Chapter 2

Application Layer Protocols

After introducing the application layer in the previous chapter, we discuss some standard application-layer protocols in this chapter. During the lifetime of the Internet, several client-server application programs have been developed. We do not have to redefine them, but we need to understand what they do. For each application, we also need to know the options available to us. The study of these applications and the ways they provide different services can help us to create customized applications in the future.

2.1 Protocol

A protocol is a standard set of rules that allow electronic devices to communicate with each other. These rules include what type of data may be transmitted, what commands are used to send and receive data, and how data transfers are confirmed.

You can think of a protocol as a spoken language. Each language has its own rules and vocabulary. If two people share the same language, they can communicate effectively. Similarly, if two hardware devices support the same protocol, they can communicate with each other, regardless of the manufacturer or type of device. For example, an Apple iPhone can send an email to an Android device using a standard mail protocol. A Windows-based PC can load a webpage from a Unix-based web server using a standard web protocol.

Protocols exist for several different applications. Examples include wired networking (e.g., Ethernet), wireless networking (e.g., 802.11ac), and Internet communication (e.g., IP). The Internet protocol suite, which is used for transmitting data over the Internet, contains dozens of protocols. These protocols may be broken up into four categories:

1. **Link layer** - PPP, DSL, Wi-Fi, etc.
2. **Internet layer** - IPv4, IPv6, etc.
3. **Transport layer** - TCP, UDP, etc.
4. **Application layer** - HTTP, IMAP, FTP, etc.

Link layer protocols establish communication between devices at a hardware level. In order to transmit data from one device to another, each device's hardware must support the same

link layer protocol. Internet layer protocols are used to initiate data transfers and route them over the Internet. Transport layer protocols define how packets are sent, received, and confirmed. Application layer protocols contain commands for specific applications. For example, a web browser uses HTTPS to securely download the contents of a webpage from a web server. An email client uses SMTP to send email messages through a mail server.

Protocols are a fundamental aspect of digital communication. In most cases, protocols operate in the background, so it is not necessary for typical users to know how each protocol works. Still, it may be helpful to familiarize yourself with some common protocols so you can better understand settings in software programs, such as web browsers and email clients.

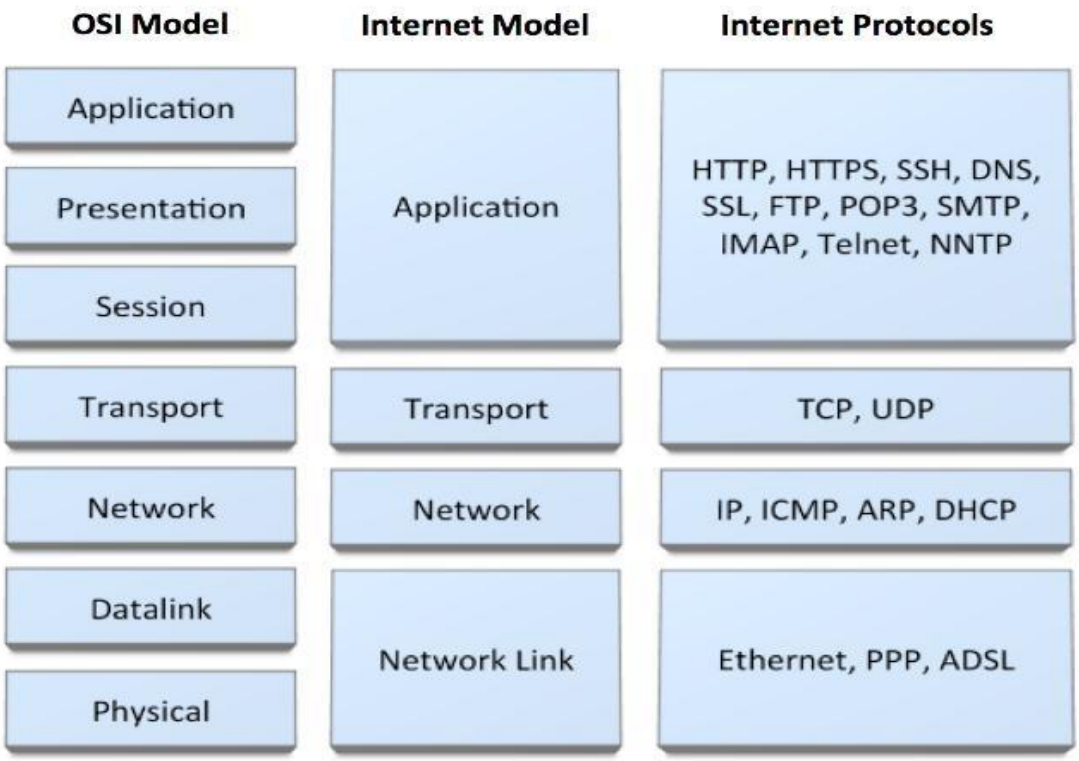

Figure 2.1 Different Layers Protocol

2.2 Protocol Types

You are developing your discrete events model, and you see your agents moving from one block to the next. Simple right? But have you stopped for a moment to ask yourself why they are even moving? What is that thing that makes them move?

If you ask a beginner, the answer will come from pure intuition: “Well, if the agent is in a delay block and the delay time is over, the agent will just know that it has to move to the next block”. But that answer is in fact wrong. The agent does nothing. The agent is in fact just an unanimated stone that is pushed and pulled around the process flow by the blocks themselves.

How does this happen? Well, first let’s acknowledge the fact that the blocks that you find in the Process Modeling Library are nothing but agents in disguise. You have a delay agent, a queue agent, etc. And as agents do, the blocks communicate with each other. When you connect two blocks in a discrete events model, you are basically generating a communication gateway for these blocks.

There are two types of protocols. Push and Pull. In the default AnyLogic world, all the blocks use the pull protocol, with the exception of the blocks that are able to generate agents (for instance the source and enter blocks), which use the push protocol. The protocol applies to the block that is attempting to send the agent to the next block.

Pull Protocol: the block will notify the receiving block that there’s an agent ready to exit. If the receiving block is able to receive that agent, then the agent is sent. If not, then the agent stays where it is and the receiving block will try to get it again in the future when it’s able to receive something.

Push Protocol: the block will be pushed to the receiving block without any request. This produces errors. And these are the errors that we first discover when we start using AnyLogic for instance when there’s no available space in a queue that exists after a source block. Yeah, because the source block uses push protocol as a default.

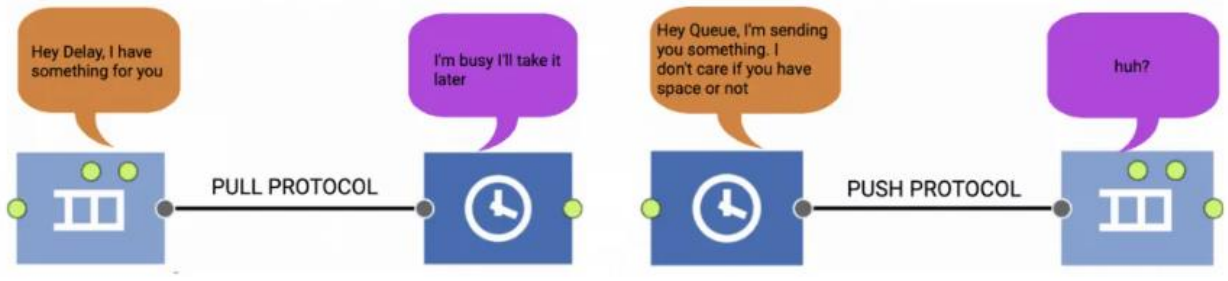

Figure 2.2 Pull and Push Protocol

So the question now is… when to use pull or push? And there's no strict answer for that. Since the push protocol gives an error, you may use it to make it evident when there's a bottleneck somewhere in your process. Or maybe you want to use a pull protocol in the source block to destroy agents that cannot be received in a queue. In general you will not need to change the defaults, but it's always good to know why things are happening, right?

2.3 HTTP

The **HyperText Transfer Protocol (HTTP)** is used to define how the client-server programs can be written to retrieve web pages from the Web. An HTTP client sends a request; an HTTP server returns a response. The server uses the port number 80; the client uses a temporary port number. HTTP uses the services of TCP, which, as discussed before, is a connection-oriented and reliable protocol. This means that, before any transaction between the client and the server can take place, a connection needs to be established between them. After the transaction, the connection should be terminated. The client and server, however, do not need to worry about errors in messages exchanged or loss of any message, because the TCP is reliable.

Nonpersistent versus Persistent Connections

As we discussed in the previous section, the hypertext concept embedded in web page documents may require several requests and responses. If the web pages, objects to be retrieved, are located on different servers, we do not have any other choice than to create a new TCP connection for retrieving each object. However, if some of the objects are located on the same server, we have two choices: to retrieve each object using a new TCP connection or to make a TCP connection and retrieve them all. The first method is referred to as a *nonpersistent connection*, the second as a *persistent connection*. HTTP, prior to version 1.1, specified *nonpersistent* connections, while *persistent* connections are the default in version 1.1, but it can be changed by the user.

Nonpersistent Connections

In a **nonpersistent connection**, one TCP connection is made for each request/response. The following lists the steps in this strategy:

1. The client opens a TCP connection and sends a request.
2. The server sends the response and closes the connection.
3. The client reads the data until it encounters an end-of-file marker; it then closes the connection.

In this strategy, if a file contains links to N different pictures in different files (all located on the same server), the connection must be opened and closed $N + 1$ times. The nonpersistent strategy imposes high overhead on the server because the server needs $N + 1$ different buffers each time a connection is opened.

Persistent Connections

HTTP version 1.1 specifies a **persistent connection** by default. In a persistent connection, the server leaves the connection open for more requests after sending a response.

The server can close the connection at the request of a client or if a time-out has been reached. The sender usually sends the length of the data with each response. However, there are some occasions when the sender does not know the length of the data. This is the case when a document is created dynamically or actively. In these cases, the server informs the client that the length is not known and closes the connection after sending the data so the client knows that the end of the data has been reached. Time and resources are saved using persistent connections. Only one set of buffers and variables needs to be set for the connection at each site. The round trip time for connection establishment and connection termination is saved.

2.4 Message Format

The HTTP protocol defines the format of the request and response messages, as shown in Figure 2.3. We have put the two formats next to each other for comparison. Each message is made of four sections. The first section in the request message is called the *request line;* the first section in the response message is called the *status line*. The other three sections have the same names in the request and response messages. However, the similarities between these sections are only in the names; they may have different contents.

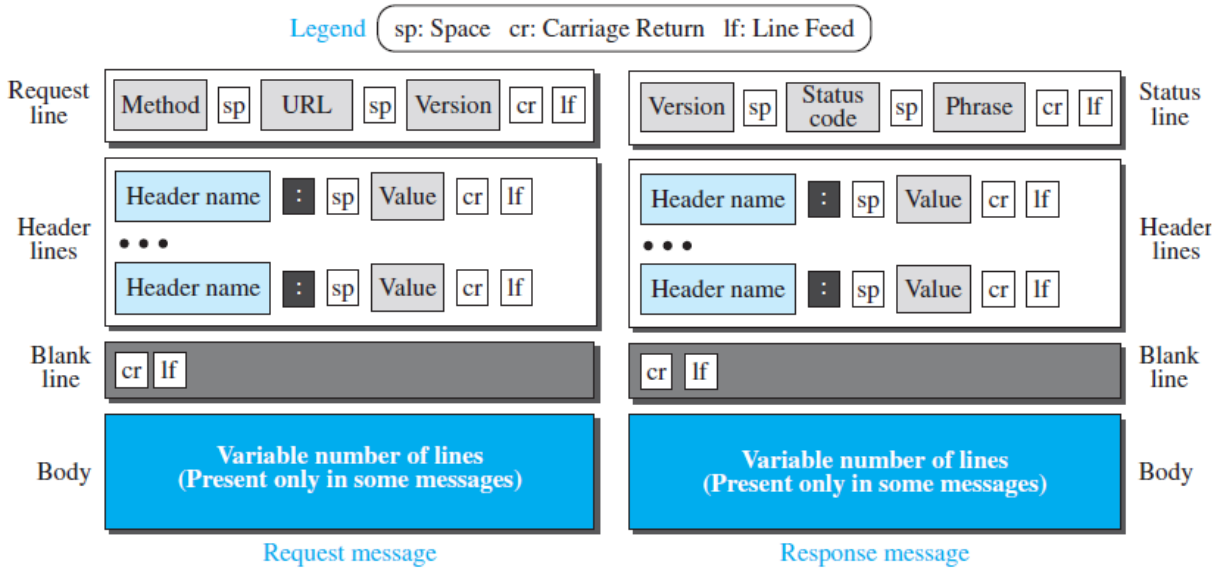

Figure 2.3 Formats of the Request and Response Messages

Request Message

As we said before, the first line in a request message is called a request line. There are three fields in this line separated by one space and terminated by two characters (carriage return and line feed) as shown in Figure 2.3. The fields are called *method, URL*, and *version*.

The method field defines the request types. In version 1.1 of HTTP, several methods are defined, as shown in Table 26.1. Most of the time, the client uses the GET method to send a request. In this case, the body of the message is empty. The HEAD method is used when the client needs only

some information about the web page from the server, such as the last time it was modified. It can also be used to test the validity of a URL. The response message in this case has only the header section; the body section is empty. The PUT method is the inverse of the GET method; it allows the client to post a new web page on the server (if permitted). The POST method is similar to the PUT method, but it is used to send some information to the server to be added to the web page or to modify the web page. The TRACE method is used for debugging; the client asks the server to echo back the request to check whether the server is getting the requests. The DELETE method allows the client to delete a web page on the server if the client has permission to do so. The CONNECT method was originally made as a reserve method; it may be used by proxy servers, as discussed later. Finally, the OPTIONS method allows the client to ask about the properties of a web page. The second field, URL, was discussed earlier in the chapter. It defines the address and name of the corresponding web page. The third field, version, gives the version of the protocol; the most current version of HTTP is 1.1.

After the request line, we can have zero or more *request header* lines. Each header line sends additional information from the client to the server. For example, the client can request that the document be sent in a special format. Each header line has a header name, a colon, a space, and a header value (see Figure 2.3). Table 2.2 shows some header names commonly used in a request. The value field defines the values associated with each header name. The list of values can be found in the corresponding RFCs. The body can be present in a request message. Usually, it contains the comment to be sent or the file to be published on the website when the method is PUT or POST.

Method	*Action*
GET	Requests a document from the server
HEAD	Requests information about a document but not the document itself
PUT	Sends a document from the client to the server
POST	Sends some information from the client to the server
TRACE	Echoes the incoming request
DELETE	Removes the web page
CONNECT	Reserved
OPTIONS	Inquires about available options

Table 2.1 Methods

Header	*Description*
User-agent	Identifies the client program
Accept	Shows the media format the client can accept
Accept-charset	Shows the character set the client can handle
Accept-encoding	Shows the encoding scheme the client can handle
Accept-language	Shows the language the client can accept
Authorization	Shows what permissions the client has
Host	Shows the host and port number of the client
Date	Shows the current date
Upgrade	Specifies the preferred communication protocol
Cookie	Returns the cookie to the server (explained later)
If-Modified-Since	If the file is modified since a specific date

Table 2.2 Request Header Names

Response Message

The format of the response message is also shown in Figure 2.3. A response message consists of a status line, header lines, a blank line, and sometimes a body. The first line in a response message is called the *status line*. There are three fields in this line separated by spaces and terminated by a carriage return and line feed. The first field defines the version of HTTP protocol, currently 1.1. The status code field defines the status of the request. It consists of three digits. Whereas the codes in the 100 range are only informational, the codes in the 200 range indicate a successful request. The codes in the 300 range redirect the client to another URL, and the codes in the 400 range indicate an error at the client site. Finally, the codes in the 500 range indicate an error at the server site. The status phrase explains the status code in text form.

After the status line, we can have zero or more *response header* lines. Each header line sends additional information from the server to the client. For example, the sender can send extra information about the document. Each header line has a header name, a colon, a space, and a header value. We will show some header lines in the examples at the end of this section. Table 2.3 shows some header names commonly used in a response message.

Header	*Description*
Date	Shows the current date
Upgrade	Specifies the preferred communication protocol
Server	Gives information about the server
Set-Cookie	The server asks the client to save a cookie
Content-Encoding	Specifies the encoding scheme
Content-Language	Specifies the language
Content-Length	Shows the length of the document
Content-Type	Specifies the media type
Location	To ask the client to send the request to another site
Accept-Ranges	The server will accept the requested byte-ranges
Last-modified	Gives the date and time of the last change

Table 2.3 Response Header Names

The body contains the document to be sent from the server to the client. The body is present unless the response is an error message.

Example 2.1
This example retrieves a document (see Figure 2.4). We use the GET method to retrieve an image with the path ***/usr/bin/image1*.** The request line shows the method (GET), the URL, and the HTTP version (1.1). The header has two lines that show that the client can accept images in the GIF or JPEG format. The request does not have a body. The response message contains the status line and four lines of header. The header lines define the date, server, content encoding (MIME version, which will be described in electronic mail), and length of the document. The body of the document follows the header.

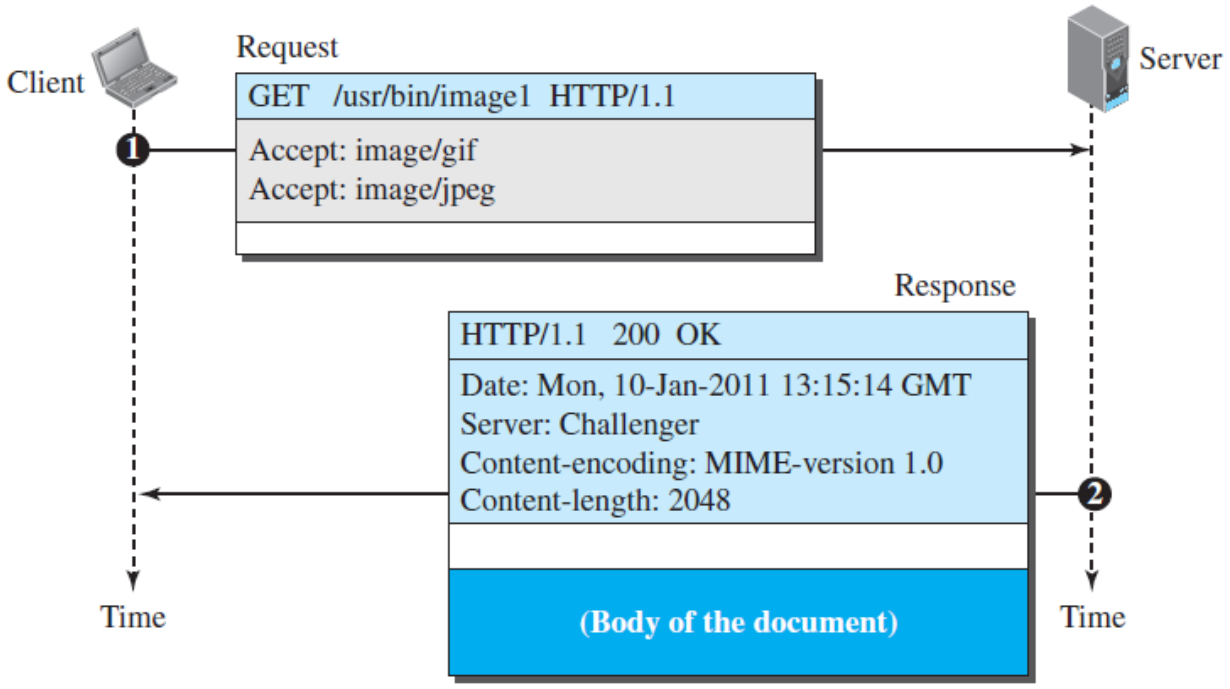

Figure 2.4 Example 2.1

Example 2.2

In this example, the client wants to send a web page to be posted on the server. We use the PUT method. The request line shows the method (PUT), URL, and HTTP version (1.1). There are four lines of headers. The request body contains the web page to be posted. The response message contains the status line and four lines of headers. The created document, which is a CGI document, is included as the body (see Figure 2.5).

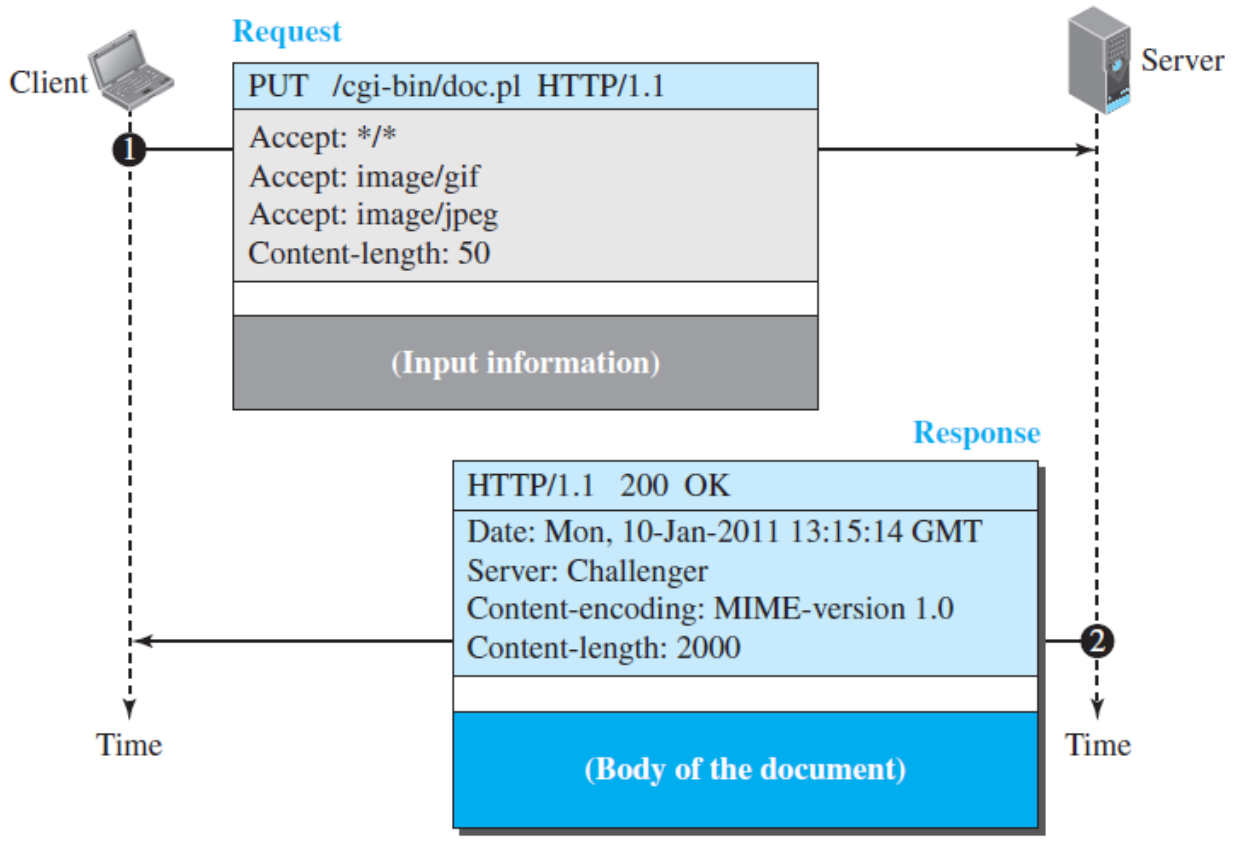

Figure 2.5 Example 2.2

2.5 DNS

The Domain Name System (DNS) is the phonebook of the Internet. Humans access information online through domain names, like nytimes.com or espn.com. Web browsers interact through Internet Protocol (IP) addresses. DNS translates domain names to IP addresses so browsers can load Internet resources.

Each device connected to the Internet has a unique IP address which other machines use to find the device. DNS servers eliminate the need for humans to memorize IP addresses such as 192.168.1.1 (in IPv4), or more complex newer alphanumeric IP addresses such as 2400:cb00:2048:1::c629:d7a2 (in IPv6).

2.6 How Does DNS Work

The process of DNS resolution involves converting a hostname (such as www.example.com) into a computer-friendly IP address (such as 192.168.1.1). An IP address is given to each device on the Internet, and that address is necessary to find the appropriate Internet device - like a street address

is used to find a particular home. When a user wants to load a webpage, a translation must occur between what a user types into their web browser (example.com) and the machine-friendly address necessary to locate the example.com webpage.

In order to understand the process behind the DNS resolution, it's important to learn about the different hardware components a DNS query must pass between. For the web browser, the DNS lookup occurs "behind the scenes" and requires no interaction from the user's computer apart from the initial request.

2.7 4 DNS servers involved in loading a webpage:

- DNS recursor - The recursor can be thought of as a librarian who is asked to go find a particular book somewhere in a library. The DNS recursor is a server designed to receive queries from client machines through applications such as web browsers. Typically the recursor is then responsible for making additional requests in order to satisfy the client's DNS query.
- Root nameserver - The root server is the first step in translating (resolving) human readable host names into IP addresses. It can be thought of like an index in a library that points to different racks of books - typically it serves as a reference to other more specific locations.
- TLD nameserver - The top level domain server (TLD) can be thought of as a specific rack of books in a library. This nameserver is the next step in the search for a specific IP address, and it hosts the last portion of a hostname (In example.com, the TLD server is "com").
- Authoritative nameserver - This final nameserver can be thought of as a dictionary on a rack of books, in which a specific name can be translated into its definition. The authoritative nameserver is the last stop in the nameserver query. If the authoritative name server has access to the requested record, it will return the IP address for the requested hostname back to the DNS Recursor (the librarian) that made the initial request.

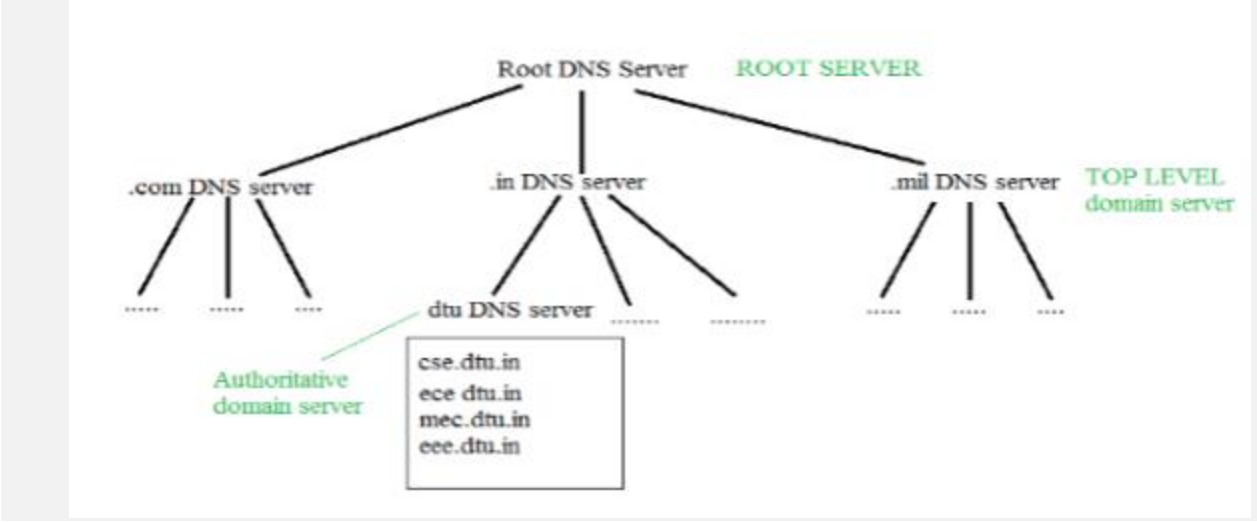

Figure 2.5 Organization of Domain

Difference between an authoritative DNS server and a recursive DNS resolver?

Both concepts refer to servers (groups of servers) that are integral to the DNS infrastructure, but each performs a different role and lives in different locations inside the pipeline of a DNS query. One way to think about the difference is the recursive resolver is at the beginning of the DNS query and the authoritative nameserver is at the end.

Recursive DNS resolver

The recursive resolver is the computer that responds to a recursive request from a client and takes the time to track down the DNS record. It does this by making a series of requests until it reaches the authoritative DNS nameserver for the requested record (or times out or returns an error if no record is found). Luckily, recursive DNS resolvers do not always need to make multiple requests in order to track down the records needed to respond to a client; caching is a data persistence process that helps short-circuit the necessary requests by serving the requested resource record earlier in the DNS lookup.

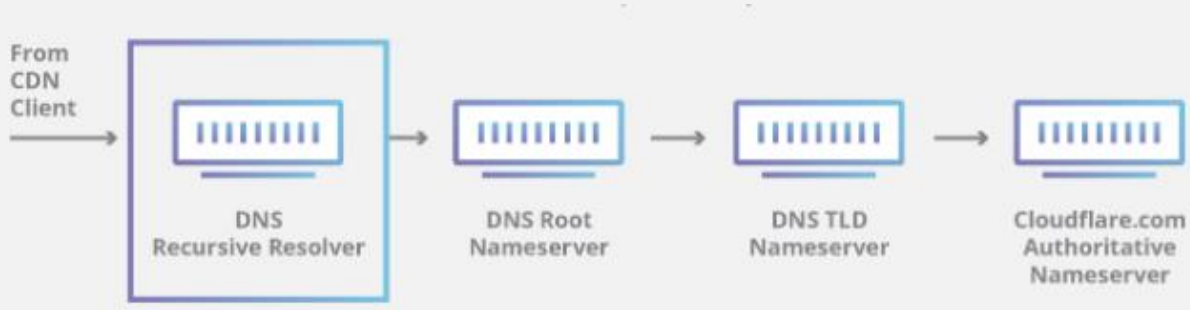

Figure 2.6 DNS Record Request Sequence for Recursive

Authoritative DNS server

Put simply, an authoritative DNS server is a server that actually holds, and is responsible for, DNS resource records. This is the server at the bottom of the DNS lookup chain that will respond with the queried resource record, ultimately allowing the web browser making the request to reach the IP address needed to access a website or other web resources. An authoritative nameserver can satisfy queries from its own data without needing to query another source, as it is the final source of truth for certain DNS records.

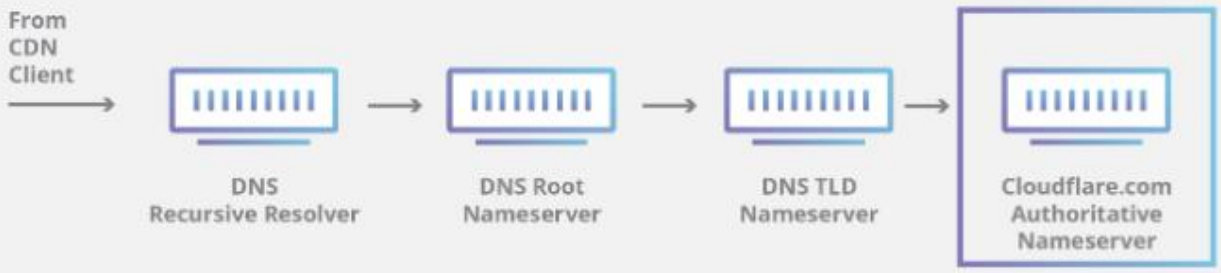

Figure 2.7 DNS Record Request Sequence for Author

2.8 2 Types of DNS Query

Recursive Query

In Recursive name query, the DNS client requires that the DNS server respond to the client with either the requested resource record or an error message i.e. the record or domain name doesn't exist.

If DNS server is not able to resolve the requested query then it forwards the query to another DNS server until it gets an answer or the query fails. We'll take very simple example to explain it, let's assume that you call either yellow pages or just dial to get the information about all the good restaurants near your locality. In this example, Just dial or Yellow pages are working on behalf to get you the required information.

Recursive query is made to DNS server by DNS client or by DNS server that is configured to pass unresolved query to another DNS Server. By default recursive query is enabled but it can be disabled if you don't want to use it in your environment.

Best way to remember Recursive query is to memorize that *burden is on Server to resolve the query*.

Iterative Query

An iterative name query is one in which a DNS client allows the DNS server to return the best answer it can give based on its cache or zone data. If the queried DNS server does not have an exact match for the queried name, the best possible information it can return is a referral (that is, a pointer to a DNS server authoritative for a lower level of the domain namespace).

The DNS client can then query the DNS server for which it obtained a referral. It continues this process until it locates a DNS server that is authoritative for the queried name, or until an error or time-out condition is met.

Best way to remember Iterative query is to memorize that *burden is on Client to resolve the query*.

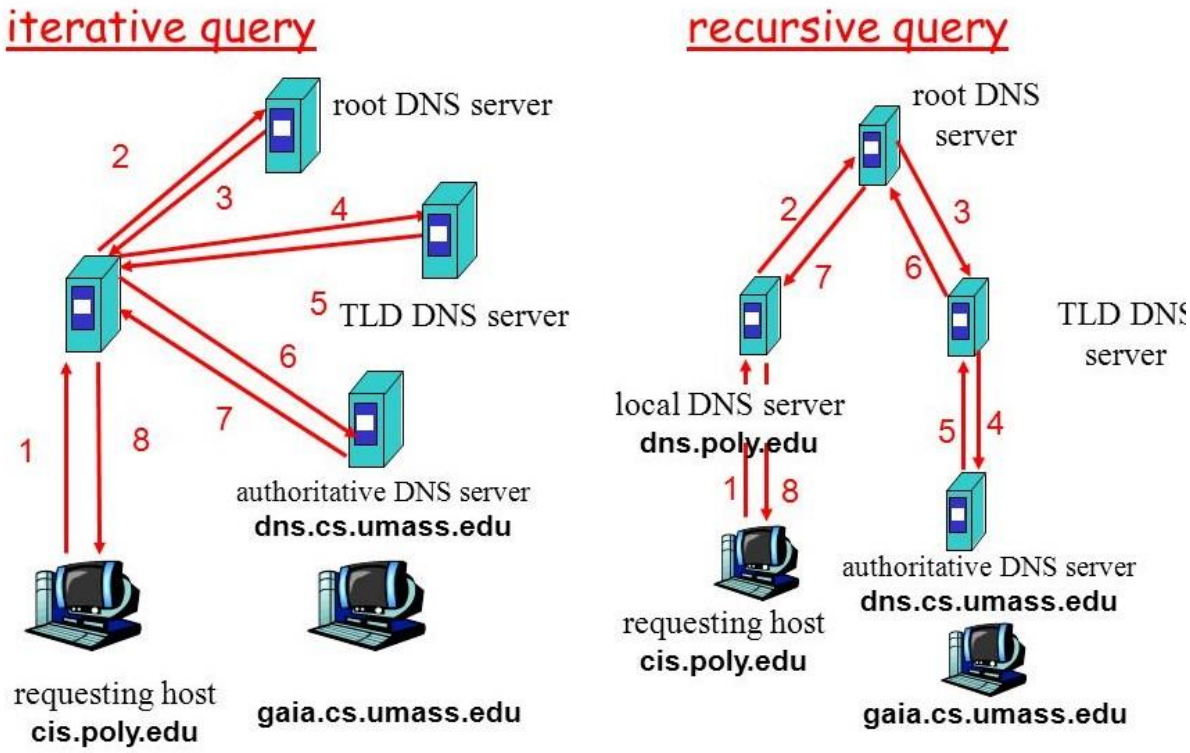

Figure 2.8 Iterative vs. Recursive Query