Jepsen methods usage for ACID compliance in Hyperscale Cloud Frameworks

Mark Jervelund
Mark@jervelund.com

Institute Of Mathematics and Computer Science,

SDU

September 15, 2021

Contents

Abstract

Databases allow modern society to store, manage and distribute data at a previously unprecedentedly scale. having data is a normal practice for most businesses, but it is often done monotonically on a single node, where modern needs require much high performance, storage, latency and availability then current systems allow. Distributing a data store in a manner that scales performance, storage, availability and desire properties is no easy feat. In this Thesis i present methods of verifying these properties and investigate how different frameworks solve this issue, or often workaround the issue in a way that aligns with the requirements of the systems.

Within the framework of the thesis i present methods of verifying the ACID properties, How these properties were solved in the past, How it can and is solved today, what trade offs some systems have made, and a analysis of modern database uses cases and whether they require the ACID properties to solve issues.

First, I Present ACID & Base, their constraints, what faults occur in database systems, how their manifests themselves, whats the underlying causes often are, and what limits the ACID properties imposes on a given system. Here Base will be introduced to explain what relaxations are introduced to a system where gaining the desire results for certain use cases.

Then, I Present Jepsen, a CLojure framework[**jepsonio**] developed by K. Kingsbury that allow for testing of distributed systems, This is solved via a direct serialization graph (DSG) and queries to the target system that are carefully chosen that allows for traceability, and recoverability.

Then, I present Azure Service Fabric(SF), SF is a distributed container orchestration system made my Microsoft, that allows for hosting of services, or apps. It includes a few different built in functions, but the primary interest here lie within the reliable containers aspect of SF, and what claims Microsoft makes concerning the behaviour of this data.

Then, I will present what modern database systems promise, what properties of ACID they follow, which they relax, and which they disregard, and well as what they gain, and which tradeoffs they suffer, Here a main focus will be on Service Fabric.

Then I will present the attempt to implement and execute a Jepsen test against service Fabrics reliable containers, and compare these results to the claims Microsoft has made.

Finally, I will present how the ACID properties compare to modern uses cases of databases, do we need to follow them strictly or can simply disregard them in some uses cases? What are the exceptions, and what is there to gain ?

Chapter 1

Introduction

Highly Distributed systems are becoming more and more common as the world grows evermore interconnected, faster paced, increasing data driven with higher expectations of services and product. where the end user expects reliability, availability combined with low- latency and cost, How do we guarantee this is a distributed system that spans the planet. Is it even possible to get both, and is it even needed in most use cases?

Most people are familiar with how YouTube, Facebook, Google and Reddit behave. What quirks they sometimes present, Most people remember the YouTube video counter that got stuck around 300, and didn't update for a while, The reason it got stuck was induced, however the number it got stuck on was not, The system was designed to stop at 301, but often went higher. This is an example of the system not being entirely consistent, most nodes where in the correct state of stopping counting when the number got larger then 300, however as some nodes around the globe weren't in an consistent state these still counted the views onto the displayed counter when they were supposed to have been sent to a different table that verified the views before considering them legitimate. This number is still not consistent around the world however for human use cases the issue of a comment or view being a few seconds or a minute delayed isn't something we notice or care about. a lot of user facing services aren't required to be ACID compliant, however there are use cases where this can have an impact is when dealing with data that is produced and consumed in real time, if this is finance, automated or autonomous systems, or other areas where having consistent, atomic and isolated transactions are of the utmost importance. these cases are where it doesn't matter what server we are querying the information from it has to be the most recent sample, and if we at any point are able to query stale data we end up being at the risk of either making an illegal transaction or an autonomous system making a wrong decision, This can a trading system fetching stale data and making the wrong transaction, a bank withdrawal that might be overdrawing the users account. or autonomous vehicle that might making decisions with stale data that can lead to fatal accidents.

How do we test and verify that the systems that handle tasks that require compliance with ACID also do so. To understand this we first need to understand the basis for and expected behavior of ACID and the different levels of each of the properties of ACID.

Chapter 2

Database transaction models

Database and data store models can be categories into two main groups, ACID and BASE. Consistent or available. The underlying reason for this limitation is either latency between nodes in a system, scale-ability both in terms of storage or Capacity in terms of queries to the system. Both models of designing the system has it's trade-offs which will be presented later in the paper.

ACID

The ACID Model of handling database transactions is considered monolithic by some, however they still serve a vital a critical function for handling critical systems that require atomicity, consistency across the entire data-set, reliability, and durability.

The ACID acronym is defined as follows in the DBMS book[**DBMSbook**].

- "A" stands for "atomicity," the all-or-nothing execution of transactions.
- "C," stands for "consistency." That is, all databases have consistency constraints, or expectations about relationships among data elements (e.g., account balances may not be negative after a transaction finishes). Transactions are expected to preserve the consistency of the database.
- "I" stands for "isolation," the fact that each transaction must appear to be executed as if no other transaction is executing at the same time.
- "D" stands for "durability," the condition that the effect on the database of a transaction must never be lost, once the transaction has completed.

ACID therefor offers strong consistency with rigorous handling of transaction isolation that prevent inaccurate data. This allows for designing of a system, where we can prevent operations on stale data, data loss, or "illegal" transactions. These faults will be presented further down in the paper.

BASE

The BASE model allows designing of a system where we value availability, throughput and scale-ability. This can cause issues with stale data, dirty reads, overwriting data, and other undesired behavior. This can benefit some applications where overwriting old data isn't an issue or where the newest version of data might not be required as long as it'll come eventually. The use for these databases are hugely beneficial for social media, logging, and other hyperscale system where consistency isn't required.

The Base Acronym was defined by Eric Brewer[**brewer2000towards**] and is defined as follows.

- Basically Available – Rather than enforcing immediate consistency, BASE-modelled NoSQL databases will ensure availability of data by spreading and replicating it across the nodes of the database cluster.

- Soft State – Due to the lack of immediate consistency, data values may change over time. The BASE model breaks off with the concept of a database which enforces its own consistency, delegating that responsibility to developers.
- Eventually Consistent – The fact that BASE does not enforce immediate consistency does not mean that it never achieves it. However, until it does, data reads are still possible (even though they might not reflect the reality).

The BASE model of databases often have a weak consistency where stale data is considered "OK" so to say, while offering a best effort approach with approximate answers. but gains availability, performance, and it can be considered a relaxed way of handing the database side of things where the system database system is simpler and easier to modify the schema..

CAP theorem

The CAP Theorem was defined by Eric Brewer[**CAP**] where it states that it is impossible for a distributed database system to provide Consistency, Availability, and Partition Tolerance in a single system, and that only two of these guarantees can be met.

It should be noted that the definitions of the terms differ from the definitions in ACID. They are all important when it comes to a distributed systems and their behaviors. firstly Consistency in ACID is define as constraints on the data. By the CAP consistency concept, ACID would follow Sequential consistency as defined by Lamport[**lamport1993how**]: "the program behaves as if the memory accesses of all processes were interleaved and then executed sequentially." while the consistency model in CAP is defined as Atomic Consistency (also called linearizability) is sequential plus a real-time constraint: "Unlike sequential consistency, linearizability implicitly assumes the notion of an observable global time across all processes. Operations are modeled by an interval which consists of the period of time between the invocation and response for the operation and each operation is assumed to take effect instantaneously at some point within this interval." [**CSL-TR-95-685**] CAP states we can only

have 2 of the 3 properties in any given data-share system. The three different options will be explained below.

- Consistency and Partition tolerance(CP)
  A system that deliveries Consistency and Partition tolerance but the trade off here is Availability, If a partition occurs in the system, the non-consistent nodes would have to be shut down or made unavailable to deliver consistent data. CP would cover majority protocols, and most distributed databases. an Example of a CP database would be MongoDB and Service Fabrics Reliable collections, These work by having partitions that contain a master and a set of replicates. The replicates simple follow the masters transaction log and apply it to their own data set. If the Primary becomes unavailable the Replicate with the most recent transaction log simply comes the new Master, doing this switch the partition becomes unavailable while the replicates catches up to the new master. this causes the network to remain consistent but limits availability.
- Availability and Partition tolerance (AP)
  If our system foregoes Consistency, and deliverers availability and partition tolerance, In the case of a partition between nodes we keep serving from all nodes but in this case we might serve stale data and occurrence might also occur where the same row contains different values due to multiple different write operations on the different nodes. An example of a AP database would be Cassandra, where the CP have a master/replicate architecture, This would cover DNS, Caching systems and databases such as Cassandra. Cassandra uses a masterless architecture, It does mean that there is multiple points of failure rather then a single one. It is able to be available and partition tolerant but consistency isn't guaranteed as nodes are always available and in case of partitioning the nodes will diverge when partitions and will then heal once they are connected back to the network.
- Availability and Consistency (CA)
  Database delivers consistency and availability but doesn't allow for partitions of the network or nodes. this results in a single node, or single cluster system as any distribution of the system introduces network instability and latency that would break the system. In this case it would cover

single node/site databases and cluster databases as well as file systems as we in practice wouldn't have a distributed system that doesn't allow for partitioning as it would be unusable. But an example of this would be a single node Database. PostgresSQL could be an example here, however, PostgresSQL dues support replication but then it becomes a CP database, which some asterisks[**aphyrpostgres**] as it doesn't quite behave as expected in that case either.

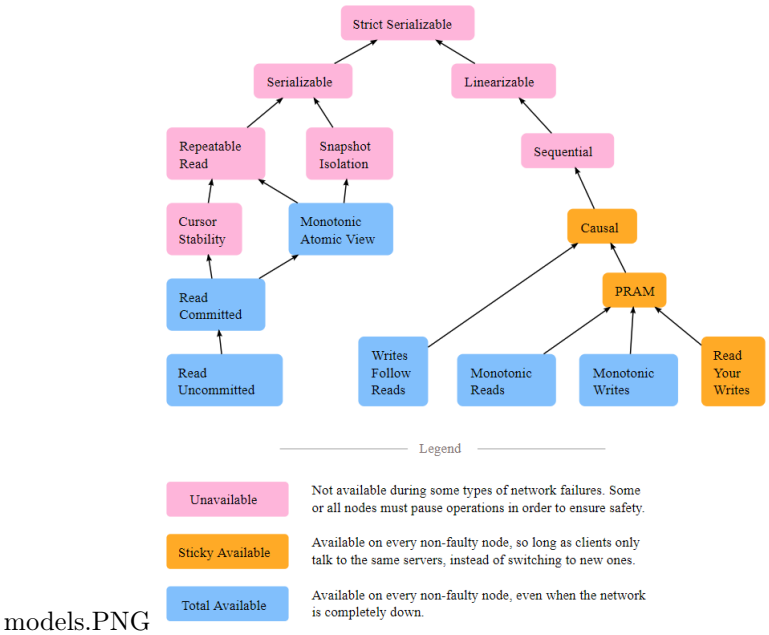
Consistency models

For explaining the the concepts within consistency i will use the a paper by Ballis et al on "Highly available transactions"[**10.14778/2732232.2732237**] that displays a good model for presenting the different consistency models and their relation to other consistency models.

Strict Serializability

For a system to be Strict Serializatbility, it is required that the entire system operationally appear to occur in the order, with regards to both the order and the real time of the operations.

Formally Strict Serializatbility is define as as Serviceable system that is compatible with a Time dependent order. "A history is serializable if it is equivalent to one in which transactions appear to execute sequentially, i.e., without interleaving. A (partial) precedence order can be defined on non-overlapping pairs of transactions in the obvious way. A history is strictly serializable if the transactions' order in the sequential history is compatible with their precedence order."[**10.1145/78969.78972**]



models.PNG

Figure 2.1: Picture from jepsen.io/consistency

Here it should be clarified that the Obvious way that that if we have transaction A and B, That A proceeds B if A Completes Prior to Transaction B Begins or in other worlds a Serialzable system with the time constraint from Linearizability.
TODO Make Diagram

Serializable

Serializability is a relaxation of Strict Serializability that foregoes that real time constraint, it defines systems where transactions occur in some total order, It is formally defined in the ANSI SQL 1999 spec as follows. "The execution of concurrent SQL-transactions at isolation level SERIALIZABLE is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins."[**ansisql1999**]

The above implies that Serializability of the transactions does not only apply to the objects in use but the system as a whole, this means that in case of a network partition some or all notes will be stuck until the network is healed. Futher as no real time constraint, we can observe stale reads, This occurs when Process A completes a Write, Then process B begin a read, the read is not guaranteed to observe the Write from process A,
TODO Make Diagram

Repeatable Read

Repeatable Read closely resembles a Serializable system, however in RR,
TODO Make Diagram

Cursor Stability

TODO Make Diagram

Read committed

TODO Make Diagram

Read Uncommitted

TODO Make Diagram

Snapshot isolation

TODO Make Diagram

Monotonic atomic view

TODO Make Diagram

Linearizable

TODO Make Diagram

Sequential

TODO Make Diagram

Casual

TODO Make Diagram

Writes Follows Reads

TODO Make Diagram

Monotonic Reads

TODO Make Diagram

Monotonic Writes

TODO Make Diagram

Read your writes

TODO Make Diagram
Two of the consistency models were already mentioned in the previous section namely Atomic Consistency and Sequential consistency, There exist 2 more within common use, these two are Causal+ Consistency and Eventual Consistency. They differ in way the consistency is handled by all end up in the same state eventually.

Faults, manifestation, and causes

First the 3 types of behavior is explained as these are used to explain the 4 levels of isolation. the are categorized as the 3 below.

- dirty read,
- Non-repeatable read and

- Phantom read

From this knowledge a table can be made that gives us an overview of the isolation levels and what behaviour and the trade off they have. Isolation level Read phenomena Dirty read Non-repeatable read Phantom read read uncommitted yes yes yes read committed no yes yes repeatable read no no y serializable no no no Maybe also include something about snapshot isolation
'Dirty read' is when a S1 can read data that S2 has written but not yet committed, It is considered dirty as S2 can rollback the transaction where S1 read data that must be considered non existent. Make example diagram
The second case of 'Non-repeatable read' is when S1 reads data that is changed by S2 and committed. so if S1 read the some data again they will have changed. this results in two equal select statements returning different results. Make example diagram
Phantom read The Third and last type is phantom read which is a special case of non-repeatable read that occurs when S1 reads data where a where condition is used to specify what data we want. After this initial read, a second session S2 inserts data that meets S1's where condition and commits the data. When S1 issues a select statement with the same where condition, it finds new records. It is called phantom read because the new records seem to be of phantom origin. Make example diagram

dirty read

Non-repeatable read and

Phantom read

Long Fork

Split Brain

The Network is split into 2 or more parts where they each believe different truths,

Transient Anomalies

Process 1 writes A, then process 1 tried to read A but a doesn't exist in the node, and later process a is able to read A

Chapter 3

Database Consistency testing Possily rename to Jepsen

Introduction

Modern Database systems, and the trade offs they make.

Atomicity, and Isolation as we often write the data to a arbitrary node, that accepts after the data has been distributed outside of it's rack, availability zone or region. and where the newest timestamp then overrules all other written data without care of data on other nodes which can breaks Atomicity, and Isolation, as a check and set(CAS), or update statement might uses stale data and due to it's newer timestamp any data written in the meantime gets discarded. Solutions here might require waiting for full distribution of any data prior to the query executing, but this would drastically slow down the entire system. This does however not factor in issues related to system faults """

Historically

Historically this was done with using a manually defined hand coded set of patterns to check how a system behaves wrt to the isolation levels above, like if some proven invariants hold, or if an anomaly is present in parallel is present by inserting record x and y in two separate transactions and and in two more transactions checking if we can observe x but not y, or y and not x, as this could show how the systems handles long forks and snapshot isolation, and more importantly if it supports it at all.
these checking are generally quite efficient and run in polynomial time, but they only check for a certain patterns and therefor don't give us the larger picture of where issues are present, and only cover a fairly limited set of configurations and isolation levels, and are defined on a system to system basis and no interchangeability is supporter.
This means that this has been a giant scope project where case by case tests are predefined and where we only test for certain scopes, which more importantly mean that tests have been done but the test coverage haven't been perfect and a lot of anomalies are never detected.
Need to add some more to this section, add examples are past test, diagrams, papers etc

Jepsen

" Jepsen is an effort to improve the safety of distributed databases, queues, consensus systems (...) exploring particular systems' failure modes. In each analysis we explore whether the system lives up to its documentation's claims. "[**jepsonio**]

Going from the above quite Jepsen is a way that allows us to check whether a given database system lives up to the requirements and premises that are given, and in a world where distributed, hyper scale or even planetary scale systems are becoming the norm the way data storage is done is required to behave in an expected manner, the reason that expected manner is used rather than correct manner is

that we may allow certain faults to occur and eventual consistency to allow for higher throughput applications and that some data may only be required locally as to consider a transaction complete even in the case where some undesirable conditions occur, but these are tolerated to a certain extent.

Two different examples of this is the YouTube 301 views that happened around 2015, where the number of views weren't globally distributed right away. while comments and likes were, this meant two things.
Comments were available globally in real time, while views would also commit globally doing low load times, or off hours to maximize utilization of resources.

another more modern scenario where the real time data distribution and correctness is of the utmost importance is within transactions in the finance industry, where we only want to consider a financial transaction committed once all shards of this data is replicated and the 'old' balance is non longer available, otherwise we could reach a condition where worst case a client is able to draw a negative balance on their account or where one of the transactions isn't recorded as only one of the two transactions is recorded.

State of the art.

The State of the art is Jepsen, or rather one of the State of the Art projects.
Jepsen supports quite a few different modules but the most power full features goes by the name Elle, inferring isolation anomalies from Experimental Observations.

The way this works is be inferring a dependency graph between client side observations and the database version history. This is done by carefully selecting database objects and operations such that the database reads reveals information about the version history.

this means that Elle is able to reveal any anomaly and provide a concise explanation as to why a fault occurred with information of what conditions were present to cause this fault to occur. Using this information we are able to say how a system behaves, what level of isolation and which behavior we'll see as well as what promises the system makes and how does promises are reflected in reality.

And these things can be manifested in multiple ways, some cases are as mentioned earlier in the paper accepted as eventually consistency and performance is valued higher or some applications where loosing some data points are acceptable to allow for higher write throughput.

and the same does for stale, dirty, non-repeatable and phantom reads aren't an issue. this could be on hyper scale social networks where if you see something that was deleted, or if the newest posts, comments and likes aren't distributed across the all notes, but they will eventually. but this is a trade off that is done to allow for much higher write performance as the chance a user is making chancing to the same page on two different nodes/clusters are minimal and the required performance drop required to facilitate the guarantee that all data is always up to date would mean that the system wouldn't scale at the rate it's required, this also brings up another interesting topic, approximate programming, where a program doesn't always have to return the correct output, but if we can see a magnitude speedup of a program and accept a faulty result rate of a few %, then the cost saving and capacity increase could very much be worth it. this could be in the case of Netflix and amazon recommencing products and services, it doesn't have to be perfect and maybe sometimes the result is wrong but if it means we're able to serve a lot more customers before having to degrade services or maybe having it as a step in the service degradation levels to support as many users as possible.

but a lot of this lay outside the scope of this thesis but it's also a super interesting field that will probably see grow in the next decade.

Past jepsen tests

https://aphyr.com/posts/294-call-me-maybe-cassandra

https://aphyr.com/posts/291-call-me-maybe-zookeeper

related work to Jepsen

K. Kingsbury. Knossos. https://github.com/jepsen-io/knossos, 2013-2019.
G. Lowe. Testing and Verifying Concurrent Objects. Concurrency and Computation: Practice and Experience, 29(4), 2017
J. M. Wing and C. Gong. Testing and Verifying Concurrent Objects. Journal of Parallel and Distributed Computing, 17(1-2), 1993.
P. B. Gibbons and E. Korach. Testing shared memories. SIAM Journal on Computing, 26(4), 1997
S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A Complete and Automatic Linearizability Checker. PLDI '10, 2010.
TODO move these to BIB file and write this section

Chapter 4

Modern Datastores

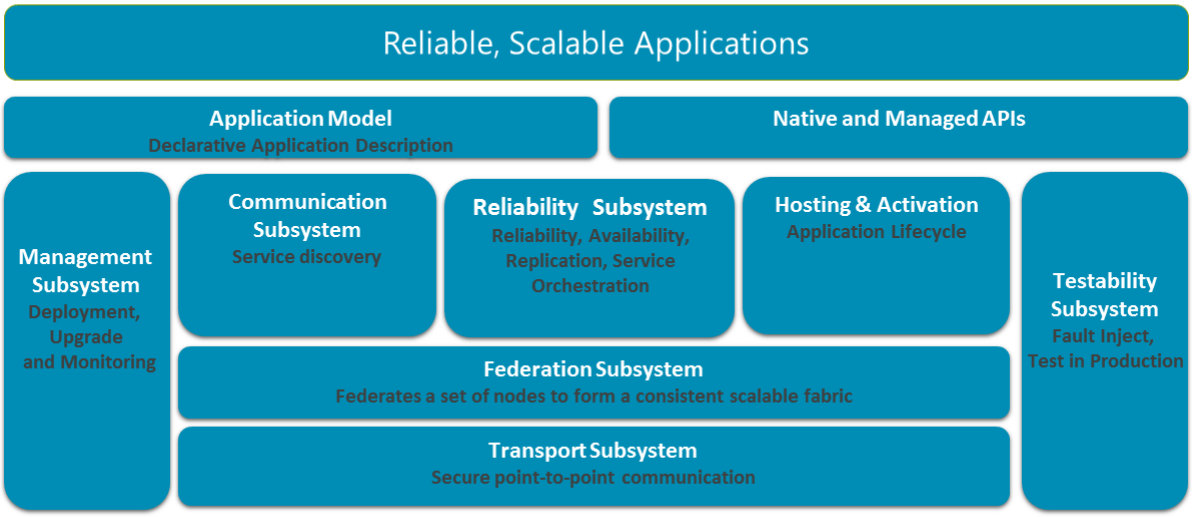
Introduction

The Thesis will primacy focus on Service Fabrics but investigation into other data stores in also done to allow comparisons and investigation into what different models affect the behavior of the database systems.

Service Fabric

Introduction

Service Fabric (SF) is a presented as a distributed systems platform something akin to Kubernetes (K8S), where the user is able to build, deploy and scale micro services and containers, one of the key points presented by on Azure Service Fabric is that you're able to run stateful services. It's presented by Microsoft as the backbone of their core services and data stores.



System Architecture

The architecture behind Service Fabric is built via a layered approach, that by Microsoft's words allow developers to write applications that are highly Available, Saleable, manageable and testable. These

layers are built on 5 Core and 2 supporting subsystems. our main interest lies within the reliable collection so i will only be investigating the layers and services that lay below this.

Transport subsystem

The lowest layer in the core stack that provides secure communication within the cluster itself and between the cluster and clients. This functionality is provided via point to point communication channels that support one way and request reply patterns, in other terms UDP and TCP. This also provides basis for broadcast and multicast within the cluster. Security is handled via wither windows security or x509 certificates. (TODO these might be serving the name purpose and the documentation might just be odd.)

Federation subsystem

The next subsystem in the stack is the Federation subsystem that uses the communication channels provided by the transport subsystem to gather the nodes in the system into a unified service fabric cluster, and provides system primitives that allow for Failure detection, leader Election and consistent routing within the distributed system.

The core of the subsystem is built around a SF-ring that was developed internally at Microsoft in the early 2000s, where both keys and nodes are mapped to a point in the ring, with keys being owned by the node closest to it in the ring. where each node also uses this ring to keep track of its immediate successor and predecessor nodes that it stores in a Neighborhood set, this set is then used by the Federation layer to run consistent membership and failure detection. Nodes also maintain long distance routing partners that are used for consistent routing.

The membership and Failure detection is doing using tow key design principles.

Strongly Consistent membership and Decoupling Failure Detection from Failure Decision

- **Strongly Consistent membership** All nodes must monitoring a given nodes status must agree on weather the node is up or down. for use in the SF ring this means that the all nodes on the node's Neighborhood set must agree on its status.
- **Decoupling Failure Detection from Failure Decision**: failure detection protocols can lead to conflicting states, therefor the decision is decoupled from detection.

To solve the issues of decoupling the failure detection and decision that is used need distribute all decisions to the responsible nodes and ensure that these decisions are all done in a consistent and reliable manner. For the monitoring aspect of this s distributed monitoring and leasing solution is implemented by SF. This implementation solves this via Lease Renewal Request(LR) and LRack. A node is simply required to maintain valid leases from all of it's monitors and these leases have to be renewed every lease period, this leasing period is calculated based on round trip times but is typically around 30 s. If a node fails to renew any of these leases it considers moving itself from the set, and if a monitor misses a request from It it considers marked the monitor as failed. both these decisions need to be approved by a Arbitrator group. if a node fails to receive a LRack within a timeout based on round trip time it repeats the LR until it receives a LRack, Due to the nature of the SF-ring these monitor relations are symmetrical, but this Lease protocol is still run independently, there are however other cases, It a node fails the renewal process is stops renewing leases to any other nodes, or if a node detects a other node as having failed it stops sending renewal request to it. these 2 cases have the potential to cause inconsistencies if they were operating alone, however decision of deciding if a node is down or up is up to the Arbitrator group, which maintains consistent memberships.

The decision of deciding on failures is preformed by the Arbitrator group this is separate from the neighborhood set. They operates independently and have two it has two ways of deciding if a given

node has failed. The way that the groups stays consistent when members join or leave the group is that when a Arbitrator joins the set, it initially rejects all requests in the first t seconds, this is done to prevent new Arbitrator making conflicting decisions with the excising nodes, and to prevent failed nodes to exist in the distributed membership protocol. this ensures that detected nodes leave before being forgotten.

1. X detects Y as failed and sends fail(Y) to the arbitrator
2. The arbitrator Ads Y to a recently failed list and sends ack(fail(y)) containing a timeout for Y to X, if The arbitrator already marked X as having failed and ignores the request,
3. If X receives this request it waits for the TO to claim Ys area of the ring. and if re receives no response within the timeout it leaves itself.

If a node is already in the recently failed list it returns the same response as the the first reporter except it calculates in time since first detection so the portion of the ring is claimed by the neighbors at the same time. This also means that the routing can continue after to with a laxity added. as all routing request for a node are added to a queued if a node is marked a failed. and the queue is released after TO+laxity that allows a neighboring node to claim that part of the ring and allows to these routing request to be preform when by it's successors.

If the case of two nodes report each other as failed the conflict is resolved either by majority, eg the node that was reported first to the most Arbitrators leave or an alternate variant that heals the membership if both nodes are healthy and allows both to stay. in cases of network congestion or partitions that result in multiple nodes detection each other as failed, in traditional distributed hash tables this can result in inconsistencies in the membership lists or ring.

The implementation in SF is presented as being failure tolerant towards cascade failures as the decision isn't made by the detectors themselves but by arbitrators. an example is used that if a given node is only dependent by it neighbors and of them fails to renew it's lease it will leave itself. No mention here is written about how the node handles being fragmented into two partitions and rejoined later, but mentions that this approach scales to entire data centers, no mentioned about how it scales across regions or outside of data centers.

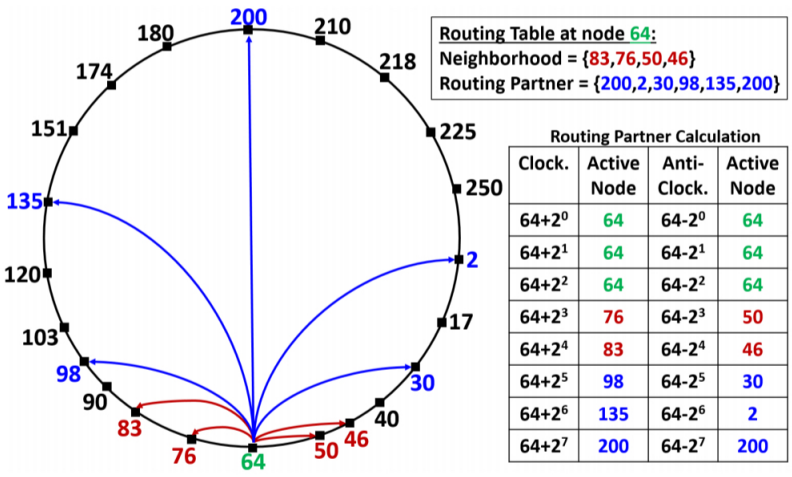


Figure 5: Routing Table *of node 64. The ring has* $2^{m=8}$ *points. Numbered dots represent active nodes.*

SF-Ring and consistent routing.(need to rewrite/reword this to a more coherent sentence.)
The Distributed Hash table used for service fabric routing is a evolution of SF-ring that was developed within Microsoft in the 2000s. The routing used offer symmetry and due to this it allows bidirectional

routing that allow for lookup and routing via binary search, this allows SF to in practice routes message around the ring in a fast manner, with multiple routing options in that there always exists 2 routing partners on either side of the destination that both helps distributed the load and is able to route even with stale routing tables. Initially all packages were routed using this method however this has changed over the years, today the hash table is used to build a routing table and maintain it when a new node is added to the cluster, and ti route message to virtual addresses. After discovery a direct route is used via the destination IP address.

Routing tables work differently depending on the size of the cluster. however it is able to route in two ways, If the size of the routing table is smaller then the number of nodes a direct route is used, if there exist more nodes in the cluster then the routing tables it stitches to routing via routing tables to reduce memory consummation but causing an increase in time complexity to O(Log(n)) from O(1)

Routing Tokens. Each node owns a token that includes a portion of the routing token it is responsible for. The sf-ring protocol ensures two properties, That each token is only owned by one node at any given time, and that eventually every token is owned by 1 node. This is ensure in the way that SF handles nodes joining and leaving the ring. Initially the bootstrap node owns the entire ring. after this inertial bootstrap node any joining node will split the ring segment between them.(TODO I was unable to find information on this precisely is handled however from the wording in the paper the initally split would be 50/50 and a third node would be 50/25/25? is it split evening it 33/33/33 but a Fourth node would be 22/22/22/33 that would cause unbalance unless some shifting is done)

These routing tokens are also used for leader election, Simply the owner of a given key is the owner.

Reliability subsystem

The reliability subsystem is in charge of load balancing, replication and availability. These three aspects are provided via 3 services, Failover Manager, Naming and Resolution, and a Placement and Load Balancer service.

The Failover Manager is running as stateful service, that has a instance running on each node in the cluster, that provide 3 actions, start a replica, move a replica, reconfiguration.

- **Create a replica** Create a new replica when instructed by the PLB
- **Move a replica** migrate a replica when instructed by the PLB to a different node.
- **Reconfiguration** If a primary replica becomes unavailable to promotes a secondary as the new primary, in case the old primary comes back only it is demoted to secondary.

A service called Failover Master Manager is also running that is able to restart the failure manager via a cached state in case it fails. in case the master fails it is able to rebuilt its state via the SF-ring, the Master Failover manager is running on the node who's token range contains ID0.

The naming and resolution service maps instance names to endpoints that running services are listening on, this also allows for consistent routing from outside of the cluster via URI that doesn't change over a their lifetimes.

The Placement and Load Balancer(PLB) service is stateful and is in charge of placing replicas and instances of services at nodes and ensure even load throughout the cluster. they claim that unlike other solutions where services are hashed onto the ring in SF the PLB explicitly assigns each services replicas Primary and Secondaries to nodes in the Ring. it continually monitors the cluster for available resources to both assigned and migrate services to underutilized nodes and migrates services away from a node that is about to be upgraded or is overloaded due to a long workload spike.

The technique used for selecting placement of nodes is done via Simulated Annealing, as it is presented to provide a near optimal solution as to where services can be placed, This way the system is modeled

is via resource use in the system where a even load is desired, but some constraints need be met such as fault tolerance, avoiding replica co location and some service might have strict set of nodes to run on.

Reliable Collections

Reliable collections provide stateful services in Service fabric via Reliable Directories and reliable Queues that are available for C# and java programming, and are promised to be:

- Available and Fault-tolerant via replication,
- Persistent via disk
- Efficient via asynchronous via API that are non blocking.
- Transnational via APIs with ACID semantics.

One of the key differences between storage systems build on SF and other highly-available systems is that states are kept locally in the replica while also being made highly-available, this causes most reads to be local.

Writes are relayed from the primary replica to secondary replicas via passive replication and are considered complete when the majority of secondaries acknowledges it. the relaxations allow an application to achieve weaker consistency by relaxing where a read can go, eg always read from primary to read from secondary.

SF is presented as the only "self-sufficient microservice system that can be used to build a transactional consistent database which is reliable,available, self-*, and upgradable because the lower layers assure consistency"[**SFpaper**]

Consistency models/isolation levels SF reliable collections promise two Select-able consistency models that the user can pick. Repeatable Read and Snapshot isolation. eg there is no promise about Serializability of the transactions.

	Role	
Operation	Primary	Secondary
Single Entity Read	Repeatable Read	Snapshot
Enumeration, Count	Snapshot	Snapshot

Table 4.1: isolation level defaults for Reliable Dictionary and Queue operations.
[**SF_RC_Transactions**]

Programming for Service Fabric

Comparison

In many of the papers published by Microsoft a lot of comparisons are made to Cassadra, Redis and Redis Therefor comparisons will be made to these as they are compared in a lot of aspect, the investigation won't be as deep as with SF but investigation of consistency and failure modes will be be done.

Cassadra

Redis

Dynamo

Chapter 5

Jepsen Test on Service Fabric Reliable collections

Introduction

The Goal of the thesis was to attempt to preform a Jepsen test on service fabric with it's reliable collection as the data store feature that powers a lot of the services being the target of this endeavor. To preform such a test we need to build up both a testing suite, APIs and libraries to allow for a connection between the Jepsen service and the Service fabric core components. there for a few different applications and services need to be learnt, understood, designed, implemented, test, executed, and at last the data generated from this can be analyzed.

- Learning the required tools to implement the services and applications in the first place. C#, Clojure, and Service Fabric itself as well as reliable collections and lastly Jepsen itself
- Understanding the problem, requirement, topology, architectures and how we want to test these things
- Designing the Services, Libraries, APIs, testing suite, and the infrastructure and networking of the cluster and azure.
- implement the Services, Libraries, APIs and testing suite required for things to talk to each other.
- Test all the code and configurations and ensure things work as intended.
- execute the test itself
- analyse the data from the test.

I will only write deeper about designing, implementing testing, executing and analysing the results. however this does not mean that the learning and understanding phase of the project was any smaller, quite the contrary, Learning 2 new programming languages that I've never interacted with before isn't a easy task, adding in the complexity of write code for cloud framework that i had no prior experience working on didn't help either. spicing it with up with outdated documentation, incompatibility between versions of the framework and lack of debugging tools and broken tools meant that a lot was done via eyes blind brute forcing solutions to a lot of issues due to lacking features on the Linux version of the cluster.

Design & implementation

The Design of the services required for preforming the Jepsen test were designed on the fly as no prior experience was had with Service Fabric, Jepsen, C# or Clojure.

Initial Design

The initial design of the of the testing suite contained two applications, A stateful service Fabric application and a Clojure application running the Jepsen test.

Service Fabric Statefull Service

The Service Fabric stateful service should contain the desired operation that our jepsen test can interact with, the first designed endpoints were insert/add, Update, Check and set, delete and delete all on a reliable directory class.It would be implemented using C# as the lack of documentation for java made this this seem easier in the beginning and due to all documentation being in C#. The service would expose these end point via a web endpoint and kestrel was selected for this as it was a native provided already integrated into SF and was the one most ex samples used and was therefor deemed the safest choice to proceed with.

A diagram of the service looks as the following.

Infrastructure

Node Types

SF node

Jepsen Host name

Services

SF services

Reliable collections service

Front end

Jepsen Service

Considerations

Implementation

Designing the test started with a basic idea of what was needed in terms of services and infrastructure was 2 applications. a Service fabric stateful service that made itself available to the Jepsen service via an API and the jepsen Service itself running on a separate machine in the cluster that would control the cluster nodes via ssh as well as in charge of sending the required requests to the service. These assumptions was where the first code was written to both learn C# and clojure as well as learning the required frameworks needed as well.
This turned out not to be the case as SF only allows writes to the primary and if you need more primaries you need to add a 2nd layer of services to both forward write requests to the correct portion or to query all partition if data is needed from the whole cluster. Then the issue arouse that a lot of my Clojure code was fairly hard coded which often broke when changes in the cluster occurred, combined with

SF Services

Local Dev setup

docker run -i -d -p 19000:19000 -p 19080:19080 -p 80:80 -p 25102:25102 -p 25112:25112 dffdac13e6ad

Issues

Linux and windows versions being far from interchangeable and applications and services that function on windows/local machines fail crash on deployed cluster
visual studio being unable to load .dmp files to debug crashed applications from the cloud.
Visual studio remote debugger not working

Azure

Configuration issues,
Cluster cecoming unhealty/corrupt and nodes not being configred correctly after a reimage.

Jepsen Service

Execution of the test

Steps.
1 start Service Fabric cluster in azure. 2. Add Jepsen node to Vlan of the cluster. 3. Deploy SF application to the cluster. This is currently done via Visual studio but could also be automated from the jepson node. 4. Deploy Jepsen code to the Jepsen node. 5. Run Jepsen test. 6. Retrive Test data from test node. 7. Clean up.

Results

Chapter 6

Related Work

Chapter 7

Conclusion and Outlook

Conclusion

Outlook

Future Work

Atomic clocks ?

Chapter 8

Appendix

Proposal

'Jepsen methods usage for ACID compliance in Hyperscale Cloud Frameworks'

Introduction

The goal of this project is to assess how Jepsen[1] tests allow us to verify the properties of ACID[2] in a cloud system. The Jepsen test is a method used to evaluate the compliance of a system in relation to the ACID properties. This is done is by analysing the system via Blackbox[3] tests , in the sense that the internals of the system work are not relevant. We look at the service from a client-side and not any underlying structures or frameworks. The ACID properties are

- Atomicity
  Guarantees that each transaction is treated as a single "unit", which either succeeds completely or fails completely
- Consistency
  Ensures that a transaction can only bring the database from one valid state to another
- Isolation
  Ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially
- Durability
  Guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure

[2]

We plan to apply these methods on Service Fabric[4] , to verify if this system is compliant with the ACID properties. Service Fabric is a framework developed by Microsoft, designed to allow developers to build Hyper-Scale Cloud (HSC) deployments on the Azure[5] platform. The motivation behind this thesis is to study the ACID properties in an HSC environment and evaluate how the constraints of ACID hold up in practice, as well as verifying if these constraints are met, and more importantly if they are not.

[1] https://aphyr.com/tags/jepsen
[2] Database Management Systems by Raghu Ramakrishnan & Johannes Gehrke ISBN13 9780071231510
[3] https://youtu.be/tRc0O9VgzB0?t=293
[2] Database Management Systems by Raghu Ramakrishnan & Johannes Gehrke ISBN13 9780071231510
[4] https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-overview
[5] https://microsoft.com/en-us/azure/

Plan

The project is segmented into three parts. The writing of the report will take place at all times throughout the process, and the final phase should be a finalization and corrections phase.

The study phase

- The first part of the study will focus on analysing the previous[6] Jepsen tests performed on other systems. This will allow us to define a strategy in terms of: which tools, tactics and methods are worth investigating, and what type of faults should be taken note of. This information is useful for two aspects:
  - where our focus should be when testing,
  - which tools and packages we should be familiarize with.
- The second part of the study will centre around "Service Fabric" and getting to know how the framework is coupled together, what vectors we can access the system from, and how we can manipulate the framework to induce fault conditions.

The experimentation phase

- In the experimentation phase, we will design and perform tests on the system. If any faults occur, we will attempt to locate where, and why they occur. If the scope and the timeline of the project allow, we will assess whether we prevent the faults from occurring in the future.
- The Experimentation phase of the project will include the steps described below.
  - Planning of the test, selection of the aspects of the system to be tested, consulting with developers and users of the system to determine if and where undesired behaviour may occur.
  - Designing the tests and setting up the tools to facilitate those tests.
  - Writing the tests, verifying they work as intended and setting up a data collection framework to collect the data in a manner that allows our analysis.
  - Performing the tests on the system in different scenarios, normal conditions and different levels of faults and disaster recovery.
  - Analysing the data for faults, errors and inconsistencies.
  - Consulting with DEVs to determine where the faults occurred, and which failures or bugs led to these fault conditions.
  - Concluding whether "Service Fabric" is ACID-compliant.

The report phase

- The final phase of the project aims at finishing an initial draft, reviewing and correcting it, and finalizing the report for hand-in.
- Deliverables
  - At the end of the project, there should be a report, the tests and the results from those tests.
  - The report written in English and following the standard academic writing conventions will include
    - A study on Jepsen tests, describing what they are, as well as why and how they are done.
    - An experiment in which we will analyse "Service Fabric"
    - A discussion on the conclusiveness of the test and the compliance of the framework with service fabric.
  - The code & data will include
    - Scripts and code for tests and for analysing the data.
    - Relevant results and data from the tests.

[6] https://jepsen.io/analyses

Goal

The goal of the project is to deep dive into Jepsen tests with a focus on “Service Fabric” as the subject. The optimal goal of the project is to verify if the database aspects of “Service Fabric” comply with the ACID properties, and to locate the fault cases in case they don’t. Risk assessment There are a few risks in the project. In the case no faults are found within the Service Fabric, this reduces the scope of the project. If no faults are found, time allowing, different framework can be analysed. On the contrary, if the number of faults in the framework is more extensive than expected, then the scope of the project may grow uncontrollably. In this case, choices will be made to select a subset of the data to focus on.

Schedule

Mark Jervelund thesis plan			
November	Week 45		Study Jensen test
	Week 46		
	Week 47		Finish study on Jepsen test
	Week 48		
	Week 49	Structured study 8 weeks	Study Service Fabric
	Week 50		
December	Week 51		Christmas break
	Week 52		
	Week 53		Finish Study Service Fabric
January	week 01		
	Week 02		designing the experiment
	Week 03		
	Week 04		designing the experiment
	Week 05		
February	Week 06	Experiment 10 weeks	Execute the experiment
	Week 07		
	Week 08		Analyse the experiment
	Week 09		
March	Week 10		discover what the findings from the experiment is
	Week 11		
	Week 12		Running additional experiments if needed Full on write mod
	Week 13		
April	Week 14		Full on write mode
	Week 15		
	Week 16	Thesis writing 10 weeks	Full on write mode
	Week 17		
May	Week 18		Full on write mode Send out thesis for feedback
	Week 19		
	Week 20		Correction
	Week 21		
June	Week 22	goal	Hand in
	Week 23		
	Week 24		
	Week 25		
	Week 26		
	Week 27		
	Week 28		
	Week 29		
	Week 30		
	Week 31		

Chapter 9

REMOVE THIS BEFORE HANDIN !!!!

Things to read(temp list)

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf
https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-reliable-collections