An Erlang Distributed Database: How to Better Understand The CAP Theorem

Damian Mirizzi, UG; Graham Matthews, PhD

Abstract—Description of Area of study:
When making large ubiquitous systems such as Google calendar or email you come across the issue of storing massive amounts of data. It is inefficient to store all that data on one machine as well as unsafe because if that one system was compromised then all the data can be corrupted. Companies such as Apache or MongoDB have developed open source databases that can scale across multiple environments, while giving end users a simple API to develop with. I will be studying distributed systems and algorithms to develop my own interpretation of a distributed database.
The Problem Being Addressed:
There is a theorem in distributed systems called the CAP Theorem it is described the balance of these three attributes: Consistency, every read would get you the most recent write. Availability, every node (if not failed) always executes queries. Partition-tolerance, even if the connections between nodes are down, the other two (A & C) promises, are kept.I would like to produce a database that balances the attributes well for large distributed systems with smaller nodes such as IOT devices.

Index Terms—Computer Science, Distributed Systems, CAP.

1 Introduction

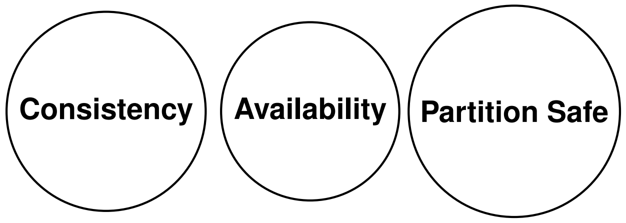
As we humans head towards a digitized future, where we are not the owners of of our data, rather we consume shared storage systems at some cost. Most of our stored information gets shoved into some database to be quarried at some later date. These databases have a number of limitations depending on the the structure they are designed in, One of their largest limitations are the hardware they are built upon. Data centers have multiple dependent systems such as hard drives and the systems that monitor those drives. Computers can only monitor a specific amount of hard drives before it becomes cumbersome and the computational costs outweigh the strengths of the machine where the hard drives reside. One resolution is to increase the speed of the hardware, but this I often expensive. One more common resolution is to distribute databases and computation over multiple different nodes. This allows for slower less expensive machines to monitor their memory better, working along side multiple other systems to complete larger more intensive tasks. My research is analyzing the system requirements required to run these distributed databases, how to make the systems more reliable, and how to speed them up when reliability is not particularly needed.

There are a number of tools available in the market that provide distributed database functionality, such as Mongodb or SQL, although both tools have very different constructions, they operate using similar commands. The commands typically base off of selections, stores, updates, and deletions. In single machined databases those commands are pretty simple; iterations over column names but when propagated over a network. Returned queries sent over the network can be massive, just the map reductions can be extremely time consuming. There are a number of technologies that try address this; by using caches rather than redoing expensive queries or by saving commonly used quarries. The problems with caching arise when caches are get hit much more than the databases and, when keeping the caches up to date. I think it is possible to create a distributed database that requires minimal caching while retaining maximum availability, consistency, and partition tolerance.

- *Graham Matthews is with the Department of Computer Computer Science, California Lutheran University, CA, 91360. E-mail: gmatthew@callutheran.edu*
- *D. Mirizzi is with California Lutheran University. E-mail: dmirizzi@callutheran.edu*

Manuscript received April 1, 2018; revised April 29, 2018.

2 The CAP Theorem

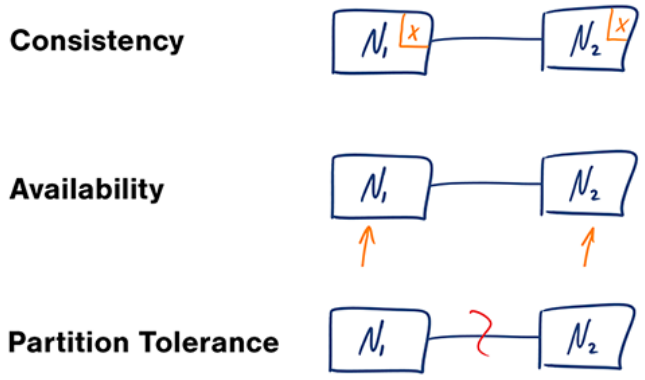


The CAP theorem was originally proposed by a Berkley professor E. Brewer, the theorem was proven ten years later by members working in MIT. The Brewer (CAP) Theorem as it is most commonly called effected databases in the 80s but the problem was not quantified and solved for thirty years. A brilliant corollary that still befuddles database designers today.

The CAP Theorem states that, in a distributed system(a collection of interconnected nodes that share data), you can only have two out of the following three guarantees across a write/read pair: Consistency, Availability, and Partition Tolerance - one of them must be sacrificed.

Consistency says that you must be able to retrieve the most recent write to a given database client. Availability states a functioning database will always return a reasonable response, not a timeout or error. The most important, partition tolerance states a system must continue working if a network error occurs and divides the database subnet.

It is important because networks are not reliable. Networks and parts of networks go down frequently and unexpectedly. Network failures happen to your system and you dont get to choose when they occur.



Given that networks are not completely reliable, you must tolerate partitions in a distributed system, period. Fortunately, though, you get to choose what to do when a partition does occur. According to the CAP theorem, this means we are left with two options: Consistency and Availability.

This leads to two common database types AP and CP systems. AP systems return the most recent version of the data you have, which could be stale. This system state will also accept writes that can be processed later when the partition is resolved. Choose Availability over Consistency when your business requirements allow for some flexibility around when the data in the system synchronizes. Availability is also a compelling option when the system needs to continue to function in spite of external error. CP systems wait for a response from the partitioned node which could result in a timeout error. The system can also choose to return an error, depending on the scenario you desire. This system typically needs to be connected to all nodes before it can be quarried. Choose Consistency over Availability when your business requirements dictate atomic reads and writes. For my implementation I will be designing my database to switch between an AP and CP system depending on network and request factors.

3 Structure

A large part of my project was how I decided to design and implement my database. There are a bunch of great programming languages and frameworks that I could have used to make my database. Primarily I picked Golang a systems language known for its concurrency, however I learned that such a system would have been cumbersome to use. I would have had to make a distributed framework to exactly fit my needs because the options available today did not meet my standards. This is how I constructed my project, trial and error, tried new and old ideas to best structure a system that I envisioned.

3.1 Key Value Store

A key-value database, or key-value store, is a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a dictionary or hash. Dictionaries contain a collection of objects, or records, which in turn have many different fields within them, each containing data. These records are stored and retrieved using a key that uniquely identifies the record, and is used to quickly find the data within the database.

Key-value databases work in a very different fashion from the better known relational databases (RDB). RDB's pre-define the data structure in the database as a series of tables containing fields with well defined data types. Exposing the data types to the database program allows it to apply a number of optimizations. In contrast, key-value systems treat the data as a single opaque collection, which may have different fields for every record. This offers considerable flexibility and more closely follows modern concepts like object-oriented programming. Because optional values are not represented by placeholders as in most RDBs, key-value databases often use far less memory to store the same database, which can lead to large performance gains in certain workloads.

This key value structure is perfect for large distributed systems because when multiple nodes return queries it is very easy to do a map reduce on the multiple key value maps. These Reductions can also be easily parallelized and is why the new big data DB structures tend to use key value stores.

The one issue I came across when structuring the data was generating unique keys for inputed data, I could have implemented a hashing function(this does not insure incorrect overwrites) but rather I chose to allow the database user to input his own unique key to give developers ways to be more space and speed efficient. This is more efficient in the example of setting the key to like a Social security number that we know is unique rather than to store it somewhere in the value set tied to some meaningless hashed key value.

3.2 Erlang OTP

The language I chose was Erlang, an old language that derives from ProLog, an even older language. It had a hard learning curve because it was not C-esqe, it is completely functional, does not have a concept of loops, uses pattern matching, and all methods return tuples with a reassuring {okay, (then some set of return data that typically takes a form of a map or list)}. But there was one massive upside, OTP!

OTP stands for Open Telecom Platform, although it's not that much about telecom anymore (it's more about software that has the property of telecom applications, but yeah.) If half of Erlang's greatness comes from its concurrency and distribution and the other half comes from its error handling capabilities, then the OTP framework is the third half of it(This is my capstone paper and I can use a halve based counting system if I want).

All behaviors in Erlang's OTP are predefined, servers manage state, supervisors manage servers, applications start and stop the Database as a whole. Intern that is how I structured my database using server modules and superiors. You can see in Figure 1 how I structured my node, all 4 of these services can run on the same physical system but only accounts for one actual database node. For my implementation I will need multiple computer systems communicating Via the internal API service to other DB nodes on the erlang network. The Paxos node will have the job of managing the nodes on the network. In Figure 2 You can see how the API on one system can control the whole network of database nodes, this is why I specified that only some nodes need to have an API manager and Paxos operation, I did this to allow for a light weight deployment on less powerful systems. We will go into greater detail about behaviors later in this paper.

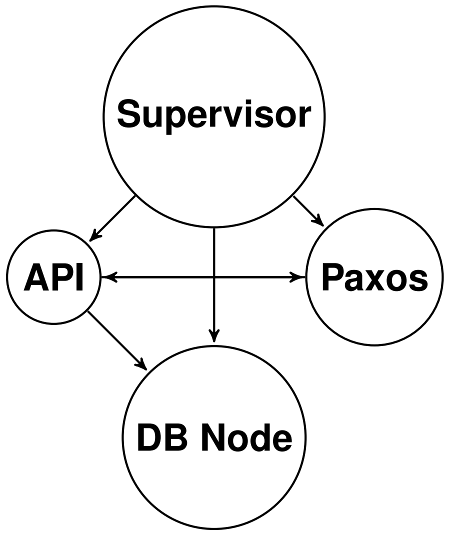


Fig. 1. Erlang Supervisor Structure

3.3 Redundancy vs Distribution

One of the large issues you come across when storing data to a distributed data base is deciding how often and how many copies of the data you should make. If every node of the database are exact copies of each other that would not be very space efficient; however it will have great partition tolerance because all nodes should be basically self reliant. This defeats the purpose of distributing your database, nodes will all have to store massive amounts of data it will not take the compute network to its advantage. So we must decide to only replicate information on some nodes. How many nodes? What nodes? Are questions you should be asking at this point.

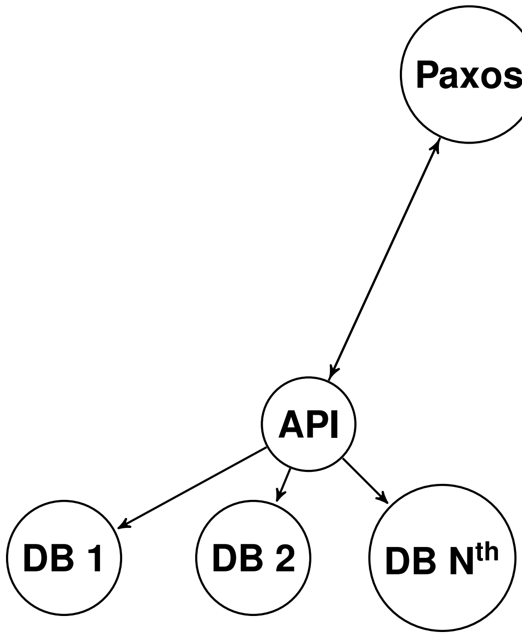


Fig. 2. The Paxos Hierarchy

For my implementation I assumed that all nodes are just as theoretically stable as the others(This statement should be made because hardware and network failures are very unpredictable). With that statement I can determine that there is not a best node to store information on, so stores can be completely random. The bigger question is how many copies should be made. For my implementation I chose a recursive counter that will for every node N in the network, when N is above two, add one to the number of copies every time N Is a prime number. So for example when N is three the number of copies should be two. This is a very conservative method because the count prime numbers to infinity tends to be linear. I chose this to be difficult and it could have easily been replaced with some linear equation with a slope between one and five. A more space efficient and lower bounds to maintaining reliability is an logarithmic curve in Figure 3. I imagine there is an upper limit to when copying information to other nodes for redundancy does not make since, ultimately the difference between one thousand copies and two is small because both would require massive concurrent network and hardware failures to cripple the system. The upper limit can probably be defined, however it is best to leave hard limits to the people deploying the database to protect their precious data unless they choose otherwise.

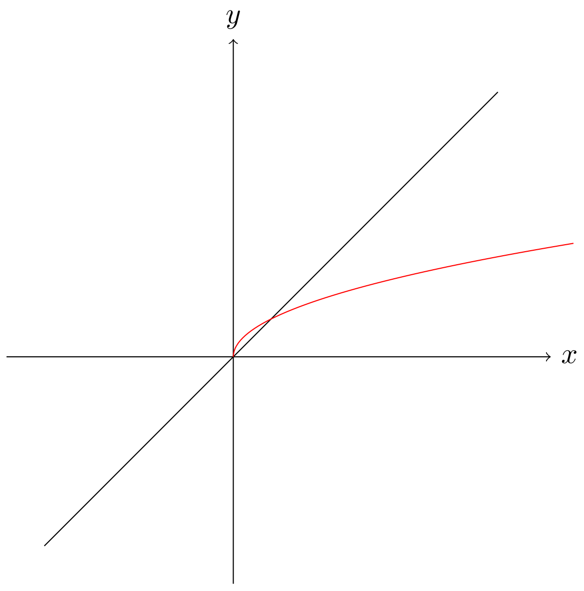


Fig. 3. Primes to Infinity(black) vs Log for Efficiency(red)

3.4 Paxos

Paxos is a mechanism for achieving consensus on a single value over unreliable communication channels. The algorithm defines a peer-to-peer consensus protocol that is based on simple majority rule and is capable of ensuring that one and only one resulting value can be achieved. No peers suggestions are any more or less valid than any others and all peers are allowed to make them. Unlike some other consensus protocols, there is no concept of dedicated leaders in Paxos. Any peer may make a suggestion and lead the effort in achieving resolution but other peers are free to do the same and may even override the efforts of their neighbors. Eventually, a majority of peers will agree upon a suggestion and the value associated with that suggestion will become the final solution.
The reason I use Paxos is to manage the nodes on my network. The API needs to know what nodes are on the network at all times and the Paxos algorithm can set a nodes as a leader, or choose to vote a node off entirely if it thinks the node is faulty, or not up to par. This algorithm is use strengthen distributed database without using machine learning, also it is a real time test so it does not use predictive models which tend to be incorrect on random real time systems like databases.

3.5 API

For the API I used Erlang message passing, so it does not meet the standards of REST, RPC, or SOAP. The reason I did this is to allow the final developer to implement their own client facing API. There are a lot of free frameworks such as Cowboy to help develop HTTPS style APIs in Erlang, but I did not want to develop a default one because it would have been time consuming given there is a fully functioning Erlang API that does not need custom documentation because the Erlang compiler can develop docs. It will also allow for developers to develop an RPC system for internal calls or a RESTful system for external calls.

3.6 Supervisors

One of the cool behaviors in Erlang is the supervisor. The supervisors are responsible for starting, stopping, and monitoring its child processes. The basic idea of a supervisor is that it must keep its child processes alive by restarting them when necessary. The children of a supervisor are defined as a list of child specifications. When the supervisor is started, the child processes are started from left to right according to this list. When the supervisor terminates, it first terminates its child processes in reversed start order, from right to left.
Supervisors have a bunch of built in methods for restarting downed children, and is one of their powers. In my implementation a downed child will just be restarted in the same place the child node died protecting the state. This structure is very important in a database node because they technically function as concurrent processes to API calls and they should not be effected if an error occurs in the case of a malformed query. There are other methods where the supervisor will restart all other nodes in the network at the same time, or kill nodes that are before it in some predefined processing queue.

3.7 Distributed Frameworks

When deciding what language to use I picked Erlang for its great distributed framework OTP, before that I was looking at goCircuit, it is a distributed framework that is based off Hoares CSP. The issue I had with Hoares CSP in the case of this project is it required all messages and channels be to some degree synchronous(Hoare also states it possible to make asynchronous system act as if it was synchronous by requiring acknowledgment to all messages, but I feel that is still too constricting). Although I understand TCP is important to get past most internal routers in business settings. I would have been fine with a UDP asynchronous framework. I fell somewhere in between, Erlangs OTP uses TCP, but does not require acknowledgment and expensive to continue on with processes. Rendezvous tend to be extremely expensive, golang and Hoares CSP tend to be full of them, I am interested how they sped up such systems(aside: Rendezvous is one of those English words that we should just change, I am sorry but it just looks way too much like an adjective and we are not French).

3.7.1 Asynchronous vs Synchronous

For those that don't know what synchronous and asynchronous messages are:
Synchronous messaging involves a client that waits for the server to respond to a message. Messages are able to flow in both directions, to and from. Essentially it means that synchronous messaging is a two way communication. i.e. Sender sends a message to receiver and receiver receives this message and gives reply to the sender. Sender will not send another message until it gets a reply from receiver.
Asynchronous messaging involves a client that does not wait for a message from the server. An event is used to trigger a message from a server. So even if the client is down, the messaging will complete successfully. Asynchronous Messaging means that, it is a one way communication and the flow of communication is one way only.
The only real difference between the two message calling protocols is context state. In a Synchronous system you can just go back to exactly what you were doing, while Asynchronous system you would typically manage each message as its own fictional call independent of context.

4 Results

I learned a lot while doing this capstone project, I learned that there is a lot of work that goes into distributed databases, and that the CAP Theorem is not going anywhere. The CAP Theorem is really just the universal laws of physics applied to distributed computing, there is no form quantum tunneling that allows for data on one machine to be automatically modified when a bit shift occurs on another.
I was able to get a working database that could be used as a simple in memory database or cache store, and then made it possible to distribute it to multiple systems. The project may not be completely deployable in an Enterprise situation because of some short comings in the API and its build tools (given I am not an expert of building and

deploying Erlang applications, especially because Erlang seems to take a different deployment work flow from a lot of modern systems). I also produced this awesome paper with a bunch of pretty pictures and so on...

5 Conclusion

As we humans head towards a digitized future, where we are not the owners of of our data, rather we consume shared storage systems at some cost. Shared storage like my Erlang Database, one that was made over a semester of college with the explicit purpose of passing my final computer science class. You can go grab Hadoop, MongoDb, or some other off the shelf database that has large developer groups and hundreds of paid developers. It is your choice in designing the best system possible. I would probably pick one of those other databases over mine. That being said the construction of this database really helped my critical thinking skills and helped me as a coder and computer scientist develop some awesome code and read thought provoking papers regarding distributed programs and concurrency as a whole.

Acknowledgments

The authors would like to thank all of the people that made this possible. Most important the families that supported us through our academic endeavors.

References

[1] Brewer, E. Towards Robust Distributed System. Symposium on Principles of Distributed Computing (PODC), 2000.

[2] Brewer, E. CAP twelve years later: How the ”rules” have changed. Computer, vol. 45, no. 2 , pp. 23-29, 2012.

[3] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. ”Communication efficient leader election and consensus with limited link synchrony”. In The Proceedings of the International Symposium on Principles of Distributed Computing (PODC), pages 328-337, 2004.

[4] Hoare, Tony. ”Communicating sequential processes”. Communications of the ACM vol 21. 666-677, 1978.

Damian Mirizzi Computer Science Major at California Lutheran University. As a Swenson Research Fellow Damian produced a distributed robot to study computer vision and spacial awareness. As his final Capstone project he wanted to produce a distributed database because he wanted to challenge himself. Damian is currently working at XYPRO Corp where he is a Software Developer in the information security field.

Graham Matthews B.Sc. (Computer Science) with First Class Honors, University of Auckland, Auckland, New Zealand.
M.Sc. (Computer Science) with Distinction, University of Auckland, Auckland, New Zealand.
Ph.D. (Mathematics), University of Georgia at Athens, Athens, USA.