which is a number that is used for reading or writing from an open file. Instead, with a socket, a program is reading (receiving) or writing (sending) from or to the network.

To create a socket, open a python shell and type the following:

```
>>> from socket import *
>>> sock_object=socket(AF_INET, SOCK_DGRAM)
```

This creates a socket object `sock_object`, which is an Internet Protocol Version 4 (IPv4) socket (AF_INET) of the datagram variety (SOCK_DGRAM). These two arguments are required because sockets are a general purpose communication mechanism beyond just communication on the Internet. For instance, Unix domain sockets allow two processes on a Unix machine to communicate with each other. For this course, we'll only worry about IPv4 sockets.

OK. Now, how do we send using this socket? First, where do we send to? Let's create a UDP receiver to send data to. For this, we'll use the `nc` utility, which is a testing utility to send and receive data on TCP and UDP sockets. Open up a separate terminal, and type:

```
nc -l 8000 -u
```

This tells nc to create a UDP receiver (the argument -u) that is expecting data on port 8000. Now, let's use `sock_object` to send data to nc. Go back to the python shell and type:

```
>>> sock_object.sendto(b"hello", ("127.0.0.1", 8000))
```

You should now see a hello appear at the terminal running nc. What are the two arguments here? The first argument in sendto is the data you want to send to the other end. The second argument identifies the other end using an IP address (you can also use a domain name instead) and port pair. The port is 8000 because that's where we setup the nc receiver. The IP address is 127.0.0.1 because that is a reserved private IP address to refer to your own local machine.[4]

OK. Let's look at how we receive data on a socket. We'll do that by replacing the nc program with our own Python program that listens on port 8000 and receives data on it. Terminate the nc program, start a new Python shell, and type:

```
>>> sock_receiver = socket(AF_INET, SOCK_DGRAM)
>>> sock_receiver.bind(("127.0.0.1", 8000))
>>> sock_receiver.recv(4096)
```

The first statement is identical to before. The second demonstrates the `bind` function, which attaches the socket to a particular port and IP address. This is typically done for servers that need to be at a well-known port and address so that clients (end hosts) know where to contact them. The third statement demonstrates how a socket object receives data. In this case, we are telling the socket receiver to receive up to 4096 bytes

[4]The b prefix before hello is a Python detail because `sendto` expects a byte array as an argument. The b prefix converts a string to a byte array.

of data. The `recv` function *blocks*, i.e., the Python shell is occupied by this function alone, until some data is received by this socket. The return value from this function is the number of bytes actually received.

To receive some data using `sock_receiver`, go back to the Python terminal with `sock_object` and retype the sendto statement.

```
>>> sock_object.sendto(b"hello", ("127.0.0.1", 8000))
```

You should see some data being received at `sock_receiver`. The return value should tell you how much data you received.

Now, one problem with `recv` is that it is blocking, meaning that until some data is actually received the function is held up. This is an issue if you're expecting to receive data from two different sockets. One way around this is to make the `sock_receiver` object non-blocking so that the recv() call returns immediately if there is no data:

```
>>> sock_receiver.setblocking(False)
```

Now, if you try the recv() call again and don't have data readily available, the function will return right away,[5] and you can proceed to check on another socket if one exists. So if you wanted your application to continuously receive data on 2 sockets, you would write code like this.

```
>>> s1.setblocking(False)
>>> s2.setblocking(False)
>>> while(True):
...   ret1 = s1.recv()
...   # do something if ret1 is not zero, i.e., at least one byte was received
...   ret2 = s2.recv()
...   # do something if ret2 is not zero.
```

The problem with this is that the application is continuously executing recv() in the off chance that some data is available. However, because data is only sporadically sent to the application (typically because of a user interaction), the processor is mostly wasting CPU cycles executing the recv() function and checking the return of the recv() call. These CPU cycles could be put to better use by running other processes in the system.

An alternative solution is to use the `select` function call, which allows an application to simultaneously listen for data on multiple sockets. The `select` function call is blocking in that it will prevent further processing in the same application[6] until at least one of the sockets has something to read.

Importantly, `select` doesn't take up CPU cycles, and the OS can schedule other processes to run until the application that called `select` has something to read. There is a lot going on under the hood to make `select` happen, but informally, this is implemented by having the Network Interface Card wake up (or interrupt) the OS when it has received some data. The OS in turn wakes up and schedules the process calling `select` if the data belongs to that process. As is usually the case with computer systems, the easier and more pleasant an interface is to a programmer, the harder you have to work underneath to faithfully present this interface. `select` is a good example of this.

OK, how do we use the `select` function in Python? Let's say you have two sockets `s1` and `s2` that you want to simultaneously receive from.

```
from select import *
r = select.select([s1.fileno(), s2.fileno()], [], [], 1.0)
# do something if r is not none.
```

[5] In the Python API, it throws a BlockingIOError exception, which you can catch.

[6] Strictly speaking in the same application thread, but we'll assume single-threaded applications in this course because multi-threaded network applications pose their own unique headaches.

The arguments to select are three lists and an optional timeout parameter. The three lists represent sockets that are we waiting on for reading data, for writing data, and for exception conditions. For our purposes, we'll mostly leave the second and third lists empty. The fourth, optional, argument is the timeout that specifies how long in seconds the select function should block before it returns anyway. The timeout mechanism provides a convenient way to implement timers. The return value from select is a triple of lists, where each list indicates a subset of the read, write, and exception sockets that are actually ready for being read, written, or checked for exceptions.

One thing to note here is that the select function takes a list of file descriptors, and not sockets. This conversion is performed by the function fileno. This is because the select function can be used to wait on certain kinds of file descriptors as well, and sockets are really just file descriptors as far as the OS is concerned. This is a good illustration of the underappreciated power of modularity. By designing the select function call to take any generic file descriptor, a programmer can reuse the same select call for both real files and "virtual files," such as sockets, without having to invent a new API just for sockets alone.

3.2 TCP sockets

The procedure for TCP sockets shares many similarities with the procedure for UDP sockets, but there are a few key differences that stem from the fact that TCP is a reliable, in-order, byte-oriented abstraction, as opposed to an unreliable packet/datagram-oriented abstraction. The first difference is relatively mundane: TCP sockets are created using the SOCK_STREAM socket type.

```
>> tcp_socket = socket(AF_INET, SOCK_STREAM)
```

The next difference is more substantial. Because TCP sockets provide a reliable, in-order stream of bytes from the sender to the receiver, we need a way to synchronize the sender and the receiver, so that they can both agree on where the first byte in the byte stream starts. This synchornization requirement manifests itself as a few changes in the API for TCP sockets relative to UDP sockets.

First, a TCP server that has been bound to a particular port and IP address (similar to our UDP server earlier) needs to be put into a LISTEN mode that allows it to synchronize with a client. This is distinct from the ESTABLISHED mode where it receives an in-order bytestream containing application data from the client. You can put the server in LISTEN mode using the function listen, which is called after the bind function call.

```
>>> from socket import *
>>> tcp_server = socket(AF_INET, SOCK_STREAM)
>>> tcp_server.bind(("127.0.0.1", 8000))
>>> tcp_server.listen()
```

Now, on the client side, we need to use a new function call connect to synchronize the client with the server so that they both agree on what the first byte in the reliable, in-order stream of bytes is. The connect function needs to know what the client is trying to synchronize to, so it takes the server's address and port as arguments. The connect function is blocking: it waits until the client has synchronized.

```
>>> tcp_client = socket(AF_INET, SOCK_STREAM)
>>> tcp_client.connect(("127.0.0.1", 8000))
```

Now, the listen function call is non-blocking. It only puts the server in LISTEN mode, it doesn't actually synchronize the server to a client just as yet. To accomplish the server side of the synchronization (i.e., the server's version of the client's connect), the server uses the blocking function accept, which returns two things after a successful synchronization attempt: (1) the address of the client the server just synchronized with, and (2) a new socket that the server can use to send/receive application data to/from this client.

```
>>> (comm_socket, client_addr) = tcp_server.accept()
```

This new socket object (`comm_socket`) is distinct from the `tcp_server` socket object. This is to allow the server to accept multiple connections from different clients using the same `tcp_server` object, which is always bound to the same address and port. The return value from each accept() call results in a new socket object that the server can use to communicate with the just accepted client.

The alternative of reusing the same `tcp_server` socket for communicating application data as well would have likely resulted in a simpler API. But it creates a new problem. For instance, to view www.cs.nyu.edu, you connect to the IP address represented by www.cs.nyu.edu at the designated port 80 for HTTP connections. Let's say you start browsing www.cs.nyu.edu as the first user to that web page. Now, no other user can connect on port 80 because the port is tied up with sending and receiving data from you. The next user would have to connect on some other port, but it's not clear which other port to use. Using two separate sockets allows the bound server socket to continue accepting new connections on the same publicly advertised port.

Once the communicating socket (`comm_socket`) has been created, communication between the client and server proceeds similar to UDP. One difference at the API level is that the TCP client can use the function `send()` instead of `sendto` because once the synchronization has completed and the connection has been established, the client socket knows the address of the server.

```
>>> tcp_client.send(b"hello")
```

Another difference is the fact that TCP is a byte-oriented abstraction. So if you issue two `sends` at the client, but a single `recv` at the server, the server will read out the sum total of data sent by both send calls. In other words, TCP only guarantees a bytestream and it is not obliged to preserve packet boundaries. For UDP, on the other hand, you would need two reads() to read out each of the messages that the client sent using the send() method calls. This is because UDP is a datagram-oriented abstraction that does preserve packet (datagram) boundaries.

Finally, although we have presented all communication as one directional (from the client to the server), we should mention that the Sockets API is bidirectional. The server can send data to the client using the same `send()/sendto()` calls that we demonstrated for the client above.