Section 13: Distributed Systems and Networking

December 4-6, 2019

Contents

1 Vocabulary

- **Replication** A strategy for fault tolerance. With replication, one or more **replicas** are responsible for trying to maintain the same state.
- **Primary/Secondary** or **Coordinator/Worker**. A scheme for separation of responsibilities. In this scheme, there is typically a single active primary and one or more secondaries. The primary is typically responsible for being the source of truth and directing operations towards the secondaries. This scheme is also sometimes called Master/Slave.
- **Failover** A fault tolerance procedure invoked when a component fails. Typically this involves switching over to, or promoting a secondary.
- **2PC** - Two Phase Commit (2PC) is an algorithm that coordinates transactions between one coordinator (Master) and many slaves. Transactions that change the state of the slave are considered 2PC transactions and must be logged and tracked according to the 2PC algorithm. 2PC ensures atomicity and durability by ensuring that a write happens across ALL replicas or NONE of them. The replication factor indicates how many different workers a particular entry is copied among. The sequence of message passing is as follows:

```
for every worker replica and an ACTION from the master,
origin [MESSAGE] -> dest :
---
MASTER [VOTE-REQUEST(ACTION)] -> WORKER
WORKER [VOTE-ABORT/COMMIT] -> MASTER
MASTER [GLOBAL-COMMIT/ABORT] -> WORKER
WORKER [ACK] -> MASTER
```

  If at least one worker votes to abort, the master sends a GLOBAL-ABORT. If all worker vote to commit, the master sends GLOBAL-COMMIT. Whenever a master receives a response from a slave, it may assume that the previous request has been recognized and committed to log and is therefore fault tolerant. (If the master receives a VOTE, the master can assume that the worker has logged the action it is voting on. If the master receives an ACK for a GLOBAL-COMMIT, it can assume that action has been executed, saved, and logged such that it will remain consistent even if the worker dies and rebuilds.)
- **TCP** - Transmission Control Protocol (TCP) is a common L4 (transport layer) protocol that guarantees reliable in-order delivery. In-order delivery is accomplished through the use of sequence numbers attached to every data packet, and reliable delivery is accomplished through the use of ACKs (acknowledgements).

2 Distributed Key-Value Stores (Part 1)

a) Consider a distributed key-value store using a directory-based architecture.

Keys are 256 bytes, values are 128 MiB, each machine in the cluster has a 8 GiB/s network connection, and the client has a unlimited amount of bandwidth. The RTT between the directory and data machines is 2ms and the RTT between the client and directory/data nodes is 64ms.

i) How long would it take to execute a single GET request using a recursive query?

There would be 66 ms of latency + $\frac{2^{27}}{2^{33}} = 0.0156$ seconds of transfer time. The total time would be 82ms.

ii) How long would it take to execute 2048 GET requests using recursive queries?

Assuming we are able to carefully implement pipeline parallelism, we would still have 66 ms of latency. The transfer time would now be $\frac{2^{11} \times 2^{27}}{2^{33}} = 2^5 = 32$ seconds.

iii) How long would it take to execute a single GET request using an iterative query?

There would be 128 ms of latency and $\frac{2^{27}}{2^{33}} = 0.0156$ seconds of transfer time so the total time would be 143ms.

iv) How long would it take to execute 2048 GET requests using an iterative query?

Assuming we can take advantage of pipeline parallelism for resolving nodes, it would take 64 ms + $\frac{2^{11} \times 2^{8}}{2^{33}}$ seconds to resolve all the keys.

We assume each data request can be executed in parallel, so it would take 64 ms + $\frac{2^{27}}{2^{33}} = 0.0156$ seconds to transfer all the data.

This is a total of 143 ms. Notice that in the recursive query case, the directory server was a much larger bottleneck.

Note it's reasonable to assume that we can execute our requests in parallel because we can use a hash function to effectively map a key to a uniform random address in our hash table (assuming a non adversarial client). We are also asssuming that there are a large number of nodes, so no single node is limited by bandwidth.

Here's a proof that our objects should be conveniently distributed across our nodes: https://inst.eecs.berkeley.edu/

Pay special attention to the assumptions made, and whether or not they make sense in practice!

Also note that the peak bandwidth in this scenario exceeds 1TiB/s, which would require careful client design.

v) Now imagine our client is located in the same datacenter, and the RTT between all components is the same (this is a common assumption when modeling datacenter topology).

Briefly describe how your results would change.

In broad terms, the RTT latency would now become far smaller than the actual transfer time, removing the previous bottleneck on latency for small transfers.

From a performance perspective, it is almost strictly better to use iterative querying under these assumptions about latency.

Note that there could still be other reasons to use a recursive query strategy even under these conditions. For example, one could try to simplify their design, ensure ordering/timestamp their outputs, aggregate the data, etc, but perhaps this provides insight into why iterative

querying is a popular design for DHT's within datacenters (such as GFS and the many systems based on it).

vi) What are some advantages and disadvantages to using a recursive query system?

Advantages: Faster, easier to maintain consistency.
Disadvantages: Scalability bottleneck at the directory/master server.

vii) What are some advantages and disadvantages to using an iterative query system?

Advantages: More scalable.
Disadvantages: Slower, harder to maintain consistency.

b) **Quorum consensus:** Consider a fault-tolerant distributed key-value store where each piece of data is replicated N times. If we optimistically return from a put() call as soon as we have received acknowledgements from W replicas, how many replicas must we wait for a response from in a get() query in order to guarantee consistency?

We must wait for at least $R > N - W$ responses. If we have any fewer than this number, there is a possibility that none of our responses contain the latest value for the key we are requesting.

c) In a distributed key-value store, we need some way of hashing our keys in order to roughly evenly distribute them across our servers. A simple way to do this is to assign key K to server i such that $i = \text{hash}(K) \mod N$, where N is the number of servers we have. However, this scheme runs into an issue when N changes — for example, when expanding our cluster or when machines go down. We would have to re-shuffle all the objects in our system to new servers, flooding all of our servers with a massive amount of requests and causing disastrous slowdown. Propose a hashing scheme (just an idea is fine) that minimizes this problem.

We can treat the possible hash space as a circle, where every possible hash maps to some point on the circle. We then roughly evenly distribute our servers across this circle, and have each hash be stored on the next closest server on the circle. Then, when we add or remove servers, we need only move a portion of the objects on one server adjacent to the server we just added or removed. This technique is commonly known as **consistent hashing**.

3 Distributed Key Value Stores (Part 2)

Consider a distributed key value store, in which for each KV pair, that pair is stored on a single node machine, and we use iterative querying (this is essentially what we looked at in the previous DHT problem).

1. Describe some limitations of this system. In particular, focus on bandwidth and durability.

Bandwidth: This system could still be bottlenecked by bandwidth. If there is a popular key in the store, all reads/writes to that key will be directed to the same machine, and thus be limited by the bandwidth of a single machine.

Durability: Writes to this system are only durable, so long as the machine backing the node does not fail. If this machine is backed by RAID then it may have fault tolerance with respect to a disk failing, but if a flood damages the entire machine, data written to that node will be

lost.

2. Propose some strategies for overcoming these limitations.

One strategy for overcoming these limitations is replication.

If multiple replicas contain the same data, the directory server, or a load balancer, could direct GET requests to any machine which contains the data, thus the bandwidth of the node is now scaled by the number of replicas.

Note that this replication now introduces new challenges, such as consistency and consensus.

3.1 Fault tolerant nodes

For the remainder of this problem assume that nodes are backed by multiple replicas and each replica is capable of processing multiple transactions concurrently.

Also assume that each node supports the following *local* operations.

- GET(key) - If the machine contains the corresponding KV pair, return the value, otherwise gracefully return NULL.
- PUT(key, value) - Adds a KV pair to the machine if it does not already exist. If the machine contains the key, but a different value, abort/fail.

1. In order to achieve fault tolerance, we want to ensure that our system is available if a machine fails. Describe a few ways in which replication can be used.

Quorum Consensus: One possibility is to use a quorum consensus protocol for PUT requests. In order to ensure that our writes are durable in the face of machine failure, we should ensure that we write to W machines before returning success. If W ¿ $N/2$ there will be no write conflicts and the system will be eventually consistent.

GET requests can be routed to any machine that is part of the node.

Two Phase Commit: Another possibility is to use two phase commit. Here we would pick a coordinator a follow the two phase commit protocol. This system would have strong consistency.

Again, GET requests can be routed to any machine that is part of the node.

2. In the case of quorum consensus think about how queries will be executed and how much bandwidth will be required. We conveniently provide you with some space to write down your thoughts.

Here we need some routine for writing data to multiple nodes. We provide analysis regarding a few options.

Quorum Consensus: One approach here could be to modify the client to write to all W client machines. This would require writing a total of $W * S$ bytes where S is the size of the request. The client must split its bandwidth across W nodes here, so the write will take $\frac{B}{W*S}$

We could also pick a single machine as the coordinator, then send our query to the coordinator. The coordinator could pick W machines to send the query to. If the client far away, this would have the advantage of decreasing the bandwidth between the client and DHT. Typically this "gateway" bandwidth is significantly smaller than the internal bandwidth of the datacenter.

This part is out of scope, but interesting and common in the real world

Alternatively we could take send the data to a single machine, which would forward the data to another machine. Here we transfer the same amount of data, but we can take advantage of

the full inbound and outbound bandwidth of all of our machines. Using pipeline parallelism this would take $\frac{B}{S}$ time with some additional overhead latency (which can be minimized by forwarding to the nearest nodes). For more details see section 3.2 of The Google File System (https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf).

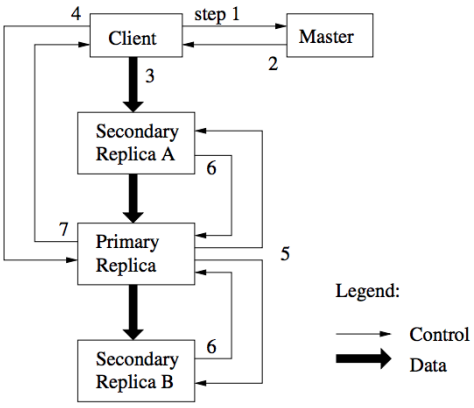


Figure 2: Write Control and Data Flow

3. When might this system be in an inconsistent state?

Consider the case in which a client calls PUT. The KV pair will be stored on W machines. Now suppose another client calls GET on the same key very shortly after. If the request is routed to a different machine (not one of the W machines), it will return that the key does not exist. This is inconsistent because we just stored the key.

3.2 Two-Phase Commit

Quorum consensus is able to provide weak consistency. For this section, assume the DHT backs each node with multiple replicas. Further, assume that these machines use a two phase commit protocol to commit information in a strongly consistent manner.

1. 2PC requires a single coordinator and at least one worker. By default, all replicas can be workers. How should the coordinator be picked?

The general problem here is leader election - the bootstrap process of choosing and agreeing a leader in distributed systems.

Naively, we could pick a single machine to be the coordinator for all transactions and hard code that machine as the leader. If the leader crashes, then we must wait for that leader to restart before the system can make any forward progress.

Modern leader election algorithms do not rely on a hard coded leader but rather allow the workers to choose who they want as their leader. In the event that the leader fails (as detected by some form of health checking), the subset of the machines that are still connected would rerun the algorithm and find a new leader.

This provides better availability because a new leader can be choose on the fly, but it also introduces further complexities. For example what if there was a network partition and a single cluster of works split into two where each one elects their own leader. Then the partition heals how do you deal with the fact that there are now two leaders.

Additionally in any situation where we choose a single leader, we would be bottle necked by its upload bandwidth, since it would have to send the request to all replicas. To mitigate this problem, we could pick a leader at random for each request. This would spread resource usage for the dominant resource (upload bandwidth).

Note we typically don't consider consider GFS style chaining. It doesn't fit in the byzantine generals model, and may not be fast if we can't do a shortest hop tour (in the event of a failed machine).

2. Briefly describe the messages that the **coordinator** will send and receive in response to a PUT. Also describe when and what would be logged.

   - RECEIVE (from client): A PUT query
   - LOG (to journal): PREPARE query
   - SEND (to workers): N PREPARE PUT messages
   - RECEIVE (from workers): COMMIT or ABORT per worker
   - LOG (to journal): GLOBAL-COMMIT or GLOBAL-ABORT
   - SEND (to workers): GLOBAL-COMMIT if it receives COMMIT from all workers, otherwise GLOBAL-ABORT
   - RECEIVE (from workers): ACK
   - SEND (to client): SUCCESS if it sent GLOBAL-COMMIT, otherwise FAIL.

3. Briefly describe the messages that a **worker** will send and receive.

   - RECEIVE (from coordinator): A PREPARE PUT
   - LOG (to journal): PREPARE query
   - SEND (to coordinator): COMMIT or ABORT
   - RECEIVE (from coordinator): GLOBAL-COMMIT or GLOBAL-ABORT
   - LOG (to journal): GLOBAL-COMMIT or GLOBAL-ABORT
   - SEND (to coordinator): ACK

4. Under the current model, multiple PUT queries could be sent to separate coordinator machines. Propose set of worker routines which can handle this. In particular, consider the case in which 2 coordinators receive PUT queries for **the same key but different values**. The state of the system should remain consistent.

   Upon receiving a PREPARE PUT request, `try_acquire` a lock on the key.

   If acquire fails, ABORT, otherwise COMMIT.

   Upon receiving a GLOBAL-COMMIT, set the value then release the lock. Release the lock without setting the value on a GLOBAL-ABORT.

   The state of the system will always be consistent now. Note that a potentially unintuitive outcome of 2PC here is that both queries could fail here, leaving the key empty despite a PUT request.

   One potential solution to this problem (which is out of scope) while maintaining strong consistency could include using PAXOS to determine an order of these queries, and accept only the first.

Note: Many popular DHTs in the real world take different approaches to handling this edge case. Most of these DHTs are eventually consistent.

4 Networking

a) (True/False) IPv4 can support up to 2^{64} different hosts.

False, it has 32 bits per IP address

b) (True/False) Port numbers are in the IP header.

False, they are in the transport layer. (TCP/UDP).

c) (True/False) UDP has a built in abstraction for sending packets in an in order fashion.

False, this is a part of the TCP protocol. In UDP there is no notion of a sequence number.

d) (True/False) TCP provide a reliable and ordered byte stream abstraction to networking.

True.

e) (True/False) TCP attempts to solve the congestion control problem by adjusting the sending window when packets are dropped.

True.

f) In TCP, how do we achieve logically ordered packets despite the out of order delivery of the physical reality? What field of the TCP packet is used for this?

The seqno field.

g) Describe how a client opens a TCP connection with the server. Elaborate on how the sequence number is initially chosen.

3 way handshake. Client sends a random sequence number (x) in a syn packet. Server sees this and sends another random seqeuence number back (y) in addition to acknowledging the sequence number that it received from the client (sends back x+1) in a syn-ack packet. Client acknowledges this sequence number in an ack packet by sending back y+1.

It is important to randomize the sequence number so an off path attacker cannot guess it and send spurious packets to you.

h) Describe the semantics of the acknowledgement field and also the window field in a TCP ack.

The acknowledgement field says that the receiver has received all bytes up until that number (x). The window field says how many additional bytes past x the receiver is ready to receive.

i) List the 5 layers specified in the TCP/IP model. Layering adds modularity to the internet and allows innovation to happen at all layers largely in parallel. What is the function of each layer?

1) Physical Layer: the physical layer is responsible for the delivery of raw bits from one endpoint to another. It includes ethernet, fiber, and other mediums of data transmission.
2) Datalink Layer: this layer adds the packet abstraction and is responsible for the local delivery of packets between devices within the same network (usually a LAN but can also include WANs). Devices here include switches, bridges, and network interface cards (NICs).
3) Network Layer: this layer is the skinny waist of the internet and routing algorithms here only speak IP. The network layer is responsible for global delivery of packets across one or more

networks.

4) Transport Layer: this layer provides host to host or end to end communication. Because processes operate in terms of streams of data rather than individual packets, the transport layer handles the multiplexing of packets into streams, end to end reliable delivery, and introduces the concept of flows. This layer lives in the operating system, and TCP and UDP are examples of popular transport layer protocols.
5) Application Layer: the application layer provides network support for applications. The protocols that live here include: HTTP, FTP, DNS, SSH, TLS, etc.

j) The end to end principle is one of the most famed design principles in all of engineering. It argues that functionality should **only** be placed in the network if certain conditions are met. Otherwise, they should be implemented in the end hosts. These conditions are:

- Only If Sufficient: Don't implement a function in the network unless it can be completely implemented at this level.
- Only If Necessary: Don't implement anything in the network that can be implemented correctly by the hosts.
- Only If Useful: If hosts can implement functionality correctly, implement it in the network only as a performance enhancement.

Take for example the concept of reliability: making all efforts to ensure that a packet sent is not lost or corrupted and is indeed received by the other end. Using each of the three criteria, argue if reliability should be implemented in the network.

i) Only If Sufficient

NO. It is not sufficient to implement reliability in the network. The argument here is that a network element can misbehave (i.e. forwards a packet and then forget about it, thus not making sure if the packet was received on the other side). Thus the end hosts still need to implement reliability, so it is not sufficient to just have it in the network.

ii) Only If Necessary

NO. Reliability can be implemented fully in the end hosts, so it is not necessary to have to implement it in the network.

iii) Only If Useful

Sometimes. Under circumstances like extremely lossy links, it may be beneficial to implement it in the network. Lets say a packet crosses 5 links and each link has a 50% chance of losing the packet. Each link takes 1 ms to cross and there is an magic oracle tells the sender the packet was lost. The probability that a packet will successfully cross all 5 links in one go is $(1/2)^5 = 3.125\%$. This means the end hosts need to try 32 times before it expects to see the packet make it through, taking up to # of tries $\times$ max # of links per try $= 32 \times 5 =$ 160 ms. Likewise at each hop, if the router itself is responsible for making sure the packet made it to the next router, each router would know if the packet was dropped on the link to the next router. Thus each router only has to send the packet until it reaches the next router, which will be twice on average. So to send this packet, it will take on average # of tries per link # number of links $= 2 \times 5 = 10$ ms. This is a huge boost in performance, which makes it useful to implement reliability in the network under some cases.

5 Virtual Machines, Containers, and Orchestration

a) What happens when an application running on the Guest OS issues a system call?

The system call is forwarded to the VMM (since the host knows that a Guest OS application is running on that core), which forwards it to the Guest OS transparently.

b) What happens when the Guest OS tries to execute a privileged instruction?

The guest traps to the Host OS, which performs the privileged instruction (modified to operate on the Guest OS' state) on behalf of the guest.

c) Which page table is used for address translation when an application running on the Guest OS is executing?

A shadow page table, maintained by the VMM, is used for address translation.

d) Explain what a container orchestrator does and why its necessary.

(a) Provides a clean, easy to use virtualization of your data center resources
(b) It manages the sharing of these resources between different applications and users
(c) It provides common services such as load balancing, authentication, service discovery, etc
(d) It manages the complexity of deploying and running containerized microservice applications

e) What does Mesos (without additional libraries) offer that is similar to modern container orchestrators? How is it different from an orchestrator the likes of Kubernetes?

Mesos provides data center resource virtualization and can be thought of as an operating system for the data center / cloud. Its different because it concerned primary with resource allocation management rather than the container and service lifecycles, so it doesn't do as much to manage the complexity of deploying and running microservice applications.

f) What does Mesos (without additional libraries) offer that is similar to modern container orchestrators? How is it different from an orchestrator the likes of Kubernetes?

Mesos provides data center resource virtualization and can be thought of as an operating system for the data center / cloud. Its different because it concerned primary with resource allocation management rather than the container and service lifecycles, so it doesn't do as much to manage the complexity of deploying and running microservice applications.

g) Explain why reconciliation is a useful method of desired state management.

Reconciliation is an example of eventual consistency. Its useful because we can make changes the the central desired state, and that change will get propagated through reconciliation to those who care about it. It makes managing the desired state much easier because the responsibility of making sure the actual state is valid then falls on the local actor rather than the user making the change. In Kubernetes, changes to the desired state are done declaratively by the user using high level concepts like Deployments and Services. Through reconcilation, the Kubernetes components responsible will update the actual state to match what is desired by the user.

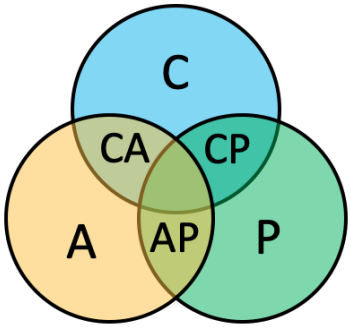
h) Explain what each of the following Kubernetes objects are: Node, Pod, Deployment, Service, Ingress

- Node: an object that represents a virtual or physical machine where containers can be deployed. A Kubelet sits on each Node.
- Pod: the unit of execution, a small group of one or more containers that is always deployed together
- Deployment: an object that creates a number of replicas for a given Pod template
- Service: an onject that most closely represents a microservice. Has a persistent abstract name and load balances to all Pods registered under it.
- Ingress: an endpoint where outside traffic is allowed to enter your cluster, responsible for load balancing and routing

6 CAP Theorem

This question is not in scope, but may be an interesting concept to think about.

One of the most famous and relevant distributed systems concepts today is the CAP theorem, first conjectured by Professor Eric Brewer (Berkeley) in 1998. It states that it is impossible for a distributed system to simultaneously have all three of the following properties:

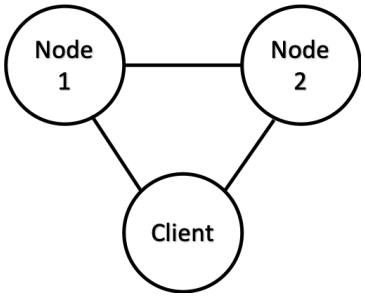


Consistency: every read must be correct, and so can either return the most recent write or an error
Availability: every request must not return an error, but read requests do not necessarily have to return the most recent write
Partition Tolerance: the system must be able to continue to operate despite an arbitrary number of messages between nodes dropped or delayed, such as in the event of a complete network partition.

Many systems are architected to have two of the three properties, and some even have just one. For example, a system that is Consistent and Available (CA) must always return the most recent write for every read, without fail.

Because of the nature of distributed systems, network partitions are considered an inevitable fact of life. Thus many distributed applications are forced to be either CP or AP, forfeiting either C or A in favor of partition tolerance.

a) The proof for the CAP theorem, while conceptually simple, was not formulated until 2002 by Gilbert and Lynch (MIT). Imagine you had a distributed database with two nodes, and a client that can read or write to either of the nodes.



Intuitively, argue why this system cannot achieve all three CAP properties.

For the system to have partition tolerance, then it must exhibit consistency and availability even if the link between Node 1 and Node 2 fails. With the link removed, the client can make a write request for some key K to Node 1. Then, it makes a read request for K at Node 2. Because the link between Node 1 and Node 2 is gone, Node 1 has no way to communicating to Node 2 about the value of K. Thus Node 2 must either return an error (because it knows somehow that it may not have the most up to date value) or it can return an older version of K. This violates either availability in the former case and consistency in the latter case. By contradiction, this system cannot be CAP.

b) Key value stores sometimes use Two Phase Commit to update the value of its keys. Which CAP properties does the 2PC protocol exhibit?

In the event of a network partition, such as if some subset of nodes cannot be reached or have failed, 2PC simply blocks until timeout and then sends out global abort messages. Therefor it cannot produce meaningful work in the event of a partition, and thus does not hold the P property. It does however hold C because consistency is the whole point of having 2PC. For availability, we look at 2PC under normal circumstances (since we already ruled that it does not have P). 2PC requests are simply forwarded to the workers, which by the fact that there are no failures, will return correctly and therefore giving Two Phase Commit A.

c) The consistency defined for the CAP theorem refers to *strong* consistency, of which there are two variants:

- Linearizable consistency: all operations appear to have executed atomically in an order that is consistent with the global real-time ordering of operations.
- Sequential consistency: operations can be reordered, but this new order must be consistent across all nodes.

There are also variants of *weak* consistency. A famous one is eventual consistency:

- Eventual consistency: if no new updates are made to a key, then eventually all nodes will agree on the last value for that key.

Is it possible to achieve all three CAP properties if C referred to weak consistency instead of strong consistency?

The reason why an AP system cannot also be strongly consistent is that after an update for a particular key, one partition will always have a stale value for that key since the new value cannot be propagated to that partition. However if we can assume that after some time the network partition will be resolved, then it is possible for an AP system to have eventual consistency. Cassandra is a KV store that is an example of an AP system with eventual consistency.