Networking

TCP vs UDP

TCP	UDP
Reliable: TCP is connection-oriented protocol. When a file or message send it will get delivered unless connections fails. If connection lost, the server will request the lost part. There is no corruption while transferring a message.	Not Reliable: UDP is connectionless protocol. When you a send a data or message, you don't know if it'll get there, it could get lost on the way. There may be corruption while transferring a message.
Ordered: If you send two messages along a connection, one after the other, you know the first message will get there first. You don't have to worry about data arriving in the wrong order.	Not Ordered: If you send two messages out, you don't know what order they'll arrive in i.e. no ordered

Heavyweight: – when the low level parts of the TCP “stream” arrive in the wrong order, resend requests have to be sent, and all the out of sequence parts have to be put back together, so requires a bit of work to piece together.	Lightweight: No ordering of messages, no tracking connections, etc. It’s just fire and forget! This means it’s a lot quicker, and the network card / OS have to do very little work to translate the data back from the packets.
Streaming: Data is read as a [illegible] begins. There may be multiple packets per read call.	Datagrams: Packets are sent [illegible] per one read call.
Examples: World Wide Web (Apache TCP port 80), e-mail (SMTP TCP port 25 Postfix MTA), File Transfer Protocol (FTP port 21) and Secure Shell (OpenSSH port 22) etc.	Examples: Domain Name System (DNS UDP port 53), streaming media applications such as IPTV or movies, Voice over IP (VoIP), Trivial File Transfer Protocol (TFTP) and online multiplayer games

HTTP session

Stateless applications

- Web application servers are generally "stateless":
  - Each HTTP request is independent; server can't tell if 2 requests came from the same browser or user.
  - Web server applications maintain no information in memory from request to request (only information on disk survives from one request to another).
- Statelessness not always convenient for application developers: need to tie together a series of requests from the same user. Since the HTTP protocol is stateless itself, web applications developed techniques to create a concept of a session on top of HTTP so that servers could recognize multiple requests from the same user as parts of a more

complex and longer lasting sequence.

Structure of a session

- The session is a key-value pair data structure. Think of it as a hashtable where each user gets a hashkey to put their data in. This hashkey would be the "session id".

Server-side session vs client-side cookie

Category	Session	Cookie
Location	User ID on server	User ID on web browser
Safeness	Safer because data cannot be viewed or edited by the client	A hacker could manipulate cookie data and attack
Amount of data	Big	Limited
Efficiency	Save bandwidth by passing only a reference to the session (sessionID) each pageload.	Must pass all data to the webserver each pageload
Scalability	Need efforts to scale because requests depend on server state	Easier to implement

Store session state in client-side cookies

Cookie Def

- Cookies are key/value pairs used by websites to store state informations on the browser. Say you have a website (example.com), when the browser requests a webpage the website can send cookies to store informations on the browser.

Cookie typical workflow

```
// Browser request example:
```

```
GET /index.html HTTP/1.1
Host: www.example.com

// Example answer from the server:


HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: foo=10
Set-Cookie: bar=20; Expires=Fri, 30 Sep 2011 11:48:00 GMT
... rest  of the response

// Here two cookies foo=10 and bar=20 are stored on the browser.
The second one will expire on 30 September. In each subsequent
request the browser will send the cookies back to the server.


GET /spec.html HTTP/1.1
Host: www.example.com
Cookie: foo=10; bar=20
Accept: */*
```

Cookie Pros and cons

- Advantage: You do not have to store the sesion state anywhere in your data center. The entire session state is being handed to your web server with every web request, thus making your application stateless in the context of the HTTP session.
- Disadvantage: Session storage can becomes expensive. Cookies are sent by the browser with every single request, regardless of the type of resource being requested. As a result, all requests within the same cookie domain will have session storage appended as part of the request.
- Use case: When you can keep your data minimal. If all you need to keep in session scope is userID or some security token, you will benefit from the simplicity and speed of this solution. Unfortunately, if you are not careful, adding more data to the session scope can quickly grow into kilobytes, making web requests much slower, especially on mobile devices. The coxt of cookie-based session storage is also amplified by the fact that encrypting serialized data and then Based64 encoding increases the overall byte count by one third, so that 1KB of session scope data

becomes 1.3KB of additional data transferred with each web request and web response.

Store session state in server-side

- Approaches:
  - Keep state in main memory
  - Store session state in files on disk
  - Store session state in a database
    - Delegate the session storage to an external data store: Your web application would take the session identifier from the web request and then load session data from an external data store. At the end of the web request life cycle, just before a response is sent back to the user, the application would serialize the session data and save it back in the data store. In this model, the web server does not hold any of the session data between web requests, which makes it stateless in the context of an HTTP session.
    - Many data stores are suitable for this use case, for example, Memcached, Redis, DynamoDB, or Cassandra. The only requirement here is to have very low latency on get-by-key and put-by-key operations. It is best if your data store provides automatic scalability, but even if you had to do data partitioning yourself in the application layer, it is not a problem, as sessions can be partitioned by the session ID itself.

Typical server-side session workflow

1. Every time an internet user visits a specific website, a new session ID (a unique number that a web site's server assigns a specific user for the duration of that user's visit) is generated. And an entry is created inside server's session table

Columns	Type	Meaning
sessionID	string	a global unique hash value
userId	Foreign key	pointing to user table
expireAt	timestamp	when does the session expires

2. Server returns the sessionID as a cookie header to client
3. Browser sets its cookie with the sessionID
4. Each time the user sends a request to the server. The cookie for that domain will be automatically attached.
5. The server validates the sessionID inside the request. If it is valid, then the user has logged in before.

Use a load balancer that supports sticky sessions:

- The load balancer needs to be able to inspect the headers of the request to make sure that requests with the same session cookie always go to the server that initially the cookie.
- But sticky sessions break the fundamental principle of statelessness, and I recommend avoiding them. Once you allow your web servers to be unique, by storing any local state, you lose flexibility. You will not be able to restart, decommission, or safely auto-scale web servers without braking user's session because their session data will be bound to a single physical machine.

DNS

- Resolve domain name to IP address

Design

Initial design

- A simple design for DNS would have one DNS server that contains all the mappings. But the problems with a centralized design include:
    - **A single point of failure**: If the DNS server crashes, so does the entire Internet.
    - **Traffic volume**: A single DNS server would have to handle all DNS queries.
    - **Distant centralized database**: A single DNS server cannot be close to all the querying clients.
    - **Maintenance**: The single DNS server would have to keep records for all Internet hosts. It needed to be updated frequently

A distributed, hierarchical database

- **Root DNS servers**:
- **Top-level domain servers**: Responsible for top level domains such as com, org, net, edu, and gov, and all of the country top-level domains such as uk, fr, ca, and jp.
- **Authoritative DNS servers**:
- **Local DNS server**: Each ISP - such as a university, an academic department, an employee's company, or a residential ISP - has a local DNS server. When a host connects to an ISP, the ISP provides the host with the IP addresses of one of its local DNS servers.

Internals

DNS records

- The DNS servers store source records (RRs). A resource record is a four-tuple that contains the following fields: (Name, Value, Type, TTL)
- There are the following four types of records
  - If Type=A, then Name is a hostname and Value is the IP address for the hostname. Thus, a Type A record provides the standard hostname-to-IP address mapping. For example, (relay1.bar.foo.com, 145.37.93.126, A) is a Type A record
  - If Type=NS, then Name is a domain and Value is the hostname of an authoritative DNS server that knows how to obtain the IP addresses for hosts in the domain. This record is used to route DNS queries further along in the query chain. As an example, (foo.com, dns.foo.com, NS) is a Type NS record.
  - If Type=CNAME, then Value is a canonical hostname for the alias hostname Name.
  - If Type=MX, then Value is the canonical name of a mail server that has an alias hostname Name.

Insert records into DNS DB

- Take domain name networkutopia.com as an example.
- First you need to register the domain name network. A registrar is a commercial entity that verifies the uniqueness of the domain name, enters

the domain name into the DNS database and collects a small fee for its services.
    - When you register, you need to provide the registrar with the names and IP addresses of your primary and secondary authoritative DNS servers. For each of these two authoritative DNS servers, the registrar would then make sure that a Type NS and a Type A record are entered into the TLD com servers.

DNS query parsing

- When a user enters a URL into the browser's address bar, the first step is for the browser to resolve the hostname (http://www.amazon.com/index.html) to an IP address. The browser extracts the host name www.amazon.com from the URL and delegates the resolving task to the operating system. At this stage, the operating system has a couple of choices.
- It can either resolve the address using a static hosts file (such as /etc/hosts on Linux)
- It then query a local DNS server.
    - The local DNS server forwards to a root DNS server. The root DNS server takes not of the com suffix and returns a list of IP addresss for TLD servers responsible for com domain
    - The local DNS server then resends the query to one of the TLD servers. The TLD server takes note of www.amazon. suffix and respond with the IP address of the authoritative DNS server for amazon.
    - Finally, the local DNS server resends the query message directly to authoritative DNS which responds with the IP address of www.amazon.com.
- Once the browser receives the IP addresses from DNS, it can initiate a TCP connection to the HTTP server process located at port 80 at that IP address.

Types

Round-robin DNS

- A DNS server feature that allowing you to resolve a single domain name to one of many IP addresses.

GeoDNS

- A DNS service that allows domain names to be resolved to IP addresses based on the location of the customer. A client connecting from Europe may get a different IP address than the client connecting from Australia. The goal is to direct the customer to the closest data center to minimize network latency.

Functionality

DNS Caching

- Types:
  - Whenever the client issues a request to an ISP's resolver, the resolver caches the response for a short period (TTL, set by the authoritative name server), and subsequent queries for this hostname can be answered directly from the cache.
  - All major browsers also implement their own DNS cache, which removes the need for the browser to ask the operating system to resolve. Because this isn't particularly faster than quuerying the operating system's cache, the primary motivation here is better control over what is cached and for how long.
- Performance:
  - DNS look-up times can vary dramatically - anything from a few milliseconds to perhaps one-half a second if a remote name server must be queried. This manifests itself mostly as a slight delay when the user first loads the site. On subsequent views, the DNS query is answered from a cache.

Load balancing

- DNS can be used to perform load distribution among replicated servers, such as replicated web servers. For replicated web servers, a set of IP addresses is thus associated with one canonical hostname. The DNS database contains this set of IP addresses. When clients make a DNS query for a name mapped to a set of addresses, the server responds with the entire set of IP addresses, but rotates the ordering of the addresses within each reply. Because a client typically sends its HTTP request to the IP address that is the first in the set, DNS rotation distributes the traffic among the replicated servers.

Host alias

- A host with a complicated hostname can have one or more alias names. For example, a hostname such as relay1.west-coast.enterprise.com could have two aliases such as enterprise.com and www.enterprise.

DNS prefetching

Def

- Performing DNS lookups on URLs linked to in the HTML document, in anticipation that the user may eventually click one of these links. Typically, a single UDP packet can carry the question, and a second UDP packet can carry the answer.

Control prefetching

- Most browsers support a link tag with the nonstandard rel="dns-prefetch" attribute. This causes teh browser to prefetch the given hostname and can be used to precache such redirect linnks. For example

- In addition, site owners can disable or enable prefetching through the use of a special HTTP header like:

> X-DNS-Prefetch-Control: off

Security

SSL

Definition

- Hyper Text Transfer Protocol Secure (HTTPS) is the secure version of HTTP, the protocol over which data is sent between your browser and the website that you are connected to. The 'S' at the end of HTTPS stands for 'Secure'. It means all communications between your browser and the website are encrypted. HTTPS is often used to protect highly confidential online transactions like online banking and online shopping order forms.

How does HTTPS work

- HTTPS pages typically use one of two secure protocols to encrypt communications - SSL (Secure Sockets Layer) or TLS (Transport Layer Security). Both the TLS and SSL protocols use what is known as an 'asymmetric' Public Key Infrastructure (PKI) system. An asymmetric system uses two 'keys' to encrypt communications, a 'public' key and a 'private' key. Anything encrypted with the public key can only be decrypted by the private key and vice-versa.
- As the names suggest, the 'private' key should be kept strictly protected and should only be accessible the owner of the private key. In the case of a website, the private key remains securely ensconced on the web server. Conversely, the public key is intended to be distributed to anybody and everybody that needs to be able to decrypt information that was encrypted with the private key.

How to avoid public key being modified?

- Put public key inside digital certificate.
  - When you request a HTTPS connection to a webpage, the website will initially send its SSL certificate to your browser. This certificate contains the public key needed to begin the secure session. Based on this initial exchange, your browser and the website then initiate the 'SSL handshake'. The SSL handshake involves the generation of shared secrets to establish a uniquely secure connection between yourself and the website.
  - When a trusted SSL Digital Certificate is used during a HTTPS connection, users will see a padlock icon in the browser address bar. When an Extended Validation Certificate is installed on a web site, the address bar will turn green.

How to avoid computation consumption from PKI

- Only use PKI to generate session key and use the session key for further communications.

Web server

Apache and Nginx

- Apache and Nginx could always be used together.
  - NGINX provides all of the core features of a web server, without sacrificing the lightweight and high-performance qualities that have made it successful, and can also serve as a proxy that forwards HTTP requests to upstream web servers (such as an Apache backend) and FastCGI, memcached, SCGI, and uWSGI servers. NGINX does not seek to implement the huge range of functionality necessary to run an application, instead relying on specialized third-party servers such as PHP-FPM, Node.js, and even Apache.
  - A very common use pattern is to deploy NGINX software as a proxy in front of an Apache-based web application. Can use Nginx's proxying abilities to forward requests for dynamic resources to Apache backend server. NGINX serves static resources and Apache serves dynamic content such as PHP or Perl CGI scripts.

Apache vs Nginx

Category	Apache	Nginx
History	Invented around 1990s when web traffic is low and web pages are really simple. Apache's heavyweight, monolithic model has its limit. Tunning Apache to cope with real-world traffic efficiently is a complex art.	Heavy traffic and web pages. Designed for high concurrency. Provides 12 features including which make them appropriate for microservices.
	One process/threads per connection.	Asynchronous event-driven model. There is

Architecture	Each requests to be handled as a separate child/thread.	a single master process with one or more worker processes.
Performance	To decrease page-rendering time, web browsers routinely open six or more TCP connections to a web server for each user session so that resources can download in parallel. Browsers hold these connections open for a period of time to reduce delay for future requests the user might make during the session. Each open connection exclusively reserves an httpd process, meaning that at busy times, Apache needs to create a large number of processes. Each additional process consumes an extra 4MB or 5MB of memory. Not to mention the overhead involved in creating and destroying child processes.	Can handle a huge number of concurrent requests
Easier development	Very easy to insert additional code at any point in Apache's web-serving logic. Developers could add code securely in the knowledge that if newly added code is blocked, ran slowly, leaked resources, or even crashed, only the worker process running the code would be affected. Processing of all other connections	Developing modules for it isn't as simple and easy as with Apache. Nginx module developers need to be very careful to create efficient and accurate code, without any resource leakage, and to interact

	would continue undisturbed	appropriately with the complex event-driven kernel to avoid blocking operations.