LOVE FOR PROGRAMMING

Distributed Systems Part-2: Consistency versus Availability, A Pragmatic Example!

Pawan Bhadauria

As we discussed in Distributed Systems Part-1: A peek into consistent hashing!, of this series, it is difficult to get distributed system work perfectly, all the time. Though, it has self healing capabilities but sometimes we might have to tradeoffs certain desired qualities for efficiency & scalability. We will talk more about those in this post. Also, we will try to use our distributed system developed in Part-1, for tackling other important business use cases. So lets get started.

Recall, using consistent hashing, how you had developed a system which provided both load balancing and fault tolerance. You had key-values spread across nodes for distributing load. For achieving fault tolerance, you also replicated a key-value to the predecessor node on consistent hashing ring. This had worked great for you till now. You feel on top of the world.

You manager is pretty impressed with you. Seeing your system scale well with consistent hashing, he wants to implement a critical part of application using your distributed system. With both mobile and desktop clients, your app has been growing rapidly. He wants a blazingly fast experience for customers who add items to shopping carts which forms a critical part of business. He wants you to utilize your 20 node cluster for storing customer shopping cart data.

You are off course excited about it, aren't you? But being burnt by earlier mistakes, you are more mature now. There are many questions popping in your mind but for now, you ask him three pertinent ones:

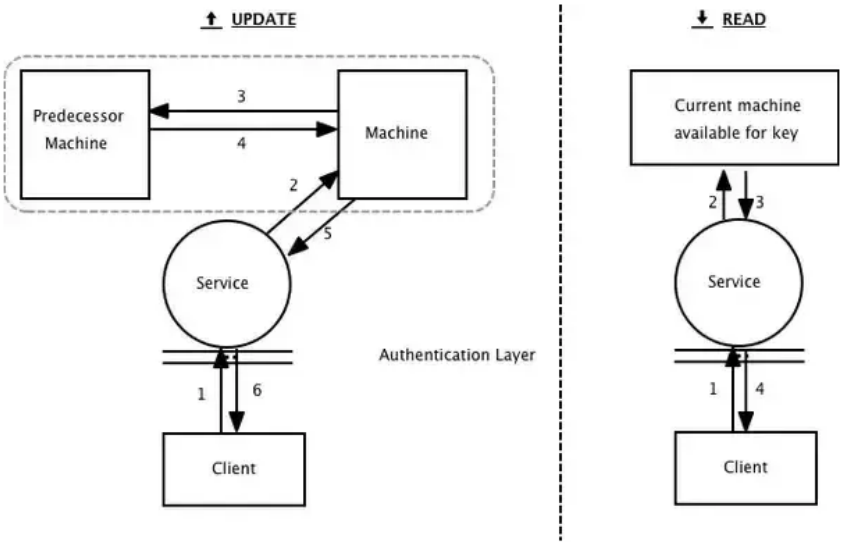
1. *How many registered users do we currently have & how fast are they growing?*
2. *What is the response latency he is expecting?*
3. *Who is the consumer of the service and what operations the service is required to support?*

[**Food for thought**: *If you look at above questions, you are asking about data size, operations to support & expected response time. These are the primary questions you should ask before coding any algorithm, apart from few other ones*]

He has no idea about couple of those. He needs time & vanishes away. Peace! The questions above, don't stop you from visualizing the initial design of the system. You understand what he is looking for. So you start thinking. To simplify design for first phase, you think about five operations which your service may provide:

1. *Get shopping cart items for customer.*
2. *Add item/items to shopping cart for a customer [create shopping cart if one doesn't exist]*
3. *Delete item/items from shopping cart for a customer.*
4. *Update item/items in shopping cart for a customer.*

These are traditional CRUD operations on shopping cart. Since the data is much more critical now, you need to be careful. So you start thinking about the user experience and the associated workflow. You start visualizing about storing shopping cart details against a customer id on various nodes. As and when, a new shopping cart operation comes for a particular customer id, you will use your proven consistent hashing algorithm to navigate request to the associated node, where you store that customer shopping cart details, and create/update/delete results. *Before the node modifies & persists the shopping cart data, it will ask predecessor node to modify the details as well.* Once everything is done, the call returns back to client via service. So your service's simple user workflow might look something like this (In reality there might be several other component but lets keep it simple for now):



While you are thinking about it, your manager returns back with answers to the question you had posed. Quick turnaround!!

1. How many registered users do we currently have & how fast are they growing?
[Manager] We have 10 million registered users as of now and we are adding roughly 1 million users every month. This rate will only get better moving forward.

2. What is the response latency he is expecting?
[Manager] Couple of seconds.

3. Who is the consumer of the service and what operations the service is required to support?
[Manager] Consumers can be both internal or external. We need to support all CRUD operations.

Frankly speaking, you already had an idea about all those, didn't you? You do some quick back of the envelop calculations for user numbers. If you have replication factor of 2 (10*2 = 20 million records), you will have data for roughly 1 million customer records on each node (20 node with replication) if distribution is even. This will grow by 100, 000 customer records per month on each node as of now. You look at the data size & feel that this looks quite manageable for now but can increase drastically if your app gains further traction. If you have data imbalance on any node, virtual nodes will help. Second answer is typical of a manager which means we should be as fast as possible.

Third one, apart from CRUD operations which you had already visualized, is a little thought provoking. You decode his third answer. You think that probably company

internal and external clients can use it. Great. Coming up with URLs for operations on a customer shopping cart resource is pretty easy. So here you are:

Base URL: api/v1/customers/{customer-id}/carts [you only have one cart associated with user for now]

GET [get cart details]
api/v1/customers/{customer-id}/carts/1

POST [add new item(s) to cart, create cart if none exist]
api/v1/customers/{customer-id}/carts/1
{ new item(s) payload }

DELETE [delete an item from cart]
api/v1/customers/{customer-id}/carts/1/items/{item-id}

UPDATE [update item(s) in cart]
api/v1/customers/{customer-id}/carts/1
{ update item(s) payload }

DELETE [delete the cart]
api/v1/customers/{customer-id}/carts/1

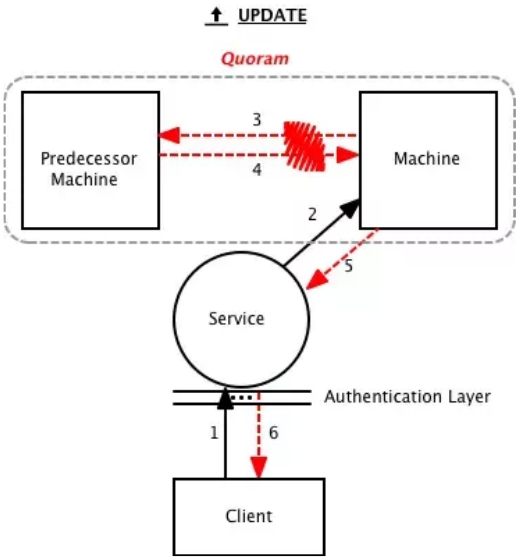
[We will talk more about REST is some other posts].

Back to problem. So you just designed your REST interface and supported operations. You go full speed on implementation. You design the shopping cart service using REST. At the back-end, you use your existing infrastructure. Your daemon service on each node gets the request, processes it (create/update/read/delete) and communicates with predecessor node to update its data. Once done, it returns the call back to service. Since the data for each customer has small footprint on each node (1.1 million customer records with replication as of now), the speed is stunning. You utilize the existing authentication & authorization infrastructure to wrap your REST service for making it accessible to both internal and external clients. Great loosely coupled design. You test your code & show it to your manager.

Your manager is having less and less questions for you now. He is glad. After testing some more use cases, you push your code to production. Everything works like a charm. Its an amazing feeling to see your code work in production!

One day in the morning, your manager hurriedly comes to you. He has received few complaints from customer support. Some customers have complained that they are getting "Your request cannot be processed now" error, once in a while. Its annoying and screwing up the user experience. You ask him for sometime to investigate the issue. He nods and vanishes away. Peace!

You look at the code & dig throw the log files to see what's going wrong. After a while, you find issue in Step-3.



You find that when a node gets a request for updating shopping cart data, it first tries to co-ordinate with predecessor node to persist the change and once that is successful, it commits the change to local storage and returns the call. If there is an issue with communication with predecessor node, it throws an exception. You did this because you wanted to make sure that your cluster should be fault tolerant with replicated data on at least one other node. In this particular case, the node is not able to communicate with its predecessor, so it is throwing an error since it can't compromise on consistency in case of failure.

In technical terms, you are writing data using a Quorum [Quorum (distributed computing)] with at least two votes. This falls in broader category of consensus based protocol like Paxos, [Paxos (computer science)]. Traditional RDBMS use a stringent consensus based protocol which is often called as 2-phase commit. Here is formal definition of quorum from Wikipedia:

"A quorum is the minimum number of votes that a distributed transaction has to obtain in order to be allowed to perform an operation in a distributed system. A quorum-based technique is implemented to enforce consistent operation in a distributed system."

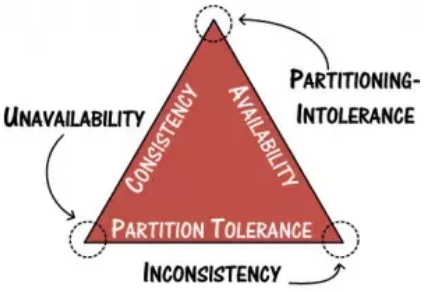
Anyways, while writing code, you never thought that the execution flow will ever go to this block but here you are. Everything is possible in production with distributed systems, mark my words. You take a deep breadth. In technical terms, you chose "consistency" over "availability" when an update couldn't have a quorum.

In retrospect, you think that you should have probably chosen "availability" as it is important for user experience. So if a request comes to a node, you could *persist the data to local storage first* & then try to communicate with the predecessor node to store the replicated data. If predecessor node is not available, you could return call with success since you have data on at least one node. Some obvious issues stare at your face. What happens if your current node goes down after writing data? You feel that, you are trying to choose "availability" now without worrying about "consistency" which will bring in its own set of challenges. You look concerned. You approach your nerdy friend who works at a fast growing internet consumer company & explain your situation. He understand the problem and tells you to look at CAP theorem. You search it on internet and get the following from Wikipedia:

The CAP theorem, also known as Brewer's theorem, ***states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:***

Consistency *- All nodes see the same data at the same time*

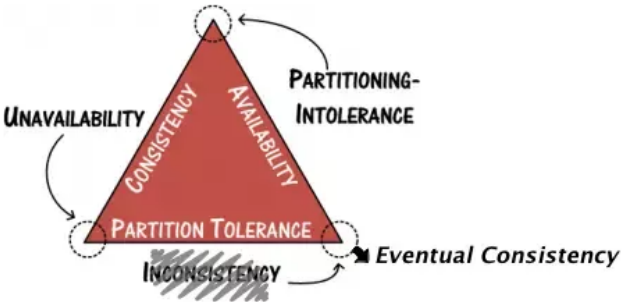
Partition tolerance - The system continues to operate despite arbitrary message loss or failure of part of the system.



As shown in above diagram, you can choose only one vertex in above triangle. You will get properties of the sides converging on that vertex, leaving out the property associated with opposite side.

You can relate to it. Since you can't compromise on partition tolerance (system can't go down if a part of system fails or is partitioned), it seems you need to make a choice between first two, availability & consistency. Having worked on RDBMS for a long time, you never in your dream, thought that one day you will need to make a choice between "availability" & "consistency". Its time to talk to your manager.

You explain him the situation with reference to CAP theorem & tell him that he needs to make a decision between "availability " & "consistency". He understands, thinks about it and tells you that availability is his choice but needs more time to ponder over it. Before vanishing away, he poses a question, "Will the system be inconsistent forever or is there a way to eventually make it consistent?". He is ready to tradeoff small inconsistency window for "availability" provided that system becomes eventually consistent in future. He thinks that he will get a highly available system with partition tolerance that provides eventual consistency. Smart chap! You ask him for sometime to think about it. He nods and vanishes away. Peace!!



You like his idea. One question which lingers in your mind is, "How long will the system remain inconsistent?". This window should be as small as possible, probably couple of seconds. Also if customer data is not modified during this "inconsistent window", everything should be fine since you don't have to deal with conflicting changes. But if customer makes a change during that window, you will have to resolve conflict which will make your job difficult. Before you think more about it, your manager arrives. He pulls out a list of questions and writes all five on the board:

1. If quorum based approach is problematic, can we just have node write to local storage, hand the data to a background process for replicating to other nodes & return call back to customer?
2. If writes are done in quorum, are we doing reads in quorum as well? If not, what is its impact on data consistency?
3. Lets say if we synchronize data in background, what might be the worst case latency?

5. If we follow an asynchronous replication mechanism, how would we handle data conflicts?

He tells you that if you can convincingly answer all five questions, it will help him understand both system and proposed "eventual consistency" model better. Also this will help him make better decision. Frankly, you don't know the answer to those questions yourself. But lets take one question at a time. You give it a shot:

1. If quorum based approach is problematic, can we just have node write to local storage, hand the data to a background process for replicating to other nodes & return call back to customer?
[You] Yes, we can do that. This will obviously create a inconsistent window before the data gets replicated on all nodes but we can make the window as small as possible. As long as customer doesn't update data during this "inconsistent window", all replicas will converge towards identical copies.

2. If writes are done in quorum, are we doing reads in quorum as well? If not, what is its impact on data consistency for user?
[You] We are not doing reads in quorum. We go to the node currently responsible for the key on the consistent hashing ring and get the data. Since we had writes in quorum so, as long as, one of the two nodes is available, we will get the right data. If both go down, we are in trouble. We should probably increase replication factor to 3 for stronger fault tolerance and better consistency.

3. Lets say if we synchronize data in background, what might be the worst case latency?
[You] It depends. But it should be few seconds since we have a small node cluster.

4. What is the process for synchronizing a node with other parts of the system, once it comes online (new or failed one)?
[You] As of now, we remap the keys from the predecessor node but we should do this more intelligently. My friend told me that they use Merkle tree to accomplish this. We can do the same but it will require time.

5. If we follow an asynchronous replication mechanism, how would we handle data conflicts?
[You] This is a good question. My friend's company uses Vector clock but VCs are complicated and grow forever. Since we just have one actor in our use case, VC might be overkill. We can either use LWW i. e. Last Write Wins or keep both conflicting versions. LWW is simple to implement and might be OK for our use case. If LWW is not acceptable, we can keep both conflicting versions and let application handle it. Application can in turn show both versions to customer who can subsequently choose the best one possible. This might be a little inconvenient but would always uphold data consistency. Also the chance of happening this is very small.

You manager is attentively listening to your answers. He appreciates your honest feedback. He think for a while and tells you to go with background replication. He is ready to tradeoff small inconsistency window for high availability. He looks fine with your read methodology as of now & agrees that you should increase replication factor to 3. Also since replication will now be done in background so it will not be a bottleneck. Since you have a strict deadline to resolve customer issue, he wants to look at the new proposed mechanism to sync data nodes using "Merkle Trees" at later time. You will stick with your existing approach for now. He is not sure on how to handle conflicts. You both debate about both approaches. Finally, you both agree that since it will be bad to loose user data, you will keep both versions and let application handle the conflict. You hope that this should be rare.

asynchronously replicate data in background [We will talk more about this in later posts]. Once you implement it, the replication is now handled in background with a latency of few seconds. The node specific LRU cache is refreshed after data is synchronized on a particular node. Your replication factor is now 3 which means that the data on each node increases roughly from 1.1 million to 1.65 million records. This doesn't look that bad. You now have an option to store multiple versions of a records against a key, in case you get conflicting data set. This can be resolved by application. Finally, you are all set now.

You test your system with various use cases & are proud of your accomplishment. This is just awesome! You show the system to you manager and appreciate his contribution. You both feel on top of the world. The new changes are pushed to production & everything works perfectly. The speed is even better for updates now since the node doesn't have to wait for predecessor to complete the write. This looks like the best day of your life! Congrats!

Note: *Databases like The Apache Cassandra Project and Riak – Basho Technologies support both quorum and asynchronous replication mechanism. Apache Cassendra offers something called as "tunable consistency" which can be tuned to get weak to strong consistency. I encourage you to look at these databases for getting more details. All Things Distributed is a link to great article by "Werner Vogets" (Amazon CTO) about consistency in distributed systems. ArchitectureOverview - Cassandra Wiki is the architecture overview of The Apache Cassandra Project.*