1 Distributed Systems Background

A distributed systems is defined as – a collection of nodes working together collectively to provide functionality that transcends that of any individual node. At Google, the search engine indexes several Petabytes of data, to support this multiple machines were needed. At Facebook, to provide websites at very low latency, machines needed to placed closer to the user, and hence multiple machines were required. Regardless of the motivation for adopting a distributed system, the design of a distributed systems faces several crucial and important challenges.

In this class, we will focus on four key challenges:

- **Scale**: if many clients are making requests to a server, does the server respond within a timely manner?
- **Failure**: if a server fails, how does the protocol recovery and adapt to the failure?
- **Consistency**: if a client makes a request to a server, does the server respond with the most recent answer?
- **Availability/Reliability**: if a client makes a request to a server, does the server respond?
- **Security**: is the system secure, and resilient to attacks. (we will not discuss security in this class)

The distributed aspect of distributed systems places a unique set of challenges that often do not arise when designing and developing a simple program that runs with a process on one host. A subset of these challenges are articulated in Peter Deutsch’s fallacies of distributed computing. Below I briefly list and summarize them.

![Figure 1: Simple program to distributed system.](image)

- **Network is reliable**: the network will always deliver packets. Systems designed with this assumption in mind may fall in a deadlock waiting for packets that the network has dropped.
DISTRIBUTED SYSTEMS BACKGROUND

• Latency is zero: the network will deliver the packets instantaneously. Systems designed with this assumption may perform horrible because of waiting for packets.

• Bandwidth is infinite: the network can support an infinite amount of traffic. Systems designed with this assumption may overwhelm the network which lead to loss of data and ultimately slows down or leads to a hung system.

• Network is secure.

• Topology is constant: the devices maintain a fixed position (i.e., IP address). In reality, due to failures and other management issues, applications often change IP address.

• Networked is owned by one entity: the network can be easily modified. In reality, the network is owned by different teams. A change required coordination and orchestration.

• Transport cost is zero: the cost to send or receive data is zero. There is infact cost associated with serializing and unserializing data.

• Network is homogenous: all devices in the network have the same capabilities and limitations.

Unsurprisingly, an overwhelming number of these items are related to the network. This is unsurprising because the use of the network is what differentiates distributed systems from traditional non-distributed systems.
2 Networking

To communicate between distributed systems the first thing two computer must do is to discover each other. For $S_A$ to interact with $S_B$, $S_A$ must be able to determine the location of $S_B$. This can be done in one of two ways, first, hardcoding IP addresses in all applications. This approach is not tenable because: (1) servers can change IP-addresses for a range of management reasons, (2) server fails and applications may need to be migrated between servers. Second, using domain names to refer to server. This second approach requires a scalable service for mapping domain names to IPAddress. This scalable service must be fast, resilient, and secure (we wouldn’t discuss security).

2.1 Naming

Scaling DNS (Domain name service) is the solution to the naming service. It tackles scale by using a hierarchical design which delegates ownership and management of different subdomains to different entities. To provide resilience, the organizations can run multiple servers.

Speed To provide speed, clients can cache information received from the name servers. Althought the benefits are clear, a downside of caching is that client may use outdated and stale information. To prevent this, server often include a time value (TTL – time to live) in response to the clients and the client use this to determine the maximum value to cache.

Uses-of-DNS: At scale, DNS is often used to load balance between multiple servers. This is often used at online service providers, e.g., Amazon, Yahoo. to distribute load. For example, if there are multiple options the name server can respond with a random one. Alternatively, the name server can respond with the closest one. This is often used at online service providers, e.g., Amazon, Yahoo. to distribute load.

![Figure 2: Failures modes.](image)

2.2 Failure Assumptions

When dealing with failures, we either assume the system is synchronous or the system is not-synchronous.

In a synchronous system, time is bounded. In particular, time to send messages between two nodes is bounded. Time to process and respond to messages is also bounded. By making this assumption that time to send and time to process are bounded. Then we can use Timeout to detect failure. If time to send messages or to process is unbounded then the only feasible timeout value is infinite, which does not help to detect failures.
2.3 RPC – Remote Procedure Calls

The easiest and most intuitive abstraction for communication between two nodes is to extend on the function call abstraction. RPC provide a way for two nodes to interact directly using function call abstraction.

To support this RPC frameworks must tackle the following challenges: (1) packaging arguments into packets to be sent over the network, (2) unpacking return values and converting them into objects, (3) dealing with various failure modes.

Marshalling/unmarshalling is the act of converting objects into packets and vice versa. Most RPC frameworks automate this process and only require that the programmer define the objects and the format for the objects that will be sent and received over the network.

A key challenge with marshalling/unmarshalling is in dealing with pointers. There are two ways to address these:

- Copy Value of Pointers: For simple situation, rather than copy the pointer, the RPC framework can copy the values that the pointer points to.
- Copy address of server: for complex situations with nested pointers. Rather than performing a recursive copy with may miss crucial information. Instead, the RPC framework passes the address of the server. When operations need to be performed, an RPC call is made in the reverse direction.

The network can fail in many different ways: it may drop packets, delay packets, reorder packets, and modify packets. In designing a distributed system, the programmer must be aware of and introduce mechanisms to tackle these challenges. Fortunately, certain transport layer protocols, e.g., TCP, provide mechanism to tackle these challenges. The key challenges arise when network and server failures are explored in coordination.

The key challenge arises when network failures examined in coordinate with server failures. In a distributed system, a client is often unable to distinguish between a network or a server failure. This is particularly troubling because transport protocols do not help with the server failures. In Figure 6, we examine several different scenario: (a) the network fails, (b) the server fails before receiving the request, (c) the server fails after receiving the request and processing the request but before respond.

In all three situations, the client does not get a response and can thus assume something as gone wrong; however, the client is unable to localize the problem to the network or server.

To tackle these problems, the client needs to retry the request and hope for a response on the retry. Alternatively, the client can keep retrying until it gets a response. A potential problem with this approach is that if the client and server are in situation (C), then the server will receive the requests each time, and process the requests but crash afterwards.
As a result, the request will be performed multiple times. For a class of requests, i.e., idempotent call, performing the call multiple times is okay. An example of such a call is "read()" or "get()" performing this call multiple times does not lead to a different state. However, "increment(Var_Name)" everytime it is called will lead to a different state.

For non-idempotent calls, server logic is required to help address these situations (c) – wherein the server has already processed an analyzed the request. In particular, more logic is required in the call is non-idempotent. This logic is required to ensure that the request is not processed twice by the server. To address this, we introduce two techniques: first, to log the Request-IDs of all processed requests and to also log the responses generated. Whenever a server gets a request, the server checks the RequestID to see if it is stored in its logs, then it replays the old response. Similarly, whenever a client needs to resend a request, it uses the same RequestID as the original.

To support a distributed system, the develop is required to handle all of the above each on every interaction between any two nodes. To overcome this challenge, instead frameworks exists, called RPC, that hide the complexity of failure handle by abstracting the communication between any two nodes as function calls between the two nodes. In addition to tackling failure modes, RPC must also tackle two challenges: (1) pointers, and (2) functions arguments. A simple way to tackle function arguments is to copy all the argument. However, pointers can not be copied over because pointer refer to addresses in local memory and on a different machine the contents of that address will be totally different. For simple pointers, the most intuitive approach is to copy the content of the pointer rather than the pointer. However, for complex pointer, i.e., nested pointers, the most appropriate way to send the address of the client. When memory needs to be accessed then the address is used to make a reverse call back.

RPC calls provide several different semantics. We list them in Table 2.3. Atleast once semantics are ideal of idempotent calls, when used with non-idempotent call, atleast once semantics will result in invalid state. The at most once semantics includes all of the techniques discussed in the last paragraph and this helps to ensure that the call is only made at most once. Although, exactly-once is the desired semantic. This is not always achievable. In particular, there are failure-modes that make it difficult for a system to respond to messages.

<table>
<thead>
<tr>
<th></th>
<th>Retries</th>
<th>Duplicate supression</th>
<th>Response Replay</th>
</tr>
</thead>
<tbody>
<tr>
<td>exactly-once</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>atleast-once</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atmost-once</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Semantics for RPCs.
3 Load Balancing

Load balancing is a key part of allowing multiple servers to coordinate and provide greater functionality than any individual node. If poorly implemented, load balancing can result in poor performance. We have all encountered flawed load balancing algorithms: during check-in for events, participants are often grouped by their name, i.e., [a-m] at one table and [n-z] at a separate table. This scheme does not work because names are not evenly distributed.

The key to a good load balancing algorithm is ensuring that load is evenly distributed across all servers. One way to do this is to use a random function or to assign work using a round-robin algorithm. These two algorithms work exceptionally well on average, however, they do assume that the servers or the request are identical. If servers or the requests are not identical, then random or round-robin will be unable to maintain perfect balance. A direct method to tackle this problem is to use weight-roundrobin or weighted-random either approach can tackle the intrinsic discrepancy between servers/requests.

One potential problem with random assignment is that when servers fail or when new servers get introduced then a large number of keys will change. For example, in Figure 3, we see a node being inserted. During insertion we observe that a large number of keys move between rows of the table.

**Question:** In a well balanced system with N nodes, and K keys, what is the maximum number of keys that should move when a new node is added or removed?

---

![Figure 3: Load balancing clients/keys to servers.](image)
4 Consistent Hashing

Consistent hashing has emerged as a promising path forward. The two key principles behind consistent hashing are:

- the keys are assigned fixed positions regardless of the number of servers in the system.
- the servers are assigned fixed locations.

The independent assignment of both keys and servers to fixed locations implies that as the number of servers changes, the keys position do not change. The only thing that changes is the new server being added or the old server being removed.

Given such a system, we consistently assign keys to servers by going clockwise within the space. Note: due to the use of "mod()", the id space wraps around.

For example, if server X is removed none of the keys change position. However, they are now reassigned to the next server "Z". When X is added back into the system it take over a subset of keys from Z.

Similar, should a new server "A" be added to the system between X and Z then it would take over a subset of the space owned by "X".

![Figure 4: Consistent hashing of servers into an ID space.](image)

Given a server space of 128 (7 bits) in Figure 6, servers have the following IDs: 11, 32, 45, 87, 111, and 126. ... Find an assignment of the following keys IDs to servers: 55, 200.

Consistent Hashing (cHash) is used in distributed key value stores such as Amazon’s DynamoDB, FB’s Cassandra.
4.1 Open Issues

- Heterogeneous Server: Some servers might have more resources than others. To deal with this imbalance we could give each server a proportional number of IPs. For example, a server double the normal resources, could have two IPs.

- Popular Content: Some keys might be more popular than others. To deal with the imbalance we could make more copies of the popular keys.

- Fault Tolerance: An interesting property of consistent hashing is that it minimizes the number of keys or clients that need to be moved between servers when a server fails or comes up. However consistent hashing doesn’t address the state associated with each client or the objects associated with each key. To address this problem, client state or key’s value are stored at multiple servers. The exact server is usually the next N servers going clockwise in the ring.

Methods for increasing the number of servers or keys:

- Salting: this method recreates multiple copies of an item by appending a predefined random sequence to the key. For example, to create multiple copies of ”brown”, we could salt it with either ”abc” or ”xyc”. To do this, we would prepend to the keyname, thus ”abcbrown”, then apply the regular hash function and mod the resulting hash value.

- Multiple keys: Another approach is to use multiple different hash functions. Instead of using just MD5 as discussed approach would could create a second copy by using SHA1.
5 Time

Today device employ a different protocol to remain in synch with each other. However, these protocols provide bounded accuracy. For example, a popular protocol, NTP, is known to provide limited accuracy. In certain systems this level of time precision is not sufficient. For higher levels of precision, atomic clocks and GPS are available however these require significant monetary investments. As a result these techniques are not widely used. Google has atomic clock, you can learn more about this in their Spanner paper.

A key observation made in the 1970s is that while humans care about time, distributed systems care about the ordering of events. Which event happens first? Which event is dependent on another event?

We define two broad types of ordering: total order and partial ordering.

- **Total Ordered:** In a total ordered system, all servers are able look at all events and order them in the same fashion regardless of the timestamp on the events. Note, in a total ordered system, events may be out of order with respect to the timestamp, what matters is that all servers agree on an ordering of the events.

- **Partial Ordered:** In a partial ordered system, all servers are only able to agree on the ordering of a subset of events. There are two popular partially ordered systems: client/FIFO and causal ordered. In a client/FIFO, the events have to be ordered according to the local timestamps of the server which will respect FIFO because time generally proceeds in a linear fashion.

Ideally, we want systems to be total ordered; however, this is challenging because it either requires all servers to be able to globally synchronized or to coordinate to determine a total ordering. This can be done either with globally synchronized clocks or a global sequencer. Both approaches are expensive in practice. The globally synchronized clocks requires the use of Atomic clocks which are expensive — Google uses them, but few other companies do. The global sequencer requires a centralized server which dispenses event-ids to the different servers. With a global sequencer, before each server can process an event, it consults with the global sequencer to get an ID for the event. Because the global sequence increments its ID after assigning, it’s ID are assigned incrementally and moreover because each server needs to get an ID before processing an event, then all events can be totally ordered based on these IDs. This global sequencer is a single point of failure and also introduces a performance bottleneck to the system. Yet, some systems still find this useful.

More practically, we want an ordering which needs to capture the causality and the causal relation between events: which event “happens before” the another event. Thus “logical clocks” and “vector clocks” were invented. Vector clocks are an advanced form of logical clocks which are able to capture more causality between events. Unlike logical clocks where-in there is a single clock, in vector clocks, there are a vector of clocks.

While vector clocks capture more information, they do this at a cost. In particular, there are two key overheads – first maintaining a large vector at each server takes up memory and second exchanging a large vector with each message incurs network overheads. Yet, vectors clocks are still used in large systems at Amazon and Facebook. Vector clocks can be made scalable by exploring a number of different approaches, e.g., Dotted Vector Clocks.
5.1 Bring it all together

To bring together consistent hashing, DNS, load-balancing and now Vector clocks. Let’s return to our shopping cart example:

- Initially to communicate with a server, your computer uses DNS to find one of the many load balancing server. Often times you are assigned the closest and least loaded server.

- This load balancer uses CHash to deterministically assign you to a web-server. You will login to this webserver and all of your future connections will go to this webserver. This webserver maintains some state for about your session and thus you should keep using it. An example of the state maintained is the vector clock. As you make changes to the database, these changes are assigned vector clocks.
• All your data (pictures, history, shopping carts) are stored in a distributed hash table (i.e., NoSQL data store). This distributed hash table may use vector clocks to maintain causality for some data.
6 Tapestry

- Tapestry Paper: [Here.]

In a traditional DHT (Distributed Hash table), there are two primary functions of the DHT protocol: locating keys and routing to keys. In the past, we’ve discussed locating keys – by using CHashing. We assumed that the client will have access to all serverID and could easily locate the server with the key and directly forward the request to the appropriate server. However, in many practical systems, the clients and servers only have access to a limited subset of the memberservers. In this situation, locating a becomes non-trivial. We will discuss tapestry a system that provides dynamic object location and routing within a highly dynamic peer-2-peer infrastructure.

Each tapestry node consists of four main components: an object store, a routing table, a list of backpointers, and a cache.

Each object has an ID, and each server has an ID – similar to consistent hashing. The “root” for ID is the server with the closest ID.

In tapestry, each client periodically “publishes” the object they own into the system. This ensure that if the servers fail, the objects are not lost. When servers fail, there is a temporarily window when the object is unavailable before the client republishes the object. When an object is published, this occurs in two stages, first the client tries to find the root of the object, then the server inserts the object into the object-store for the root server.

6.1 ID LookUp

To look up the root of an ID, Tapestry using prefix routing. In tapestry, the routing table is stored in a format to expediate routing. In particular, the routing table stores $B \times \log B N$ entries. Where B is the base representation for the table. In the lectures we will use Base 4. We will assume an ID space of 8 digits, i.e., 0 – 255, which in Base 4 becomes 0000 – 3333.

Note, the columns are based on the Base system. Each row will have B entries. There will be $\log B N$ rows. Thus the table is $B \times \log B N$. If you choose to keep c backup nodes for each entry then $c \times B \times \log B N$.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXXX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0XXX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03XX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>032X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.2 Removing Servers

A server can leave the DHT protocol in one of two ways: one it can exit unexpected or it exits gracefully. Regardless, there are two key items to be address: one updating routing tables of servers pointing to the exiting server and recovering objects stored on the exiting server.

In a graceful exit, the server does two things:
6.3 Adding Servers

- informs all the servers in its backpointers that it is leaving the network and provides each of this server with an appropriate replacement. Note: the Server A’s backpointers are a list of other servers which have server A in their routing table. Thus, when server A leaves the network, these werees will have incomplete routing table. Server A directly addresses this problem by notifying these servers and providing them with a suitable replacement.

- It detects the new root for all objects and moves the objects to the new root.

In an ungraceful exit, the servers in Server A’s backpointers will detect A’s departure through the use of heartbeats and will need to find a suitable replacement. The objects hosted on the server are temporarily lost until the client republishes them.

The default behavior is ungraceful exit. The graceful exist is an optimization which reduces the time to detect and recover objects.

6.3 Adding Servers

When adding a new server, we need to build up its routing table, back pointers list, and initial object list. Adding a new server occurs in several steps:

- LookUp the root.
- Get objects from the root
- Build a routing table using information from root
- Determine need-to-know nodes and fill in their tables

The Importance of Lookup: The Lookup function is a very critical function: it is used to add nodes, to add objects, to find objects. In particular, before an object can be added to Tapestry, we need to first find the root, this can be done using the LookUp function. Similarly, before a new node can be added we need to find its root. To find an object, we need to find the root – which is where the object is stored.

6.4 Centralized V. P2P DHTs:

We conclude by discussing the differences between a P2P-based DHT, e.g., Tapestry, and a cluster DHT, e.g., Cassandra or Dynamo. A key differences is that P2P systems are designed for nodes with little memory, e.g., IoT devices. Whereas cluster-DHT are designed for nodes with significant memory, e.g., data center clusters with over 64GB of memory. A network with 1 million nodes will require 16MB of memory which will easily fit within a server but may be challenging to fit within an IoT devices. The differences in memory changes the DHT design.
7 Replication and Consistency

Next, we discuss adding replication into our distributed system. Specifically, our distributed system aims to maintain N replicas of each object. The motivation behind maintaining N replicas is to overcome failures and potentially to provide better performance by load balancing requests across the different replicas. A key challenge that arises when multiple copies of an object exits is consistency. Essentially, ensuring that all replicas are treated in a consistent manner.

Different replication algorithms provide different levels on consistency. At one end of the spectrum, certain algorithms provide strong consistency which means that all accesses see the same sequence (e.g., used by traditional relation databases). The other provide eventual consistency (e.g., used by more recent NoSQL databases) which means that different (parallel) accesses can see objects in different orders. Eventual consistency provide the guarantee that if no changes are made to the system for a long enough time then parallel access will observe the same data.

To maintain consistency, we need to control and coordinate the ordering of events across all replicas. Thus, we need to control and coordinate the various attempts to read and write to the DHT. There are three main replication schemes and they each use a different ordering of events. The table highlights the different replication schemes and the ordering used.

<table>
<thead>
<tr>
<th>Replication Scheme</th>
<th>Active</th>
<th>Passive</th>
<th>Lazy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordering Technique</td>
<td>FIFO + Tie breaker</td>
<td>global sequencer</td>
<td>vector clocks</td>
</tr>
<tr>
<td>Consistency Guarantee</td>
<td>Strong</td>
<td>Strong</td>
<td>causal</td>
</tr>
</tbody>
</table>

7.1 Passive Replication

In this scheme, the replica servers are assigned roles. One RM server becomes the leader. The other N-1 servers become followers. All requests (both reads/writes) from all the clients go to the leader which

1. places the requests in a FIFO queue,

2. takes the request at the head of the queue. if this item is a write, the leader replicas to all the follower RMs,

3. only after all the follower RMs confirm replication, does the leader respond to the client and move unto the next item in the queue.

4. for reads, the leader responds directly.

Strong consistency is provided because the leader RM, (i) orders all events, (ii) processes events one at a time, and (iii) only moves unto the next event when the whole system has finished the event at the head of the queue.
There are different types of failures: to recover from failures, the client may need to retry requests, the leader may need to retry replication. How do you tackle duplication? or reordered requests?

Examples of passive replication schemes are: Raft, Zookeeper, Paxos, etcd, Chubby.

### 7.2 Active Replication

In this scheme, the client sends the request to all RM servers. All RM servers are an equal role. Upon receiving a message, the RM puts the message in a queues. The queue is not FIFO, instead the queue is ordered according to the logical clocks on the messages.

In the background, the RMs exchanges messages with each other to confirm that they have received the message. Each RM only processes the message at the head of the queue once all other RMs have confirmed to receiving this message.

While different from passive replication, active replication provides similar guarantees. By ordering by logical clocks, all RMs will be able to arrive at a similar ordering of events. The only exception is when there are ties, to break ties, the system uses client IP addresses. The constraints that each RM only processes the head of the queue once all other RMs have agreed ensures that all RMs agree on the head.

### 7.3 Analyzing Passive and Active Replication

We can consider the different protocol in terms of CPU and network overheads. The table below briefly discusses the different tasks that each node performs. This allows us to try to reason about the potential CPU loads and requirements.

<table>
<thead>
<tr>
<th>Passive Replication</th>
<th>Active Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leader</td>
<td>Follower</td>
</tr>
<tr>
<td>multicast to followers</td>
<td>data store</td>
</tr>
<tr>
<td>data store</td>
<td>leader election</td>
</tr>
<tr>
<td>queue maintenance</td>
<td>queue maintenance</td>
</tr>
<tr>
<td>Member</td>
<td></td>
</tr>
<tr>
<td>data store</td>
<td></td>
</tr>
</tbody>
</table>

### 7.4 Full Quorum Issues

To overcome N failures, usually you need N+1 RMs. However, a key issue with both AR (active replication) and PR (passive replication), is that it requires all RMs to agree before making forward progress. A single slow node can impact performance. Moreover, when a node fails, the system stalls.

Instead of requiring all RMs to agree, you can require only a majority (i.e., a quorum) of the RMs to agree. To allow for a majority and overcome N failures, then 2N+1 is required. Thus, if N fail, then N+1 are still available which is a majority.

### 7.5 Consistency Models

Total ordering is a very flexible ordering of events: it only requires that all servers agree on the order of the events. Any arbitrary ordering is okay, provided that all servers agree on the ordering.
There are several stronger forms of total ordering:

- **Sequential**: This version of total ordering requires that the events also be FIFO ordered.

- **Linearizability**: This version of total ordering is almost impossible to achieve in practice. This requires that all servers agree on an ordering and that the order should reflect real time (real time is a stronger form of FIFO).

The Venn diagram (below) discusses the difference between Total ordering, Sequential and Linearizability. Note while Sequential and Linearizability are both subsets of total ordering, Linearizability is a subset of Sequential, thus Linearizability is the strongest. You get a version of "linearizability" from Raft and passive replication.

![Venn Diagram](Figure_6.png)

**Figure 6: Variants of Total Ordering.**

**Eventual consistency makes no guarantees**: the only guarantee is that given enough time if no values are changed then all replica managers will agree on a total ordering of events.

### 7.6 Raft

- **Raft Paper**: [Here.](#)
- **Raft Slides**: [Here.](#)