LOVE FOR PROGRAMMING

Distributed Systems Part-1: A peek into consistent hashing!

Pawan Bhadauria

When we talk about scale, we talk about distributed systems which scale horizontally. If you want to find where all you can be proved wrong, dive into distributed systems. Anyways, the first thing we would learn about distributed systems is to distribute data efficiently across machines or efficiently balance load across servers. I will talk about the former in this post. As usual, we will start with a real world example tackling problems as they come. So lets get started.

Lets say, your boss comes to you and shows you a file which contains a list of IP address and time-stamp in each line. The IP address represents the client address and time-stamp represents the time at which client accessed your site. The file contains 100K such combinations. The timestamps are in increasing order which means the second last record always has a time-stamp less than the last one. Here is a sample format:

111.111.111.111 2014:02:01:12.0400
222.222.222.222 2014:02:01:13.0410
333.333.333.333. 2014:02:01:14.0500
111.111.111.111 2014:02:01:15.0200

He wants you to write an algorithm which, given a client IP address, should return the recent time stamp when the client accessed your site. You can relate to this key-value dataset. Since given a key, you need to do repeated look-ups, this fits pretty well with hash-map which provides O(1) look-up time. While your boss is looking at you, you do some back of the envelop calculations, think about hash-map and finally tell him that this should be pretty simple to do. Here is a rough outline of the algorithm:

You will do a linear scan through the file putting the key-value pair in a map in O(n) time. If you encounter a duplicate key, you will replace it, since time-stamp are in ascending order. Once the map is loaded, the algorithm would be able to give the most recent visited time for a client in O(1) time. You burst with pride.

Your boss asks you a very pertinent question, "Are you keeping the data in memory?". You reply in affirmation, telling him that data is not too much & it would give almost instant look-ups. He frowns and tells you that "**memory is currently not an issue but don't assume it will never be an issue**". You decode his statement and after a while, get it. He is referring to company's fast growing site which can produce quite a lot of data moving forward. If you keep all client IPs in memory, it will blow the server. With awareness of scalability creeping in, you agree with him. You ask for some time to think about this. He nods and vanishes away. Peace!!

Now you start thinking about the problem. You think that, you gave the best solution which works in O(1) time, based on your data structure knowledge, but unfortunately there are obvious issues with it. You visualize a 1 billion row file & your head starts spinning. You think about LRU caches but reject it since a miss would be deadly (remember 1 billion records on single machine) and we want extremely fast look-ups. But you are a problem solver. You see that the solution to keep data in a hash-map is nice but memory is an issue. Why not have the data distributed on multiple machines? Each machine holds a limited set of data which can be accessed faster. Good strategy and looks feasible.

But the first problem is to find the basis on which you will distribute data on different machines. You have 10 machines at your disposal. You again visualize 1 billion records file & see that 10 machines can each have 100 million records. Whats the best strategy to distribute data? You also think that the data related to an IP address, should always go to same machine as and when it arrives. You remember hashing, your best friend. The IP address string can be hashed to a number which can subsequently be mapped to an associated machine. Since there are only 10 machines, so to avoid overflow, you need to mod the key with machine count. You get your hashing algorithm:

IP address => hash number => hash number % machine count => machine id on which key should go.

For example, 111.111.111.111 => 994968954=> 994968954%10 => 4. Thus key 111.111.111.111 goes to 4th machine.

Cool!! You go full speed implementating it. Using sockets, you write a small service (daemon) which runs on each machine. You also write a light service facade which takes a key, applies the above algorithm on the associated key to find machine on which data should go & then establishes communication channel with the machine to extract data. For writing a new row, process is same where in application invokes service facade with the key (IP address in row) and some payload (time-stamp in this case) which is subsequently appended to the file on target machine. Since the data foot print on each machine is considerably low, you use LRU cache [fixed size] to cache the data before returning it. Even if there is a miss, it will not be as deadly. Fair enough!! Everything is set and you show the solution to your boss.

He is excited and asks you couple of questions about machine failures. Since you have not thought about it so he tells you to focus on scalability and efficiency & asks you not to worry about fail-over and data replication, at least not for now. You reluctantly agree!

You build a solution and deploy it in production on 10 machines. The data comes and resides on the assigned machine based on key and everything works. Since data set per machine is 10 times small, speed is stunning.

One day your boss tells you that data on each machine is growing rapidly & you probably need to add more machines otherwise each machine will start crawling with its huge data set. You agree. He suggests that you add 10 more machines. You look at you algorithm and think that you will do modulo with 20 instead of 10 and data will start flowing to 20 machines now. Your boss skeptically asks a question "What about existing data set?".

You see a problem. The keys will now map to a different number and reads will be navigated to a different machines where the data for that IP doesn't reside. You look concerned. The best solution is to redistribute data again from 10 to 20 machines. You boss asks, how much time will this redistribution take? You tell him, "less than an hour **but site will not be available for that time duration**". He frowns.

You are stuck in a weird situation. You ask him for some time to think about this. He nods and vanishes away. Peace!!

You think that if the site continues to grow at the same pace, this situation might happen every month. Also we have not even thought about fail-overs yet. You consult your friend who works at a fast growing consumer company. He bluntly says that "your hash function sucks and you should look at **'consistent hashing'** if you want to horizontally scale your system". You ask him, "What is consistent hashing?". He starts explaining:

"Consistent hashing is a special kind of hashing such that when a hash table is re-sized and consistent hashing is used, only k/n keys need to be remapped on average

traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped." [Wikipedia]

You have already seen the last part of above statement in action.

To understand better, here is some more explanation. Lets say you had 15 numbers (keys), [1,8,3,7,5,6,4,2,11,15,12,13,14] & 3 machines. You know your key range is 1-15 for this example. You allocate fixed ids to your machine, 0, 5 and 10. *Now your algorithm, based on a key, puts the data on the machine with id less than or equal to your key* [In real world, range would be really large but here, just for example, we are assuming it is fixed as 1-15]. So:

machine 0 -> 1, 2, 3, 4
machine 5 -> 5, 6, 7, 8
machine 10 -> 11, 12, 13, 14, 15

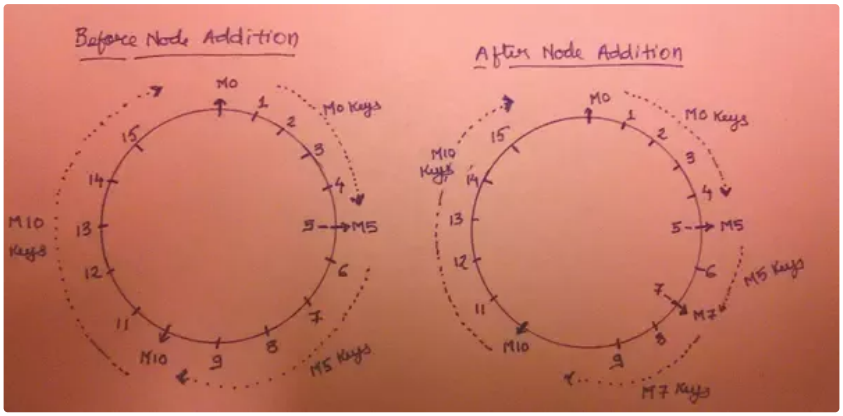
If new key, 9 comes, it goes to machine 5 which has id less then 9.

machine 5 -> 5, 6, 7, 8, 9

Now consider this. You got one more machine and want that to be added to cluster. You allocate any number between 1-15 as id to that machine. Lets say 7. Now you have 4 machines in cluster. The data needs to be redistributed. But guess what, the data on machine 0 and 10 is not impacted. Only data on machine 5 is impacted. The keys on machine 5 which are greater than or equal to 7 only needs to be redistributed. So here you are:

machine 0 -> 1, 2, 3, 4
machine 5 -> 5, 6
machine 7 -> 7, 8, 9
machine 10 -> 11, 12, 13, 14, 15

This can be best visualized on a consistent hashing ring. Here is the sample diagram for above example.

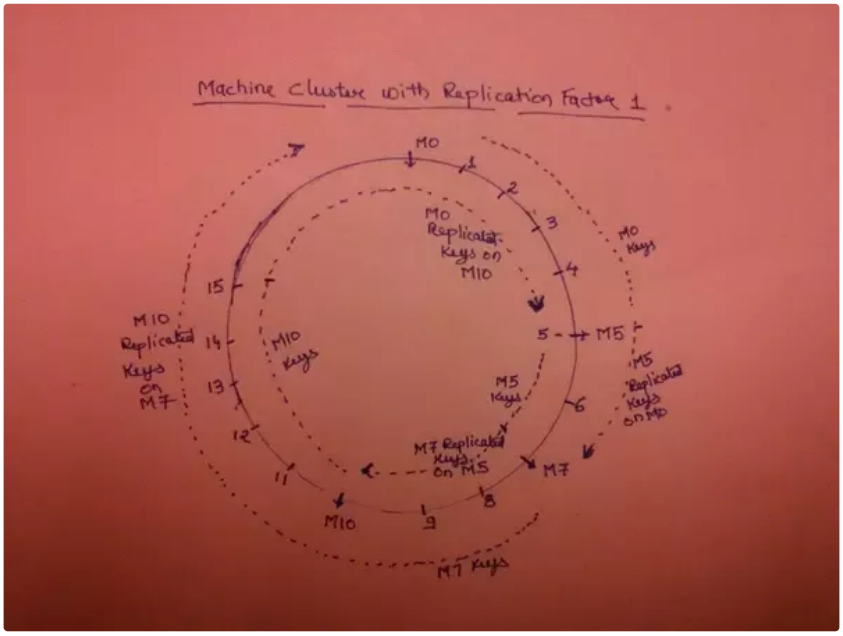


Congrats!! you have just implemented consistent hashing. So now, in case you want to add a new machine, you only need to redistribute key from a single machine which is predecessor to this machine on the ring. We have come a long way.

You show the new algorithm to your boss & he is impressed. But as usual, he asks you another question, "Adding node is fine, what happens if a node containing a set of key goes down?". You look at your consistent hashing ring & get an idea. But as usual, you ask him for some time to think about this. He nods and vanishes away. Peace!!

When ever a machine goes down, the machine occurring before that on consistent hashing ring will be responsible for crashed machine's keys as well, right? For ex, if M7 goes down, which machine will the service call for key 8? M5, isn't it? Because M5 occurs before key 8 on circle as M7 is no more available. So you think, why not keep a

use a replication factor of 1 which means that at least 1 node will have replicated data in case failures happen. This is really simple. So here is the new diagram (looks a little messed up but bear with that). As you can see, now M0 holds its keys along with replicated keys of M5. So every time a key comes to M5, it is replicated on M0 as well. Same happens for other machines as well.



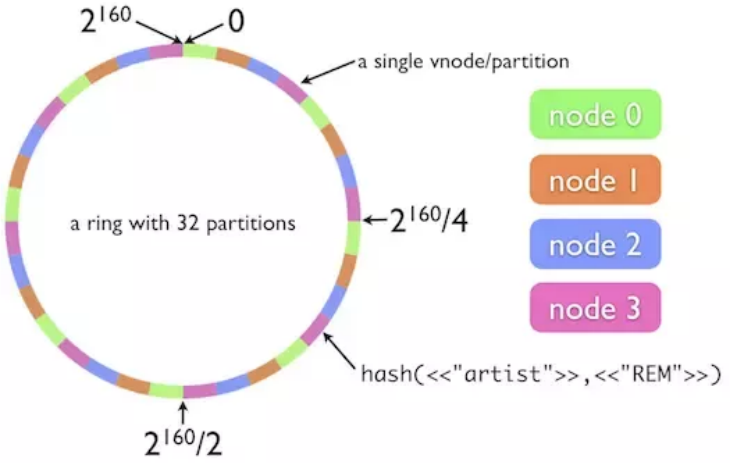
Now you have cluster of machines which work in tandem. The keys are distributed across machines and even if one goes down, the predecessor takes the responsibility. So your cluster is not only load balanced but is also fault tolerant. This is Awesome!!

With this amazing load balanced & fault tolerant architecture, you go to your boss with confidence & pride. He passes a smile and says "Great Job".

You were literally saved by consistent hashing. You not only appreciate the simplicity with which this can be implemented but also the enormous power it brings to handle scalability and failures.

The little algorithm you implemented above is in fact used in many no sql databases. There are many things which I have excluded here for obvious reasons but the fundamental thought is same. This is the first step in learning how distributed systems work & you have crossed it. Great!

Note: Many times, the fixed machine id which we have used above, is itself calculated based on hash function which is used for getting hash keys. This technique sometime results in load imbalance if machine are further apart from each other and one gets substantial load then other. This can actually happen, even with statically allocated machines ids on the ring. This is often solved with concept of virtual node, where in we put many virtual nodes on the ring which in reality point to same machine. So technically, a single machine can be responsible for many partitions in the circle. This technique is used in The Apache Cassandra Project and Riak – Basho Technologies. For visualization, here is a borrowed diagram. This diagram might look a bit complex but it isn't. It just says that, there are four machines but are responsible for 32 partitions on the ring. Each green partition is handled by node 0, orange by node 1 and so on...



Happy hashing... consistent hashing!!