3 – Object and Key-Value Storage

Context:

We are in a scenario in which data does not fit in a single machine. This means that in order to scale-up ACID guarantees will no longer hold, and instead we will have CAP (Atomic Consistency, Availability, partition tolerance and eventual consistency) guarantees.

The file content is stored in blocks to only pay the latency cost once for every block as opposed to once for every bit.

Some important numbers:

- 1-14 TB local storage / server
- 6 GB-4 TB RAM / server
- 1 - 10 Gb/s network bandwidth/server (notice the small b)

Object storage (Amazon S3)

The S3 model is a logical model that can be used to store very large objects. There are buckets, and objects can be placed in those buckets. A bucket is identified by a bucket ID, which can be used to retrieve the bucket and its objects. Each object can occupy at most 5 TB and is identified by an object id.

Notice the simplicity of the model, which is one of the reasons scaling with this model is easy.

The API offers access via a driver (e.g. JDBC), SOAP and REST. Recall that one of the main advantages of REST is that almost all languages can connect using REST since only an HTTP library is required. The resource to be accessed is identified by an URI.

In general, a URI identifying an object looks like this:

http://bucket.s3-amazonaws.com/object-name

and can be accessed using standard HTTP requests (namely GET, PUT, DELETE, POST).

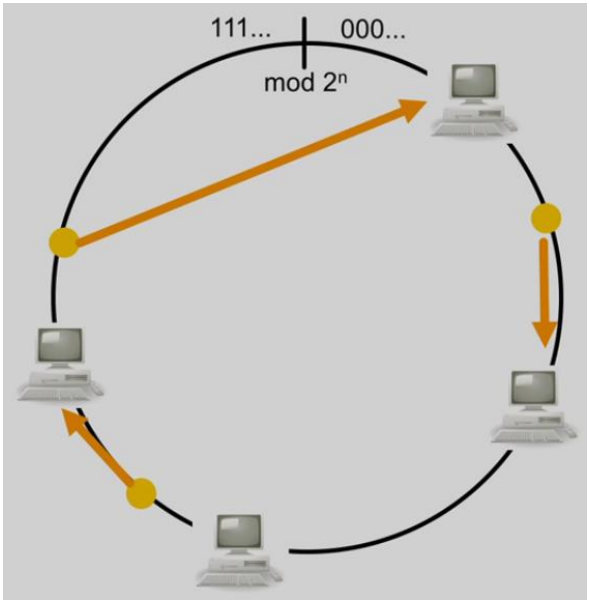
Since the only thing needed to identify an object is the bucket id and the object id, S3 is a key-value model, where the key is the bucket id and the object id. But is S3 a file system? On a logical level yes, but on a physical level no since the structure we see is only virtual (emulated on top of the key-value model).

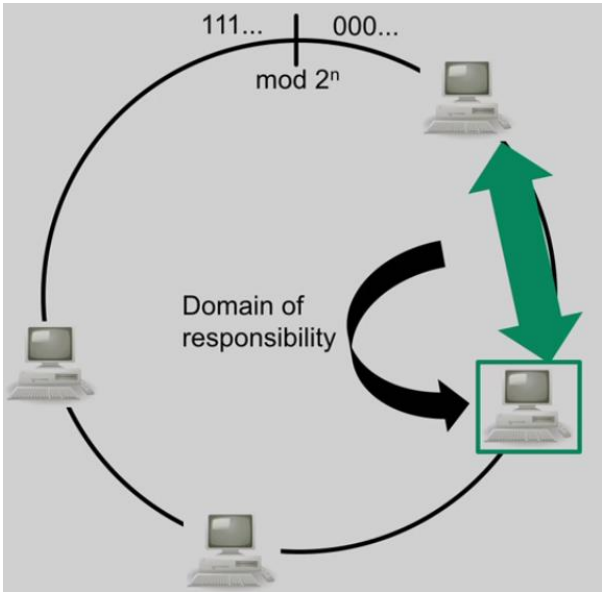
As we want our system to be fault-tolerant. There are two kinds of faults, **local faults** (node failure) and regional faults (such as natural catastrophes). These issues are both addressed by replication, on a local level but also on a global level across the globe. Replicating across the globe also means that customers will, on average, be closer to the data, so they will experience lower latency.

DynamoDB: Key-value storage

Object storage cannot be considered as a database due to its latency being much higher than that of a database (300ms vs 5ms). DynamoDB also associates keys to objects (values) but in a much faster way. Therefore, the model is also a key-value model, but in this case the objects will be much smaller than with S3 (around 400Kb as opposed to 5 TB). The API is in fact very similar; the only difference is that a "**context**" appears now on HTTP requests. A context will be returned after a GET, and a context must be supplied in a PUT.

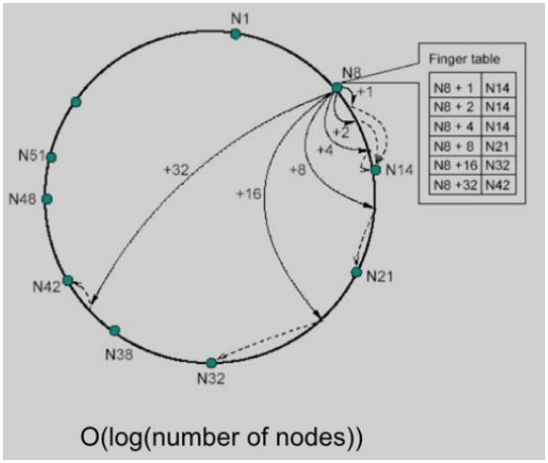
Looking at the implementation, this is implemented as a P2P **distributed hash table** (DHT). The key is hashed and the last 128 bits (modulo) are considered. Each node randomly picks one of those values in the 128 bit space. The node is then responsible for the objects whose value is bigger than the value of the previous node and smaller than the value of the node (the values "between the two nodes"). The values for which a node is responsible are called the **domain of responsibility**.





When a node is added/deleted, keys are transferred between nodes so that this structure is preserved (i.e. when a node is deleted keys are transferred to the next node). However, it could also happen that a node crashes and no time is left to transfer the keys (which would be lost). In order to avoid this, we replicate. A node will store its domain of responsibility but also the one of the previous node. That way, every key is stored twice by different nodes.

Each node has a **finger table** that allows it to know in which node a given key is stored. Since it is not feasible to store the location of each and every key, only some logarithmically spaced locations are stored to use as a reference. Then, binary search is done to find the exact location.

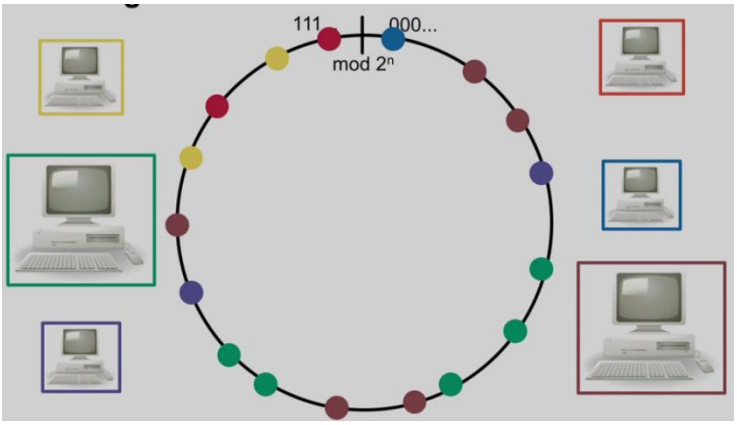


Some of the main pros of DHTs are: being highly scalable, robust against failure and self organizing.

Some of the main cons are: It is a lookup, not a search (we have to know which key we are looking for since we cannot search for objects). Data integrity and security are also concerning.

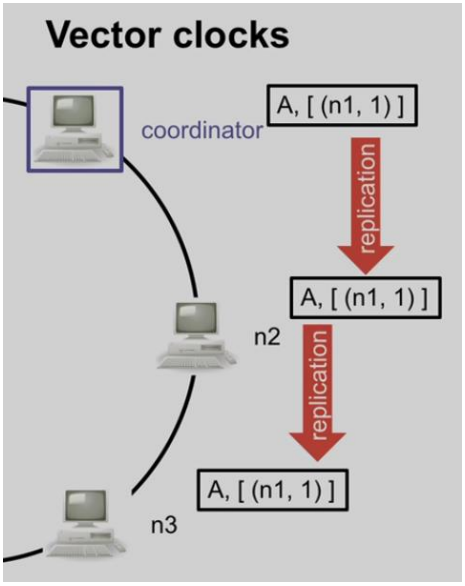
But… what happens if we are very unlucky in the random assignment of values to nodes? This is, what happens if some node ends up being responsible for a much larger proportion of values than another one? Also, what happens if there are performance disparities between nodes?

To solve these two problems we introduce the idea of **tokens**. Tokens will represent virtual nodes, and a physical machine will be responsible for multiple tokens. Tokens are randomly assigned to nodes. The more powerful a machine is, the more tokens it will be assigned.

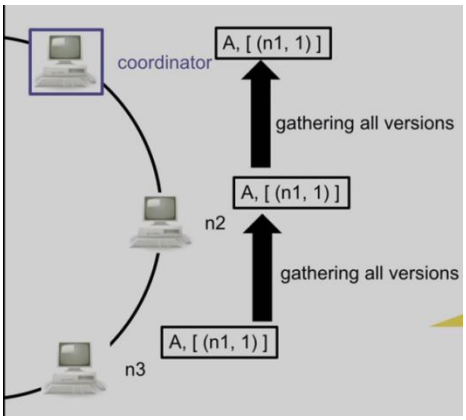


Another important issue is how to keep track of versions. Recall that an object is stored in multiple nodes for the system to be fault-tolerant. It might happen then, that two nodes write the object being unaware of the other node's write. To track versions across different nodes, we will use **vector clocks**. Vector clocks will define a partial order. When the ordering between two elements cannot be determined, a conflict is taking place and resolution is needed.

When a write (PUT) takes place, it goes to a load balancer which assigns it to a random node. The node contacts the coordinator responsible for the object which then performs the write and propagates it to the other nodes that are also responsible for this object.



Each node then stores the written value (A) and also a context that sais which node last wrote to it and how many times.



When a read happens, versions from all nodes are compared and if they all agree (no conflicts) the value is returned together with the context. The returned context must be later passed as a parameter when extra writes are desired.

The generalization to more nodes/objects is as follows: for each object, each node keeps a vector of n positions (where n is the number of nodes). When a write of that object happens at a node, the node increases his position in its own vector clock and propagates it.

E.g. If the vector clock of object A at node 0 is (0, 0, 0) and a write happens at A, it will become (1, 0, 0) and it will be propagated.

When another node receives the object with the associated vector clock it has to update his own vector clock. For this, it will compare his own vector clock with the received one and, at each position, keep the maximum of the two.

E.g. If the vector clock of node 1 is (0, 2, 0) and it receives the vector clock (1, 0, 0), his resulting vector clock will be (1, 2, 0).

Now, we can compare versions of the same object at two different nodes by comparing their vector clocks. Given two vector clocks v1, and v2, we say v1 < v2 if for all positions *i* v1[i] <= v2[i].