Patrick Holland
CSC 462 - Summer 2019
Distributed Systems
Term Project

Video Demonstration & Git Repo

The first video I made showed the entire system being setup, but it was over 12 minutes long so I attempted to make a shorter video that just focused on demonstrating the system in action. This second video still came out to be over 5 minutes which is quite a bit longer than the target of 2 minutes, but nonetheless, here they are:

https://youtu.be/5uRvufWvgdY
https://youtu.be/fIO0qVApS60

https://github.com/holland11/boat_datasatore_csc462_project/

Introduction

The idea for this project comes from a real world problem. A naval architecture firm wants to setup a database to contain the information for each part that a manufacturer offers. This way, when they need to select a new part, they can query the database for every part that fits their needs and then select the one they want to use.

The two main challenges with this type of system are storage capacity and read latency. Storage capacity is super important because this database could end up containing millions of entries so you want the system to be able to scale up easily whenever its storage gets close to being fully utilized. Read latency is important because the clients won't want to have to wait for a long time whenever they submit a query. Write latency isn't much of a concern, because it's not that important for clients to be notified anytime a new part is added to the database.

A distributed system can help tackle both of these challenges. Storage capacity in a distributed system is more or less infinite. Anytime the current storage is getting close to full, you can spin up a few more AWS instances and add their disk space to the system. Read latency can also be improved by making this a distributed system. Imagine having 1,000,000 entries on a node that the database has to search through for any given query. Now imagine having those same 1,000,000 rows split among 10 different nodes. Each node now only has to search through 100,000 rows. Also, if the rows are split based on a common query criteria, it becomes likely that any given query will only have to be run on one of the nodes. This means the other nodes have less load so the system can handle more request at any given time.

The Database

When choosing a database, the first question you need to answer is relation or non-relational. A potential oversimplification is that relational is better for structured data, while non-relational is better for horizontal scaling. The data in the context of the boat datastore is quite structured, but our focus is on distributing the system so we chose MongoDB which is a non-relational DBMS.

Another group is working alongside us and they are implementing their database in MySQL which is a relational DBMS. We will be running tests in an effort to compare the two, but it should be noted that due to the configuration complexity, the results of those comparisons shouldn't be considered very reliable when debating which system is better.

These tests could help decide which configuration settings to use when comparing results across different configs for the same DBMS. To perform this type of tuning, you'd want to have a really solid understanding of how the data is going to be used so you can setup your tests accordingly.

The Schema

Table 1: The schemas used in our database. Each item is given a unique _id field when created.

Collection	**Boat**	**Part**	**BoatPart**
Schema	{ Name: String }	{ Heading: String, Spec_Heading: String, Features: [String], Model: String, Hyperlink: String, Source: String, Weight: Float,, Material_And_Color: String, Size: String }	{ BoatID: String, PartID: String, Quantity: Number }

We were provided with sample data in CSV format. The CSV contained around 4000 rows; 2522 of which were for parts. Using this data, we came up with these schemas. They are far from complete, but they cover the basics of each collection and the core relationships between them are captured. Sitting down with some actual users to get a better idea of how the data will be used and what information is needed or beneficial would allow you to fine tune the schemas quite easily and any changes wouldn't likely have any impact on the actual system.

3D Models

One of the most important columns in the data is the part's 3D model. 3D modelling is a key process in the design and engineering of these boats so providing users with easy (and quick) access to the 3D models should be a top priority. Giving manufacturer's an easy way to upload their 3D models is also something to be aware of. Amazon S3 buckets is one of the main services to consider for this task.

S3 buckets allow files to be stored *on the cloud* at which point they provide you with a link to access that file. That link can then be saved in the part's database row. There are other services and tools that can be used to solve this problem and you could even manually manage the storing of the 3D models on your own AWS instances, but S3 buckets are simple to use and can fulfill the necessary purpose.

Mongo Cluster

One of the main reasons we chose MongoDB was that it has a very well defined and documented cluster mode which is specifically built for distributing the database across multiple hosts. The cluster is made up of 3 components: routers, config servers, and shards. The routers act as an interface between the client and the shards. The config servers store metadata and the cluster's configuration settings. The shards store a subset, or *chunk*, of the data.

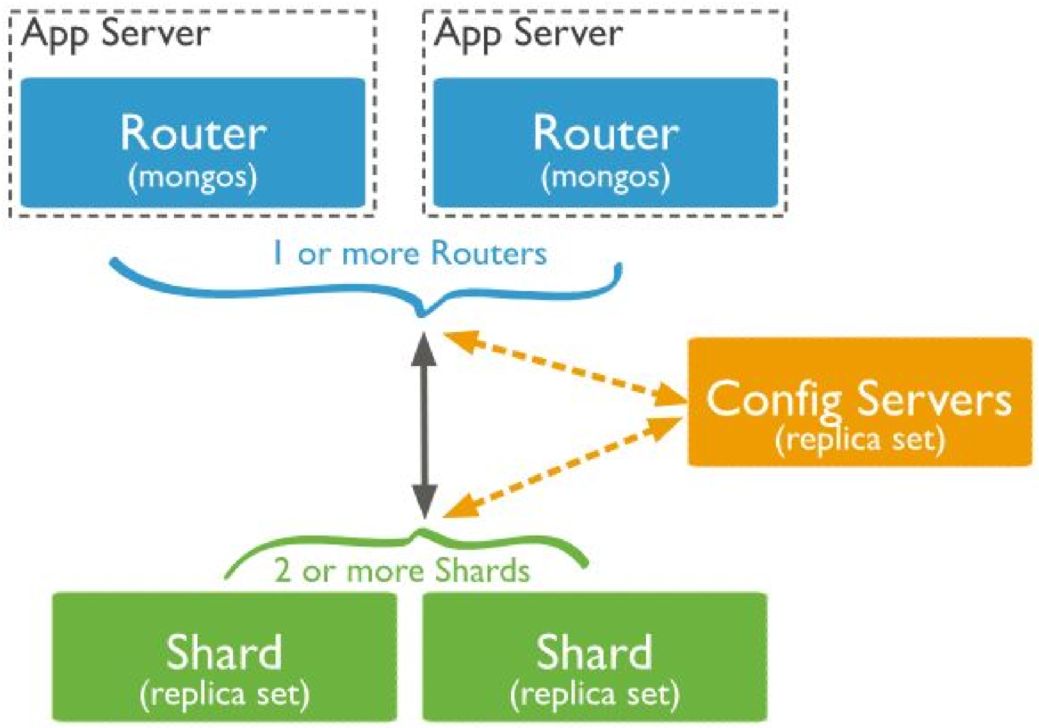

Figure 1: MongoDB cluster diagram taken from the MongoDB documentation.

In older versions of MongoDB, it’s possible to deploy a cluster with only one service for each of the components. However, this wouldn’t provide the availability or fault tolerance that is of paramount importance within a distributed system. It is recommended that each shard is made up of a *3 replica set*. These replica sets use a *leader-follower* majority protocol so under working circumstances, each replica set will have 1 primary node and the rest of the nodes will be secondaries or *arbiters*.

The config servers are part of a *3 replica set* as well and it is recommended to replicate the router as well which can provide better availability, better fault tolerance, and improved latency through the use of load balancing.

One of the most important decisions when designing the cluster is selecting a sharding key for each collection. The sharding key must be one of the columns within that collection and it is used to decide which shard any given row should be stored on. If a good sharding key is chosen, the majority of queries will be able to target only 1 of the shards. This allows the other shards to save their processing power for queries specific to them. We made the assumption that queries would generally be for a specific type of part and therefore the results of the query would all have the same *Heading* value. While working with users to fine tune the schemas, potential sharding keys should be given a lot of thought.

Arbiters

Arbiters were briefly mentioned in the above section, but they can play an important role and should be considered when designing the system. An arbiter is a member of a replica set that takes part in leader elections, but does not replicate the data and never becomes leader. The logic behind its usefulness is that you want 3 nodes per replica set so that if one node goes down, you still have enough for a majority vote. Having one of those nodes be a vote-only node allows you to save roughly 33% of your storage space by not replicating the data a 3rd time while still providing the replica set with enough members to maintain read and write availability when one node goes down.

Web App

To interface with the databases, our group and the MySQL group worked together on a web application. This web app was built on NodeJS with React for the frontend and Express for the backend. The frontend provides basic read and write functionality and the backend provides an API capable of routing read and write requests to the proper DBMS. The API can be accessed with simple POST requests rather than going through the frontend and this is what we used for our stress testing.

Containers and Docker Swarm

Containers provide a consistent and portable environment for the services to exist within. They are an incredibly important concept when working with distributed systems and Docker is the most popular tool that provides containerization. Docker provides a mode called *Swarm* which has a little extra complexity to setting up, but provides some major benefits if implemented correctly.

The main benefits that *Swarm* provides are a virtual network mesh for communicating between services on different hosts, load balancing of those services across all available hosts, and simple deployment of services across the entire swarm.

Initializing a swarm provides a command that can be used on other hosts to have them join the swarm. Deploying services from one of the swarm manager nodes will cause those services to be spread out among all of the swarm hosts. This is great because it means you don't have to manually deploy the services on each host. It also means that when a service goes down, the swarm can easily and quickly detect that and spawn another one on an available host. One issue that comes up from this is that the services can be placed on any host so there's no way to know what its IP address would be.

The services need to be able to communicate with each other and they can't do that with IP addresses due to them being very dynamic. This is where the swarm's mesh network comes in handy. The swarm maintains a virtual network with DNS capabilities so whenever serviceA wants to communicate with serviceB, it can use the hostname serviceB and the swarm's network will route the message to the proper IP address. This also means that if you want any services exposed to the public, you can assign a specific port within the swarm to that service and now any message to that port received by **any** of the swarm nodes will be rerouted to that service. For example, the web app could be exposing port 80 and its service is currently hosted on 192.168.0.5. If a message gets sent to 192.168.0.2:80 and that IP address belongs to a swarm node, the message will get rerouted by the swarm and the web app service will receive it.

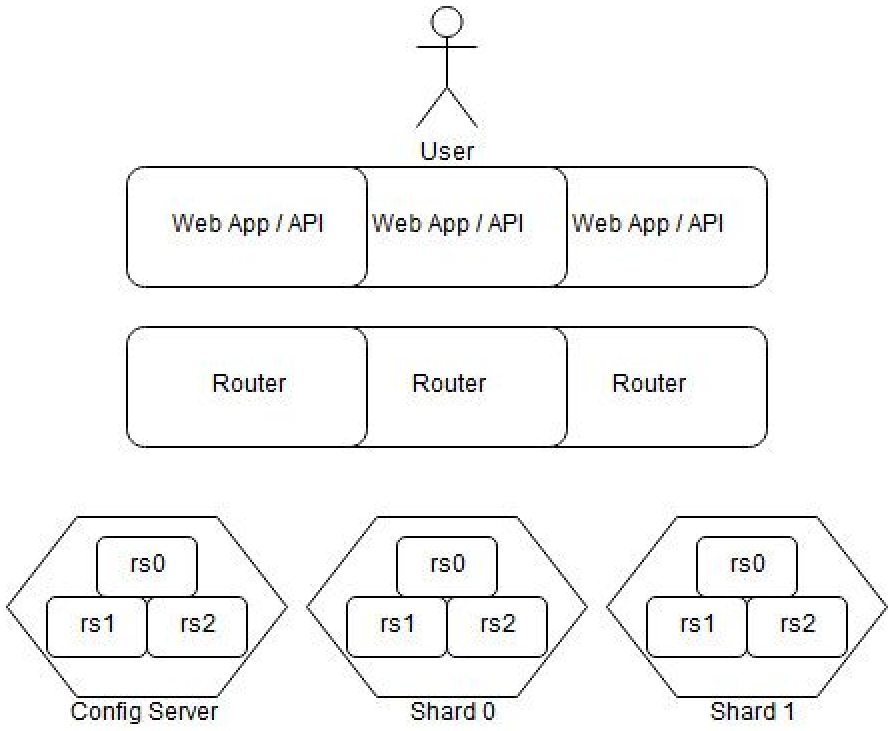

Figure 2: Diagram showing the containers / services present in our MongoDB cluster with 2 shards. Each rectangle is a service. Web App and Router services are replicated 3 times.

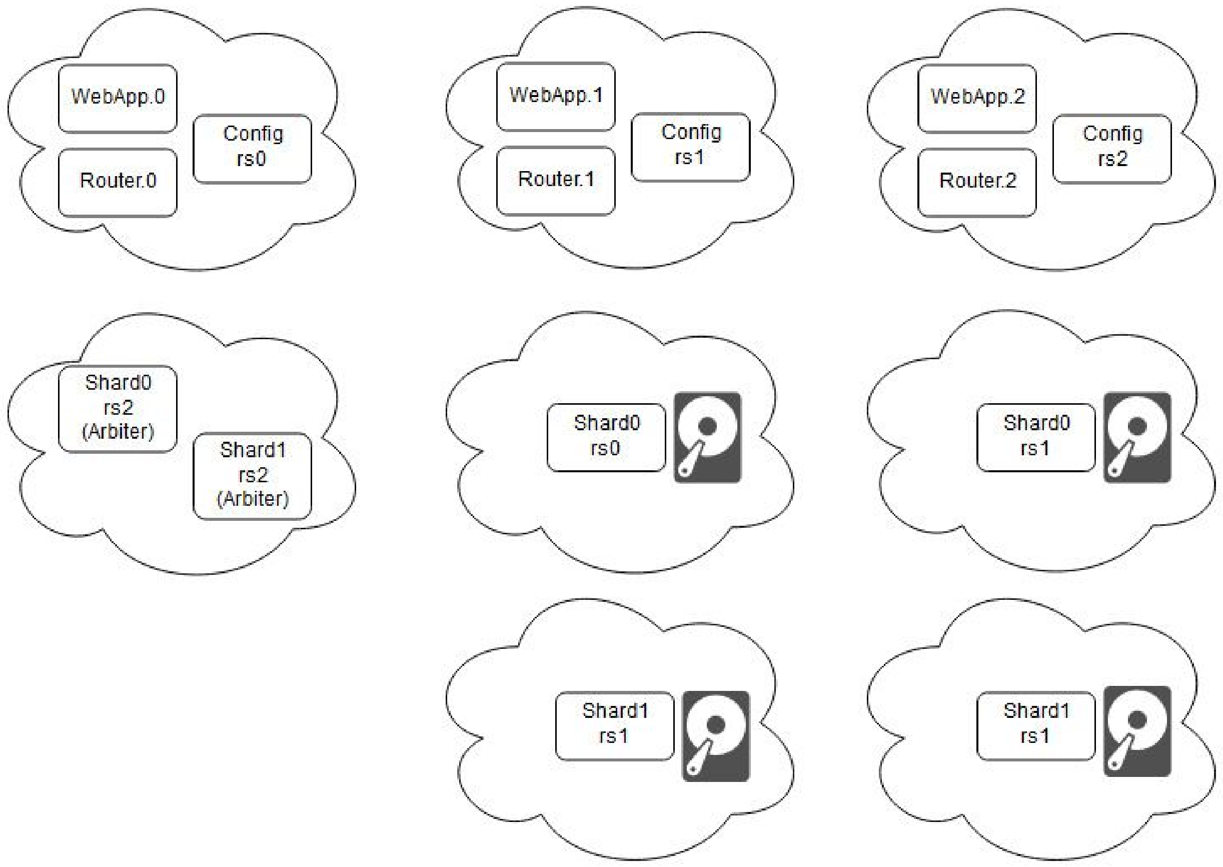

Figure 3: Diagram showing an example placement configuration for all of the services across AWS instances. Each cloud represents one AWS instance. Each rectangle represents a service.

Figure 3 above shows how the services might be spread out across multiple AWS instances. This example features 2 shards where each shard is made up of 2 storage nodes and 1 arbiter. It also has both the router and web app services replicated 3 times. Because the web app, router, and config instances do not require much storage or processing power, they can exist on the same host without much issue. Because the arbiters do not have data replicated to them, they do not require their own storage space so each arbiter can exist on the same host. The shard nodes that actually store data should not share any of their resources with other containers so they should be given their own unique host.

This isn't the only proper service placement so changes can be made or if the swarm manager doesn't place the services in this exact way, it's not necessarily a big deal. However, you want to make sure that no 2 storage services exist on the same host and you want to make sure that 1 host going down won't cause the system to become interrupted. For example, you wouldn't want Router.0 and Router.1 to be on the same host because that would defeat the purpose of replicating the service. You also wouldn't want Shard0 rs0 and Shard0 Arbiter to exist on the same host because if that host went down, Shard0 wouldn't have enough nodes left to elect a new leader.

Docker Swarm has two configuration attributes that can be used to specify how you want services placed across hosts. Placement constraints are used to say "This service **must** be placed on a host that meets this criteria". If no host exists that meets the criteria, the service will remain pending until one is found. Placement preferences have the same usage, but they are more lenient so the services will always get placed somewhere, but it might not be where you want them to be.

The System

Replication & Consistency

The replication implemented within MongoDB is *asynchronous*. This means that when a client issues a write, they receive confirmation as soon as the write is successful on the primary node of the shard. That write will be replicated eventually to any secondary nodes associated with that shard so this is an *eventually consistent* system.

This results in clients receiving faster acknowledgements from the database, however you should be aware of the tradeoff. Imagine the client issues a write, the primary node executes the write and returns back to the client. The client now assumes that their data is stored, but the primary node crashes before replicating that write to the secondary. Now the secondary gets elected leader and the original client issues a read query to see their data. Their data doesn't exist despite them receiving confirmation that it was written successfully.

A decision would need to be made regarding how acceptable this hypothetical scenario is. One possible solution would be to maintain a log of all writes and then have the server double check those writes after an arbitrary amount of time. If the written data is not found when the logging server checks for it, it can re-issue that write. The person designing and implementing this solution would need to be aware of many different possible scenarios such as writing data then deleting it immediately after so it's not a trivial problem to solve.

Fault Tolerance

This system is designed with fault tolerance at its core. Every service is being replicated so that one service / container going down won't interrupt anything in the system. As long as the services are placed correctly onto valid hosts, such as in figure 3, one host going down won't interrupt any of the services. If this system were to be implemented in a production environment, you'd probably want to have hosts in many different datacenters so that you don't introduce one datacenter as a single point of failure.

Missed

There were many different aspects that we did not have the time or expertise to dive into, but that would be important, if not necessary, for a production environment.

Security

Security is an incredibly important issue to consider when storing data. Whether the data itself is private and you don't want unauthorized users accessing it, or you just want to make sure that normal users don't have the ability to run certain commands such as deleting an entire collection of data. This is not something that we implemented or spent time on because it is not directly relevant to the *distributed system* side of the project.

Backups

Even though the data is being replicated so that one node going down won't erase all of the data, it's still incredibly important to save backups of the data regularly. Especially for this scenario where manufacturers will be inputting **a lot** of their data into the system and they would not be happy to hear that they need to do it again because their data wasn't backed up.

Logging Server

A logging server to keep track of all user requests and their results would probably be a worthwhile investment for this type of system. A lot of useful analysis could be done on the data contained in the logs. For example, it could be a good representation of what the most common queries are so that you can better optimize the database to speed up those requests. It could also show you which requests are fast vs which requests are slow which could help shape future decisions about the database or the overall system. Another use for a logging server was mentioned in the *Replication & Consistency* section (ensuring the asynchronous replication actually gets replicated successfully).