**Distributed System**:

bringing multiple systems machine, dependent of one another together in order to run some application/server.  Some set of information is distributed across all systems connected to each other over internet. Several independent computers connected to each other.

Important link for distributed system:  http://www.aosabook.org/en/distsys.html

Distributed System Architecture

* **Fault-Tolerant**: It can recover from component failures without performing incorrect actions.
* **Highly Available**: It can restore operations, permitting it to resume providing services even when some components have failed.
* **Recoverable**: Failed components can restart themselves and rejoin the system, after the cause of failure has been repaired.
* **Consistent**: The system can coordinate actions by multiple components often in the presence of concurrency and failure. This underlies the ability of a distributed system to act like a non-distributed system.
* **Scalable**: It can operate correctly even as some aspect of the system is scaled to a larger size. For example, we might increase the size of the network on which the system is running. This increases the frequency of network outages and could degrade a "non-scalable" system. Similarly, we might increase the number of users or servers, or overall load on the system. In a scalable system, this should not have a significant effect.
* **Predictable** Performance: The ability to provide desired responsiveness in a timely manner.
* **Secure**: The system authenticates access to data and services [1]
* **Manageability:** Designing a system that is easy to operate is another important consideration. The manageability of the system equates to the scalability of operations: maintenance and updates. Things to consider for manageability are the ease of diagnosing and understanding problems when they occur, ease of making updates or modifications, and how simple the system is to operate. (I.e., does it routinely operate without failure or exceptions?)

**Example of distributed system:**

* client - server architecture
* bit torrent
* web application .
* Spring Distribution.

**Some Distributed Design Principles**

Given what we have covered so far, we can define some fundamental design principles which every distributed system designer and software engineer should know. Some of these may seem obvious, but it will be helpful as we proceed to have a good starting list

* System to be well designed to handled nay kind of failure / proper fault tolerant supported.
* Keep a track of whatever data we are sending. Think carefully about how much data you send over the network. Minimize traffic as much as possible.
* Consider monitoring our latency to decide what response time we can consider to respond to  the client
* Use secure connections to connect to server back and forth.
* Caches and replication strategies are methods for dealing with state across components.

**Leader Election Algorithm;**

* In a distributed system we do have a number of system and they inter communicates with each other. through different approach of RPC and others.
* In [distributed computing](https://en.wikipedia.org/wiki/Distributed_computing), **leader election** is the process of designating a single [process](https://en.wikipedia.org/wiki/Process_(computing)) as the organizer of some task distributed among several computers (nodes)
* So here we will define one system as leader of all the systems to decide/organize.
* Need to decide which hosts is going to act as coordinator once a process/hosts goes down.

Considerations :

* All individual systems can be identified using some unique numbers.
* Process knows each other via the unique numbers that can be process number.

**Things to keep in mind while deciding distributed system:**

CAP Theorem: Consistency , Avaialbility and Persistence

Replication ./ Consistency is managed by strong and eventually consistent

**Content delivery network**

* This basically helps you to reduce the number of hops in order to help a client get there request served.
* It stores a content in manny locations as per clients requesting them.

**Design Principle for Distributed Systems:**

* Services : It should be service based architecture to keep it integrable and maintainable.

**Service Oriented Architecture:**

* This provides the facility of writing solution to our problem into multiple domains in terms of services.
* When we are performing any read/ write operation write two services for read and write separately.
* This will help us not to worry about writing and retrieving new images in the same context and in terms of maintenance as well , this will help to make it independent of optimizing things in its own way/performance.
* Other way of optimizing this is by using shards as per user data.
* In the former an outage or issue with one of the services brings down functionality across the whole system (no-one can write files, for example), whereas an outage with one of Flickr's shards will only affect those users for which the shard is impacted.
* Updating data is much simpler in context when we are using service based changes. However, managing changes and updates is quite difficult with shards based upon user data and we will have to search each shard for any search operations.

**Redundancy** :

There are different ways of implementing redundancy in our system.

* Running multiple instances of our service help you to recover any of such fault/failure of production system . So running different instance can help you get the request serve from other service running on the same node instantly.
* Another approach here is to use multiple copies of any file across different geographical locations. So if in case one of them goes down. we can have them ready in other locations.

**Fast and scalable data consideration** :

**Access**:

* We have to decide our usage in terms of our data access.
* We have to decide whether we need to read any data from disk or other memory. As reading from disk is much faster as compared to reading from memory.
* Different ways of improving your data reading feature.

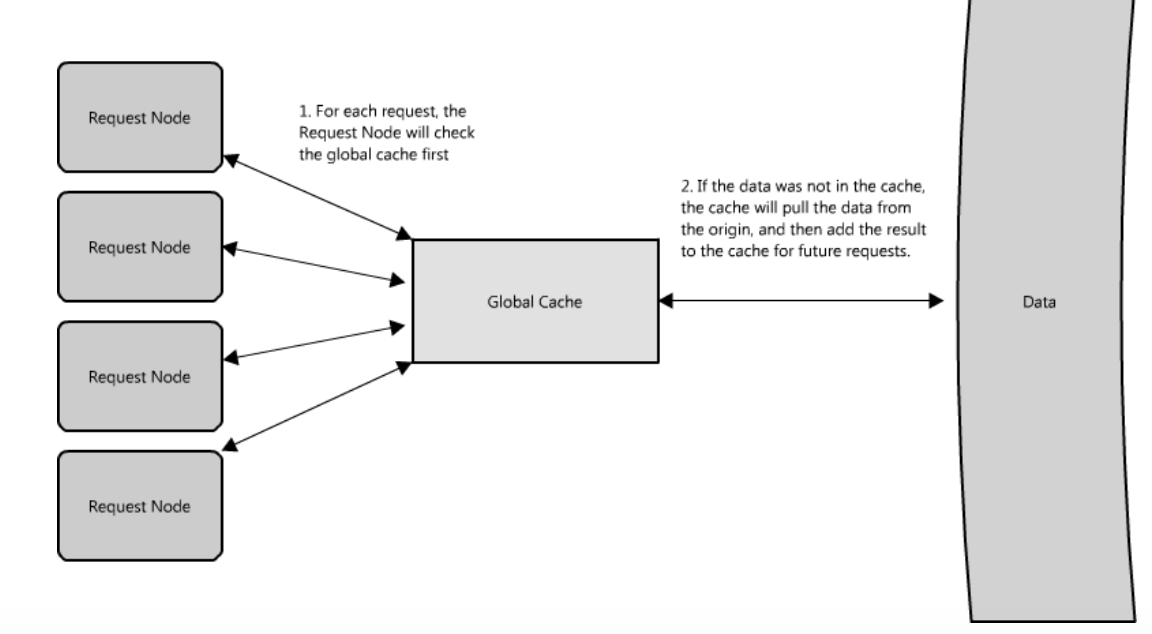
**# Caching :**

A cache is like short-term memory:

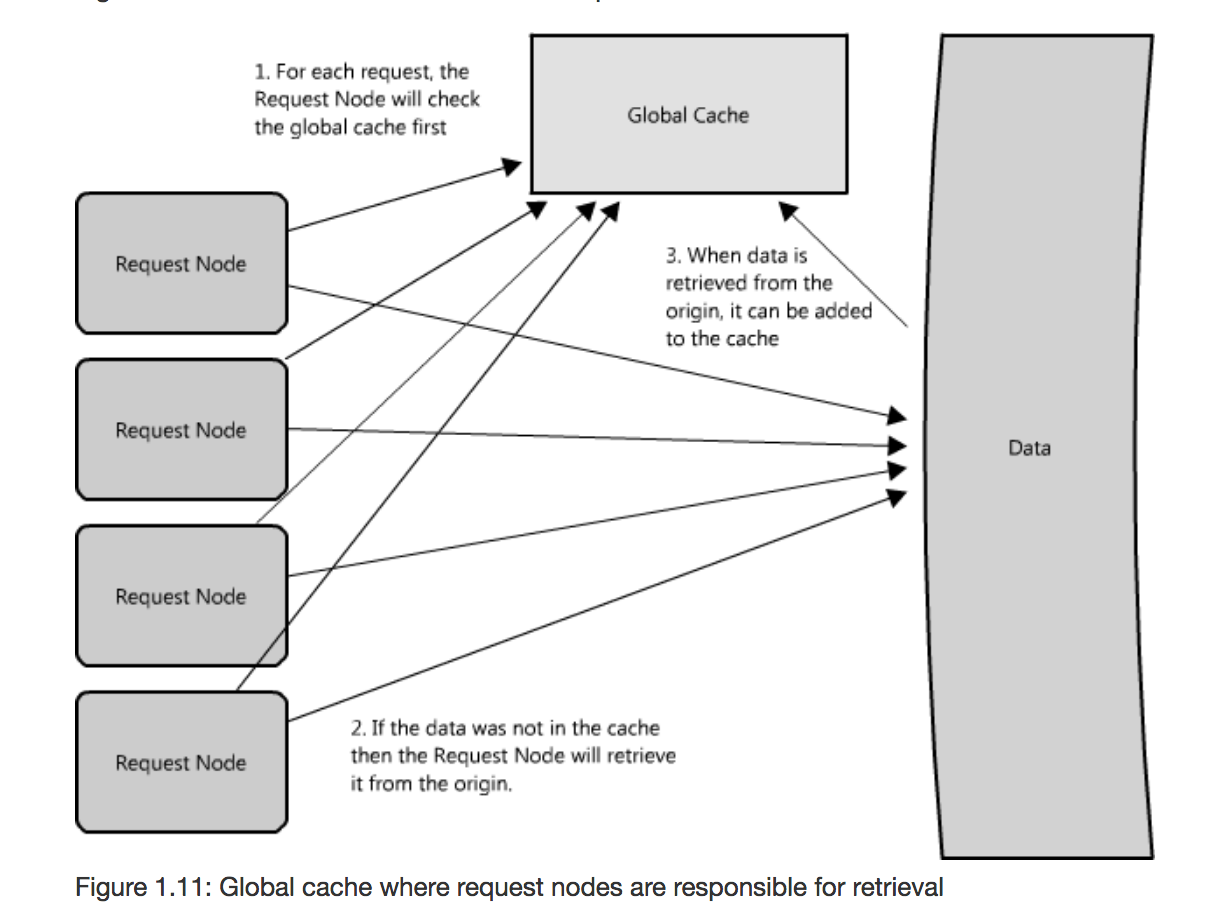
* it has a limited amount of space,
* typically faster than the original data source and
* contains the most recently accessed items. Caches can exist at all levels in architecture, but are often found at the level nearest to the front end, where they are implemented to return data quickly without taxing downstream levels.

**Different ways of implementing cache:**

* **Caching in local node** :
  + We can store our elements in our local caching for specific set of data and update them as per our usability.
  + In this scenario, first of all in case we see any data request , we will search for local storage first if it exists will return with the data else it will reach out to db and return the value along with updating the cache content locally.
  + **Cons**: This is helpful even we have a number of hosts in line running the service. Each of them will do this locally.
  + **Pros :**In this scenario , if we have multiple number of hosts and we are using load balancer to redirect traffic across them, we have to cache same element across different nodes , as loadbaanvcer can redirect traffic to any such node as per algorithm defined. so chances of cache miss will probably increase.
* **Global Cache** : To avoid redundancy of common elements across the multiple , cached across different nodes. We can use a centralized dedicated server for perofrimning any kind of such caching and for any kind of caching nodes will this server for caring elements .
  + The global will be accessed by any other nodes to get caching element.
  + Global caching host is generally decided to be very fast wrt to requesting nodes,
  + there are two ways of getting data from global cache by the client.
    - Directly redirecting the host to global cache and reading cache , in case of cache miss , this will be redirected to the backend data source and update the cache at the same time.



* + - Other approach is where we will check the global cache first and serve the request. In case of any cache miss this will be redirected to direct to the data source (origin) and later this will update the global cache also.



* + However, there are some cases where the second implementation makes more sense. For example, if the cache is being used for very large files, a low cache hit percentage would cause the cache buffer to become overwhelmed with cache misses; in this situation it helps to have a large percentage of the total data set (or hot data set) in the cache. Another example is an architecture where the files stored in the cache are static and shouldn't be evicted. (This could be because of application requirements around that data latency—certain pieces of data might need to be very fast for large data sets—where the application logic understands the eviction strategy or hot spots better than the cache.)

* **Distributed caching:**
  + Using memcahed as solution
  + Advantage:
    - increaser cache space .
    - faster in nature
  + Disadvantage
    - no recovery for failed hosts and
    - node addition is a problem.

**Proxy Vs Cache**

* Using cache prior to proxies to serve the requests faster thats already in memory without unnecessary delay/latency.

It is worth noting that you can use proxies and caches together, but generally it is best to put the cache in front of the proxy, for the same reason that it is best to let the faster runners start first in a crowded marathon race. This is because the cache is serving data from memory, it is very fast, and it doesn't mind multiple requests for the same result. But if the cache was located on the other side of the proxy server, then there would be additional latency with every request before the cache, and this could hinder performance

* Example of proxies available , example of reverse proxy : https://varnish-cache.org/ http://www.squid-cache.org/
* Proxy functions can further be categorized under reverse proxy  :
* Reverse proxy are enable at application  location so that this is the first thing to respond to a clients requests if already cached locally.

**Indexes** :

This helps to make your reading of database easier and data access performance. So maintaining a table of indexes. We have these indices stored somewhere locally with at the client side.

We can use inverted indexing as well

An inverted index, which could represent Index1 in the diagram above, might look something like the following—each word or tuple of words provide an index of what books contain them.

|  |  |
| --- | --- |
| **Word(s)** | **Book(s)** |
| being awesome | Book B, Book C, Book D |
| always | Book C, Book F |
| believe | Book B |

* Being able to find your data quickly and easily is important; indexes are an effective and simple tool to achieve this.

**Load Balancers**

Finally, another critical piece of any distributed system is a load balancer.This is a principle part of any architecture.Technically these are also known as reverse proxies as it can perform the task of filtering the request on the basis of their nature.

Usage:

* an easy way to allow you to expand system capacity in distributed system
* This also can help to test health of any node as they are actively working or not by frequent health checks pings such that if a node is unresponsive or over-loaded, it can be removed from the pool handling requests, taking advantage of the redundancy of different nodes in your system

**Queues**:

Usage of Queues in Web architecture :

* In above explained architecture there can be a possibility where different clients sending requests to one/multiple servers using load balancers via algorithms that can be implemented.
* But performance of different host can vary in terms of handling a client requests. So there can be a scenario where a particular host is not performing well and a number of request are in pending state to the hosts.
* So in case when we get more requests than it is able to serve , then each client is forced to wait for the other clients.
* even with effective load balancing in place it is extremely difficult to ensure the even and fair distribution of work required to maximize client performance.
* we will create queues to keep track of unserved client requests.
* This will help us to work with asynchronous system where we don’t have to wait for any achknowldgement
* Queues enable clients to work in an asynchronous manner, providing a strategic abstraction of a client's request and its response. On the other hand, in a synchronous system, there is no differentiation between request and reply, and they therefore cannot be managed separately.
* Queues also provide some protection from service outages and failures. For instance, it is quite easy to create a highly robust queue that can retry service requests that have failed due to transient server failures.
* Data store like Redis is a part of these queues.

/apollo/env/TPSGenerator/bin/TPSGenerator -find\_max\_tps=10 async\_remaining\_work\_poller=true period=PT3M start\_step=1 linear=true -materialset com.amazon.promotions.aws.prod.knox.credentials -transactionCreator BouncebackFulfillmentServiceLoadGenerator -transactionCreatorPath BouncebackFulfillmentService-1.0.jar variables timberFileLocation=timber://onlinelogs/BouncebackFulfillmentService/NA/Prod/2018/03/14/01/orderEvents.log -variables timberOwnerEmail=promo-team+prod@amazon.com -variables remoteDataCenter=pdx -variables timberLogStartTimeShift=PT2H -variables timberLogEndTimeShift=PT1H --loadTestingQueueUrl https://sqs.us-east-1.amazonaws.com/743084686769/BFSStressTestSQSQueue-prod --awsmaterialset=com.amazon.access.BouncebackFulfillment-prod-BFS-1

**Important links for system architecture** :

http://www.aosabook.org/en/index.html

/apollo/env/TPSGenerator/bin/TPSGenerator -find\_max\_tps async\_remaining\_work\_poller=true period=PT3M start\_step=1 linear=true -materialset com.amazon.promotions.aws.prod.knox.credentials -transactionCreator BouncebackFulfillmentServiceTestSuite -transactionCreatorPath /tmp/BouncebackFulfillmentServiceTestSuite-1.0-test-runtime.tgz**-variables timberFileLocation=BouncebackFulfillmentService/Beta/orderEvents.log** -variables timberOwnerEmail=promo-team+prod@amazon.com -variables remoteDataCenter=pdx -variables timberLogStartTimeShift=PT2H -variables timberLogEndTimeShift=PT1H -loadTestingQueueUrl <https://sqs.us-east-1.amazonaws.com/743084686769/BFSStressTestSQSQueue-prod> -awsmaterialset=com.amazon.access.BouncebackFulfillment-prod-BFS-1

Materials to refer :

http://www.aosabook.org/en/distsys.html

https://github.com/theanalyst/awesome-distributed-systems