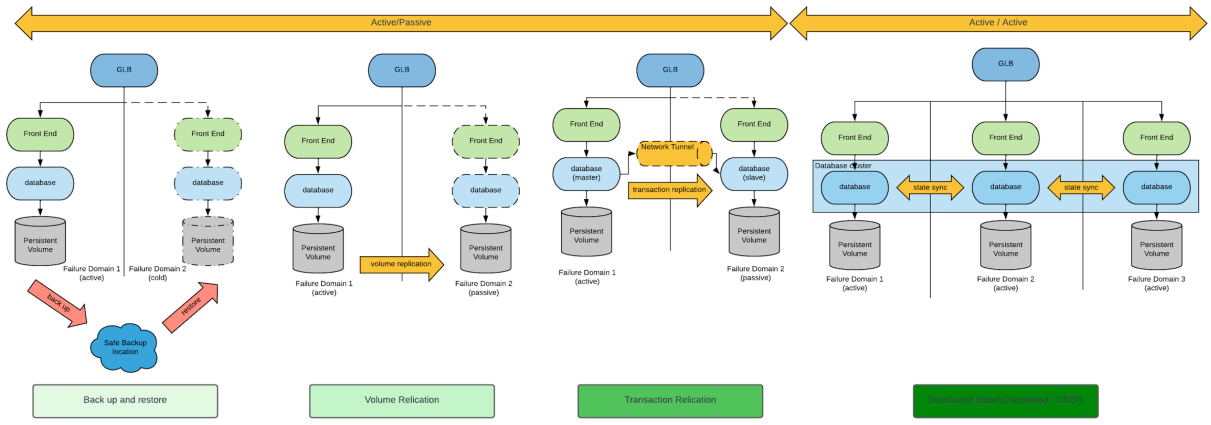
Cloud Native Disaster Recovery for Stateful Workloads

Introduction

This paper discusses disaster recovery strategies for a cloud-native environment.
We will review some archetypal approaches and then dive into the approach we call Cloud Native Disaster Recovery that most enterprises should strive for.
A word of caution: disaster recovery is a complex subject. The applications in a single company might use different disaster recovery strategies and rely on each other to recover properly in case of a disaster. This kind of complexity is difficult to capture and treat in a document.
The archetypal approaches will be discussed individually to make this topic more approachable. Additionally, we will only touch lightly on cost considerations and try to focus on the architectural and technical performance of the approaches we present.

Architectural Approaches to Disaster Recovery

The following picture summarizes the four archetypal disaster recovery approaches:



The picture shows a single application: a simple database with a front-end. We show how disaster recovery can be set up for the same application in four different approaches.

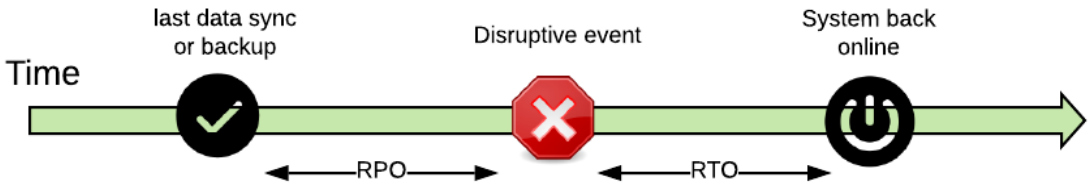
One thing to note immediately is that we don't need to know what the infrastructure supporting the application is to be able to discuss its disaster recovery strategy. Oftentimes, organizations base the disaster recovery strategy on the underlying infrastructure capabilities, forcing all the applications that run in the platform to receive the same treatment (one size fits all DR). We

argue that if you are building a platform to run applications in your company, your platform should provide capabilities to the application owners such that they can decide how to organize the disaster recovery strategy from their application. This is what the large cloud providers do, they give building blocks to create your own strategies, but they don't force you into a specific one.

Specifically, if you are building your platform on Kubernetes as your container runtimes, your disaster recovery strategy should not be focused on recovering Kubernetes clusters. Kubernetes clusters should be disposable and easily provisionable and configurable and the platform should provide self-service capabilities to the application teams for them to design their own disaster recovery strategy.

Before we delve into each of these strategies specifically it's good to know about the two main Key Performance Indicator (KPI) used to measure the performance of a disaster recovery strategy:

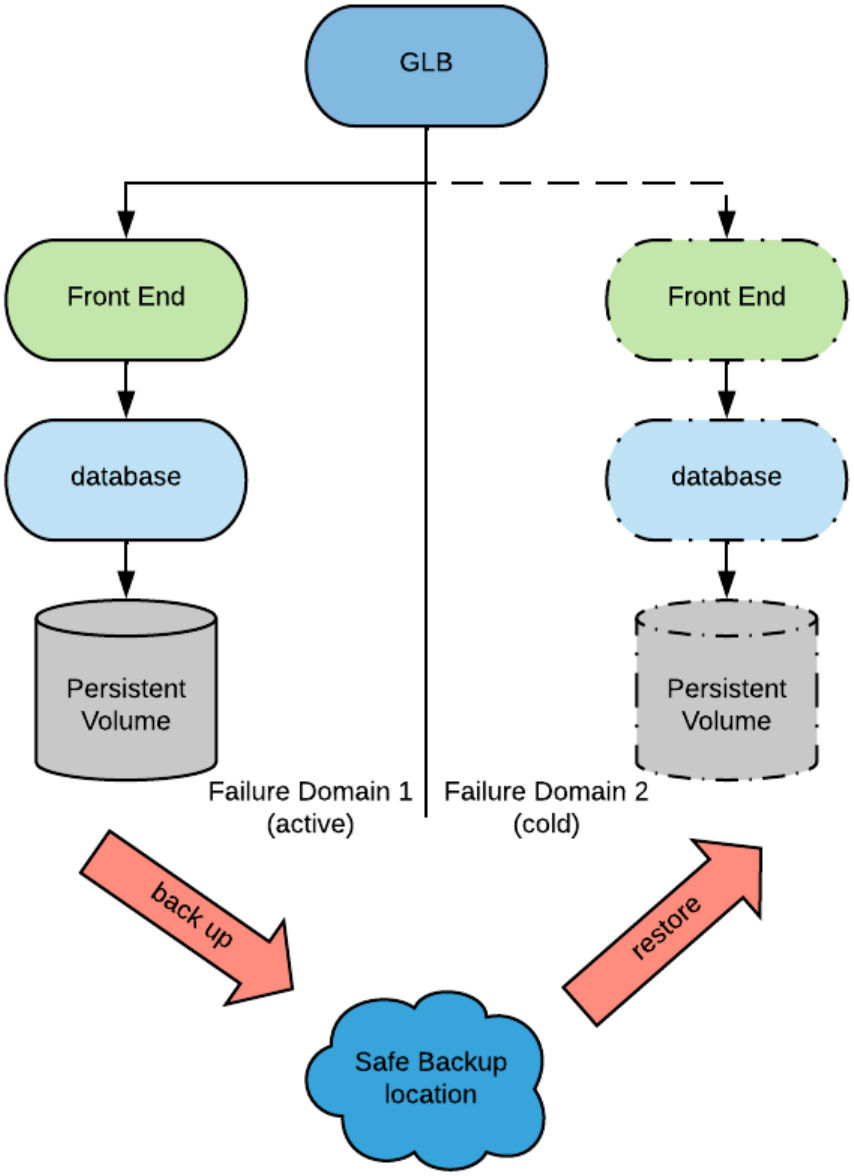
- Recovery Time Objective (RTO): the time it takes to have systems back online after a datacenter fails.
- Recovery Point Objective (RPO): time interval of state loss from the last saved state to the time the datacenter fails.



Let's now analyze these approaches individually. In the spirit of building a platform with the correct capabilities, we will also highlight which technical capabilities are needed to set up these approaches.

Backup and Restore

With backup and restore, we take periodic backup of the stateful workload volume(s).



This approach is obviously an active/passive approach and implies that the RPO of this approach is the time interval between the backups. While the RTO is the time that it takes to restore and restart the application on the secondary failure domain.
While this approach is relatively simple and inexpensive, backup solutions become harder at scale when consistency groups are a requirement.
The reason why back and restore is relatively inexpensive is that the secondary site can be cold (hinted in the picture by the dotted lines). That is nothing has to be running. This reduces resource and licensing costs.

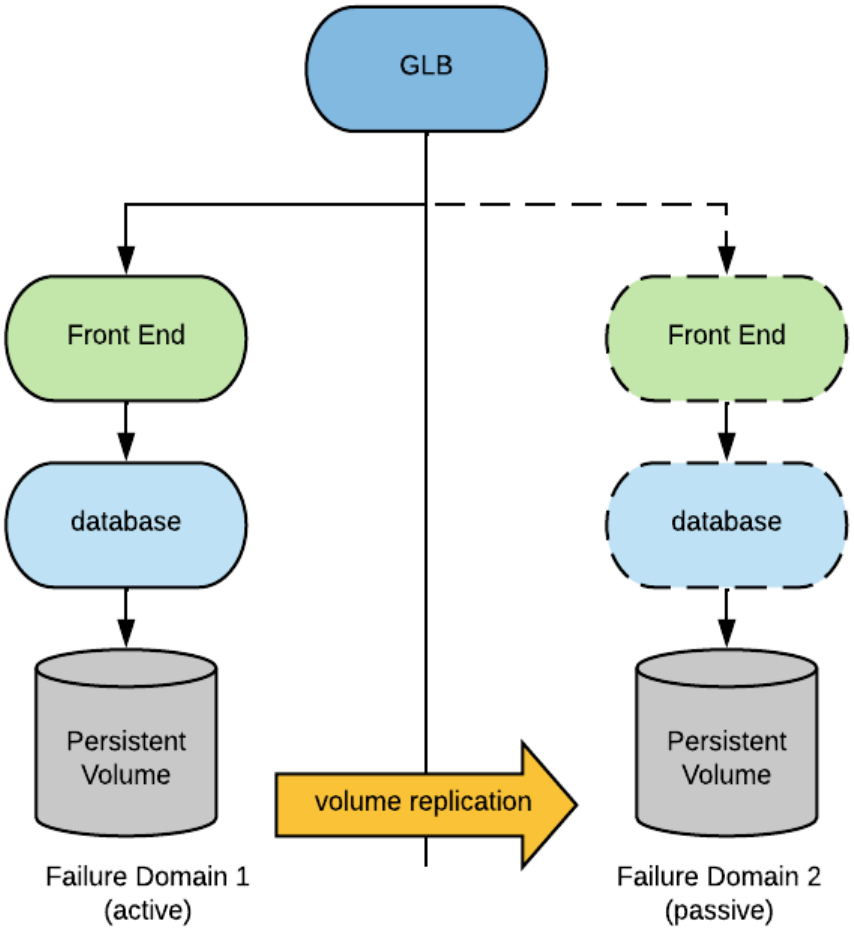
Overall back and restore offer the poorest performance in terms of RPO and RTO making this approach not suitable in most scenarios.
That said backups and restores are still needed for other data protection use cases outside of disaster recovery (e.g. logical errors, data level malicious attacks etc…).
For this reason it is recommended to build a backup restore capability in one's platform, it's just likely not going to be used for disaster recovery.

The technical capabilities that we need to make available if we are building a platform that support back and restore are:

1. Ability to configure a global load balancer with primary and secondary sites. There should be a way to switch from primary to secondary.
2. A storage infrastructure with support to back up and restore and the ability for the user of the platform to self-serve the configuration of the backup schedules.

Volume Replication

With volume replication, volumes on the primary site are configured to be replicated to the secondary site (another active/passive approach).
Replication can be synchronous or asynchronous.



Synchronous replication may introduce latency, but keeps the RPO to zero (and crash-consistency).
Asynchronous replication does not introduce latency, but allows for RPO to be greater than zero and conceptually unlimited. RPO depends on the depth of the queue of the data waiting to be synched. That is a function of the change rate and the latency between the two failure domains. In either case RTO will amount to the time necessary to switch the global load balancer and to restart the services in the secondary site pointing to the copy of the volume. Because of the crash-consistency nature of the volumes, the database load may take longer than under a clean volume situation (a.k.a. application level consistency).

Volume replication requires a storage product capable of setting up replicated volumes and it is highly sensitive to latency.

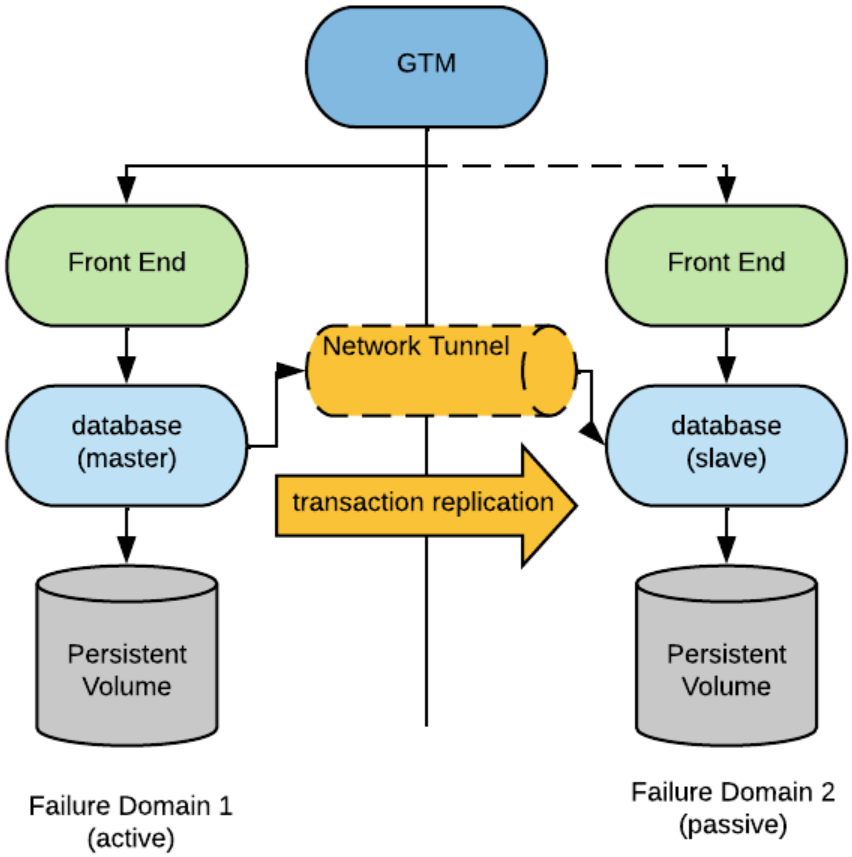
When Kubernetes is involved, volume replication requires a CSI product that allows tenants to configure volume replications across clusters. This may sometimes be tricky as normally

Kubernetes will assign a new volume to a PVC, while now we need to be able to specifically indicate which volume to assign (the replicated one).

As for backup and restore, a global load balancer is required for this approach to work.

Transaction Replication

Transaction replication is the last of our active/passive approaches.



With transaction replication, transactions or write requests are propagated from the primary site to the secondary site by the middleware itself (or the stateful workload). Most incumbent databases and messaging systems can work this way (a.k.a. master/slave or primary/secondary configuration).

Transaction replication can be synchronous (zero RPO) or asynchronous (allowing for greater than zero RPO). Transaction replication requires the stateful workload to be on also on the

secondary site, but because of optimizations possible only at the application layer, it is less sensitive to latency than volume replication.

That said, volume replication and transaction replication have similar performances with regards to RPO and RTO.

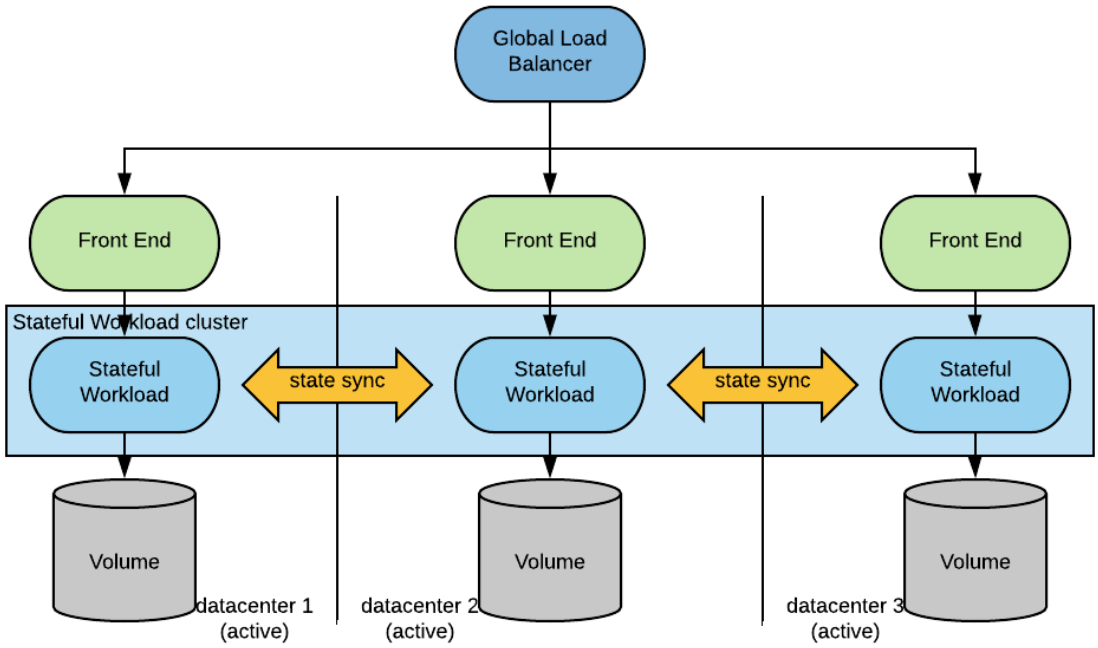
In terms of capabilities, transaction replication requires a global load balancer (as all the approaches do) and an east/west communication path between the primary and secondary failure domains.

Differently from backup and restore and from volume replication, with transaction replication we don’t require any special capabilities from storage, but we need some capability from the networking layer. Also the stateful workload is here in charge of the data replication, moving the ownership of the disaster recovery procedure more squarely in the application team domain (assuming they own their middleware).

In Kubernetes-based solutions, the east-west communication path capability may not be easy to make available. In Kubernetes pods run in a non-externally routable SDN. To make these east-west paths available one may have to use a CNI product capable with this feature or enhance the CNI product with a network tunnel capability.

Distributed Stateful Workload

Distributed stateful workloads allow for active/active deployments.



With distributed workloads the middleware is in charge of replicating transactions. But in this case any workload instance is able to take write requests and will coordinate with the other instances to confirm that transactions have been replicated, or at least a quorum of them.

Distributed stateful workloads guarantee zero RPO and a minimal RTO, typically a heartbeat timeout between the instances which allows the system to realize that some instances are lost .

As for the transaction replication approach, distributed stateful workloads do not count on any specific capabilities from the storage layer, but require the ability to send east-west traffic across failure domains.

Distributed stateful workloads are sensitive to latency. In particular one can expect the latency for a write transaction to be at least twice the latency of the closest failure domain. Over great distances this can impose severe limitations, so this architecture is not for all use cases.

As for the transaction replication approach, the disaster recovery procedure here is handled mostly by the stateful workload and therefore is owned by the team that manages the stateful workload, typically the developer team. However in this case, no human intervention is needed during a disaster. The system simply reorganizes itself.

Summary

The following table summarizes the salient characteristics of the approaches we have examined:

	Backup/Restore	**Volume Replication**	**Transaction Replication**	**Distributed stateful workloads**
Architectural style	Active/Passive	Active/Passive	Active/Passive	Active/Active
Disaster recovery process trigger	Human	Human	Human	Automatic
RTO/RPO	RTO: hours RPO: frequency of backups	RTO: minutes RPO: zero for synchronous replication, unbounded for asynchronous replication	RTO: minutes RPO: zero for synchronous replication, unbounded for asynchronous replication	RTO: seconds RPO: zero
Disaster Recovery process ownership	Infrastructure team, particularly storage team	Infrastructure team, particularly storage team	Owner of middleware, typically developer team	Owner of middleware, typically developer team
Required Capabilities	**Storage**: backup and restore **Networking**: global load balancer	**Storage**: volume synchronizing **Networking**: global load balancer	**Networking**: global load balancer, east-west path	**Networking**: global load balancer, east-west path

As one can see from this summary table, the distributed stateful workload approach provides the best disaster recovery solution when it is possible to implement. We call this approach also Cloud Native Disaster Recovery.

We adopted this name for two reasons:

- Setting up distributed stateful workloads is more complex, but with cloud era automation technologies, such as GitOps, this level of complexity is now manageable.
- With the cloud it is more likely to have access to three failure domains, which is a prerequisite for this approach to work.

In the remainder of this document we will examine the theory and the technology that make distributed stateful workloads and therefore cloud native disaster recovery possible.

Considerations on Availability and Consistency

A distributed stateful application needs to deal with Availability (the ability to successfully serve requests) and Consistency (the property of keeping state consistent across the various instances that constitute the distributed workload). There is a significant amount of literature around these concepts, here we are going to recap what is important for the sake of our disaster recovery conversation.

Failure domain

Failure domains are areas of an IT system in which the components within that area may fail all at the same time due to a single event.
Examples of failure domains are: CPUs, boards, processes, nodes, racks, entire kubernetes clusters, network zones and data centers.

As one can ascertain from these examples, failure domains exist at different scales.
When deploying a distributed stateful workload, one should consider the various failure domains at hand, and make sure that the various instances of the stateful workload are positioned in different failure domains.

In Kubernetes, there are standard node labels (topology.kubernetes.io/region, topology.kubernetes.io/zone,and kubernetes.io/hostname) to capture the idea of failure domains in a cluster. Designers of stateful workloads should consider creating anti-affinity rules based on those labels when packaging their software to be deployed in Kubernetes.

High Availability

High Availability (HA) is a property of a system that allows it to continue performing normally in the presence of failures. Normally, with HA, it is intended the ability to withstand exactly one failure. If there is a desire to withstand more than one failure, such as two, it can be written as HA-2. Similarly, three failures can be written as HA-3.

The foundational idea of HA is that the Mean Time to Repair (MTTR), a failure must be much shorter than the Mean Time Between Failures (MTBF) (MTTR << MTBF), allowing something or someone to repair the broken component before another components breaks (two broken components would imply a degraded system for HA-1).

It is often understated that something needs to promptly notify a system administrator that the system has a broken component (by the very same definition of HA one should not be able to determine a degradation solely by the normal outputs of the system).

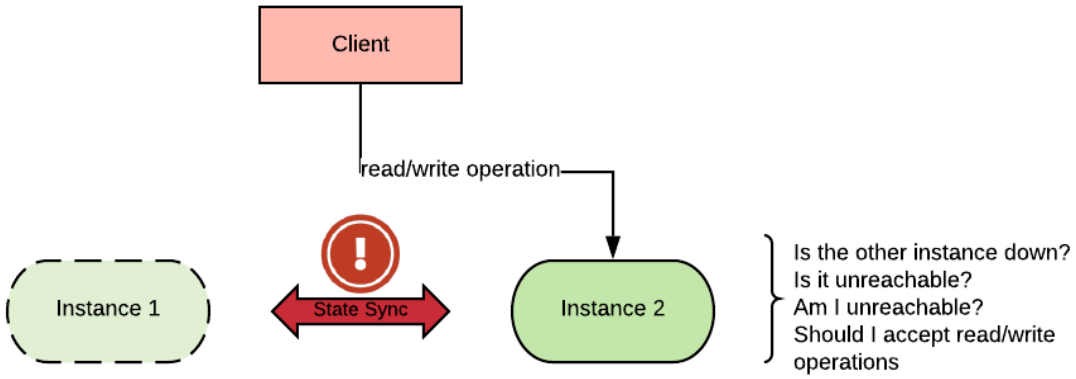
As a result, a proper monitoring and alerting system must be in place. Otherwise, an HA system would just keep functioning until the second failure occurs (~2xMTBF) and then still be broken, defeating the initial purpose of HA.

Given a failure domain, HA can be thought of as answering the question: What happens to our workload when one of the components of this failure domain breaks?

With regards to stateful workloads, HA implies that one needs multiple instances (at least two) of each workload, and that the state of these instances needs to be replicated between them.

If, for example, one builds a stateful system with two instances and instance A suddenly cannot contact instance B, instance A will have to make a decision whether to keep working or not. Instance A cannot know whether instance B is down or healthy-but-unreachable. It is also possible that instance A is unreachable. This is known as a split brain scenario.

In practice, in a distributed system, failures are indistinguishable from network partitioning where the presumably failed component has become unreachable due to a network failure.



If a piece of software is designed to keep working when the peers are unreachable, its state may become inconsistent. On the other hand, if a piece of software is designed to stop when the peers are unreachable, then it will maintain consistency, but will not be available.

Consistency

Consistency is the property of a distributed stateful workload where all the instances of the workload “observe” the same state.
The realization that by temporarily relaxing consistency, one could build stateful workloads that horizontally scale to a theoretically unlimited size gave birth to a Cambrian explosion of eventually consistent workloads. Typically these workloads expose a NoSQL interface (as the SQL interface is associated with strict consistency), however that is not necessary.

When an issue arises in eventually consistent workloads, two or more sections of the cluster are allowed to have a different state (drift) and to continue serving requests based on the state understood by each member of the cluster. When the issue is resolved, a conflict resolution algorithm ultimately decides which of the available conflicting states wins. This process can take some time, but it is guaranteed to end as long as no other changes occur.

Eventual consistency is not suitable in every scenario (for example financial applications often need to be strictly consistent), and even when it's applicable, there are several areas of concern including:

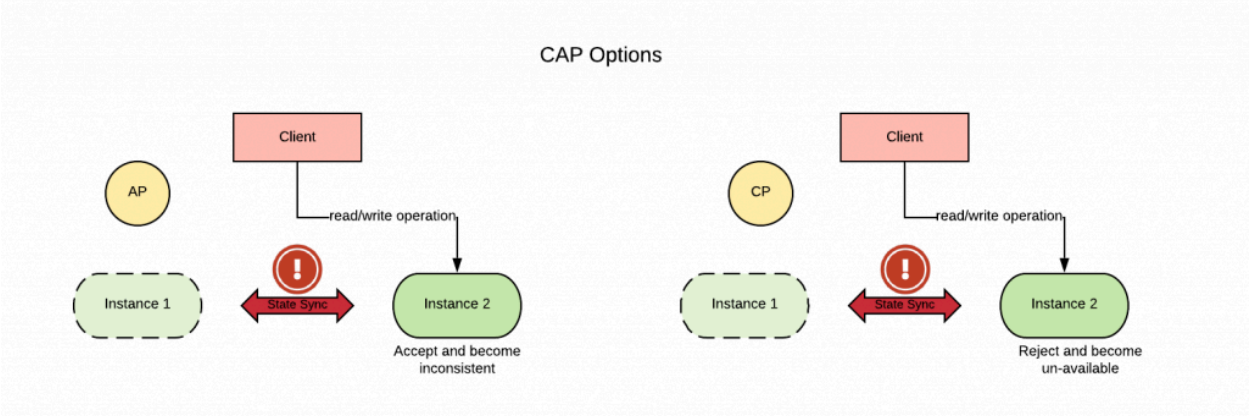
1. No SLA that can be placed on how long diverged states will take to converge. In situations where the state keeps changing rapidly, the time that it takes to catch up may be lengthy or never resolve.
2. Eventual consistency does not mean eventual correctness. While after the conflict resolution phase takes place all instances will be in a consistent state, there is no guarantee that they will end up in the correct state given the logical requirements of the business problem at hand.

The realization of the second point mentioned above has pushed many organizations to seek strictly consistent and available solutions.

The CAP Theorem

The relation between consistency and availability for distributed stateful workloads is formalized in the CAP theorem. Simply put, the CAP theorem states in case of network partitioning (P), one can choose between consistency (C) or availability (A), but cannot have both.

During a network partition, the stateful workload will need to operate in a degraded state: either read-only if the application chooses consistency, or inconsistent if the application chooses availability.



A corollary of the CAP theorem called PACELC (if Partition, then either Availability or Consistency, Else then either Latency or Consistency) states that under normal conditions (absence of a network partition), one needs to choose between latency (L) or consistency (C). That is to say that under normal circumstances, one can optimize for either speed or consistency of the data, but not for both.

The following table illustrates several stateful workload and their choice in terms of PACELC Theorem:

Product	**CAP Choice** (either Availability or Consistency)	**PACELC Choice** (either Latency or Consistency)
DynamoDB	Availability	Latency
Cassandra	Availability	Latency
MySQL	Consistency	Consistency
MongoDB	Consistency	Consistency

Source wikipedia, see the link for more examples.

The definition of network partition is described with mathematical precision by the CAP theorem and goes beyond the scope of this document, however an approximate but good mental model is the following: if the strict majority of instances can communicate with each other, there is no network partitioning. Otherwise, a network partition has occurred.

So, in terms of HA (i.e. when we account for one failure), if there are three or more instances of a stateful workload, for the CAP theorem we can have both availability and consistency. In general, if the stateful workload is deployed across three or more failure domains, it can be designed to be always available and consistent with respect to the failure of one of those failure domains.

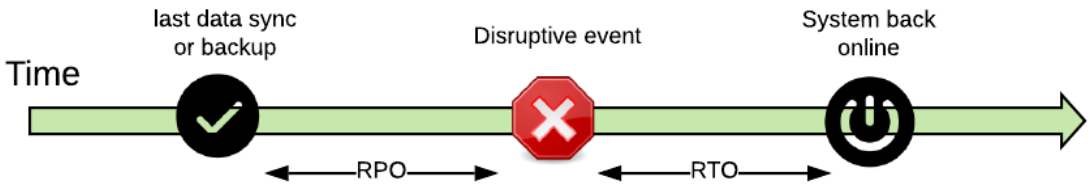
Disaster Recovery

Disaster recovery (DR) refers to the strategy for recovering from the complete loss of a datacenter. The failure domain in this situation is the entire datacenter.

Given a failure domain, DR can be thought of as answering the question: What happens to the workload when all of the components of this failure domain break?

Disaster recovery is usually associated with two metrics:

- Recovery Time Objective (RTO): the time it takes to have systems back online after a datacenter fails.
- Recovery Point Objective (RPO): time interval of state loss from the last saved state to the time the datacenter fails.



In the old days, these metrics were measured in hours, and required that users followed a set of manual steps to recover a system.

Most DR strategies employed an active/passive approach, in which one primary datacenter was handling the load under normal circumstances and a secondary datacenter was activated only if the primary went down.

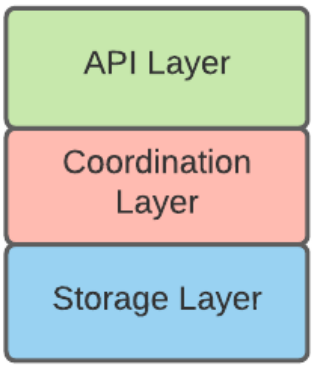
But, having an entire datacenter sitting idle was recognized as a waste. As a result, more active/active deployments were employed, especially for stateless applications.

With an active/active deployment, one can set the expectations that both RTO and RPO can be reduced to almost zero, by virtue of the fact that if one datacenter fails, traffic can be automatically directed to the other datacenter (through the use of health checks). This configuration is also known as disaster avoidance.

Given the discussion of the CAP theorem, to achieve a disaster avoidance strategy where the stateful workload is always available and consistent, one needs to spread the workload across at least three data centers.

Anatomy of a Distributed Stateful Workload

An argument can be made that all distributed stateful workloads share the same logical internal structure because, after all, they are all trying to solve the same complex problem: keeping a shared state consistent while at the same time processing requests in an efficient way. Granted that actual implementations can greatly vary, the following diagram represents the logical internal structure of a distributed stateful workload:



API Layer

The API layer is the component that exposes the externally visible functionality of the distributed stateful workload. This layer deeply characterizes the kind of workload:

- Block device API (iSCSI, FiberChannel, ceph rbd, …)
- Distributed File System (NFS, CIFS …)
- SQL Database (SQL over various binary protocols: mysql, postgresql …)
- NOSQL Database (various kinds of no sql database protocol)
- Key Value store and other cache systems
- Message queue (JMS, AMQP, kafka…)

The API layer takes care of of the following concerns

- Authentication and authorization
- Input validation
- Access strategy identification (i.e. how to efficiently access storage in order to respond as quickly as possible to the current request)
- Orchestration of the requests and/or coordination with other instances.

Coordination Layer

The coordination layer ensures replicas and shards correctly participate in the request along with updating their status if needed. This is accomplished via consensus algorithms (the following sections will provide more details about this process).

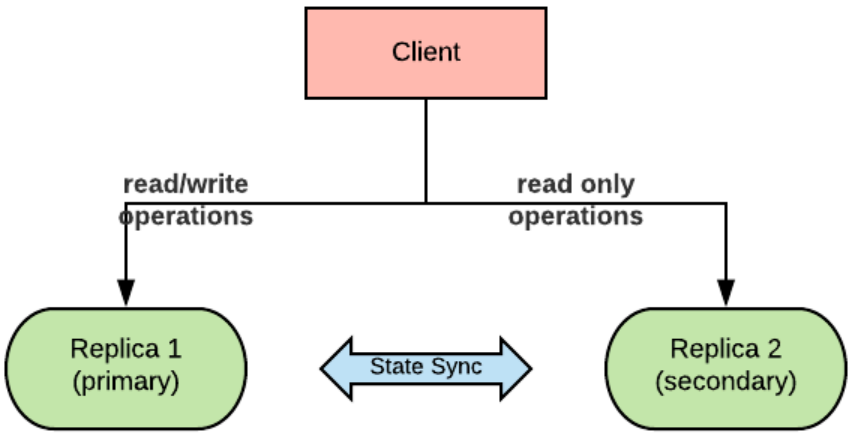
Storage Layer

The storage layer is in charge of persisting the state on durable storage. See the CNCF paper on storage for all the storage options available in this space.
The storage layer can be highly optimized depending on the API interface exposed. For example, in the case of streaming systems, essentially only one kind of write operation is allowed (append a message at the end of the queue). This very specific use case can be highly

optimized, for example, granting Kafka the ability to ingest an enormous amount of messages. On the other hand, in many cases, the access pattern can be so random that a generic storage subsystem can be used. RocksDB is one such implementation using an embeddable storage subsystem and there are several stateful workloads (SQL, noSQL, queue system, object storage, etc…) that are built on top of it.

Replicas

Replicas are a way to increase availability of a stateful workload. By having multiple replicas, the workload can continue servicing requests even when one of the replicas becomes unavailable. To do so, replicas' state must be kept in sync. Replicas can work in master/slave or multimaster mode depending on the implementation. Master replicas can execute both read and write type of requests, while normally slave replicas can only carry out read requests. In addition, replicas can also help with scaling horizontally the workload.



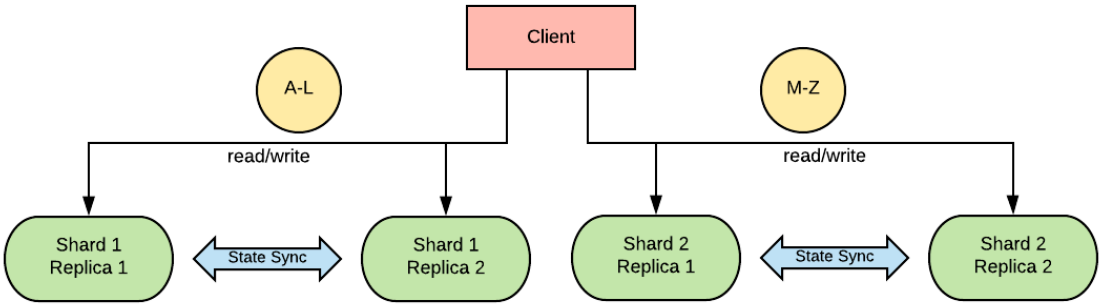
Replicas are called in different ways by different kinds of stateful workloads, but the concept remains roughly the same. The following are some such examples from popular products:

Product Name	Name used for Replicas
ElasticSearch	replica
Cassandra	keyspace
MongoDB	replica set
CockroachDB	replica

Shards

Shards are a way to increase the general throughput of the workload. Usually, the state space is broken down into two or more shards based on a hashing algorithm. The client or a proxy decides where to send requests based on the computed hash. This dramatically increases horizontal scalability, whereas historically for RDBMS, vertical scaling was often the only practical approach.

From an availability perspective, shards do not have a significant impact, although they can decrease the MTTF of the system as a whole. Each shard is an island, and the same availability considerations that apply to a non-sharded database also apply to each individual shard. Stateful workloads can have replicas of shards which sync their state to increase the availability of each individual shard.



Shards, however, while allowing for horizontal scalability, introduce the additional complication of needing to maintain consistency between them. If a transaction involves multiple shards, there needs to be a method to ensure that all of the involved shards are coordinated into participating in their portion of the transaction.

Shards also introduce the issue of deciding how to divide the data. If one has a single data-space that needs sharding, the decision is relatively simple. However, when there are multiple data-spaces in a single database that need sharding, it can be difficult to calculate the optimal sharding policy. Unbalanced or unoptimized shards can impact the availability and performance of the system.

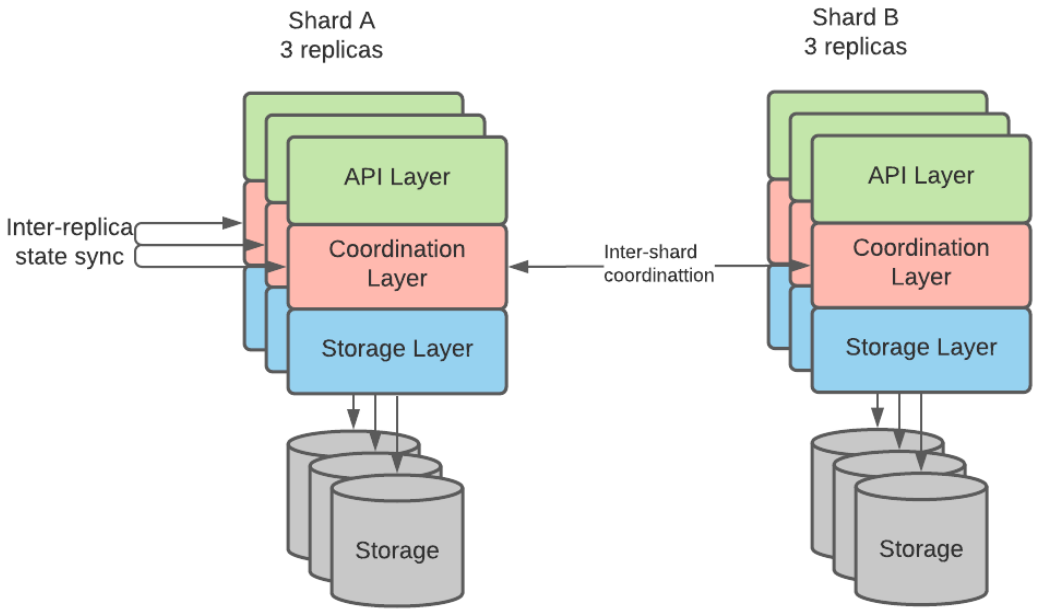
Shards are widely adopted in modern databases to allow for unbounded scalability and need to be taken into consideration especially with regard to the multi-shard consistency issue.

Shards are called in different ways by different kinds of stateful workloads, but the concept remains roughly the same. Examples include:

Product Name	Name used for Shards
ElasticSearch	index
Cassandra	partition
MongoDB	shards
CockroachDB	range

Putting it all together

The following diagram summarizes many of the concepts that have been discussed thus far and consists of a deployment of a stateful workload with two shards. Each shard has 3 replicas with independent storage volumes.



Consensus Protocols

Consensus Protocols allow for the coordination of distributed processes by agreeing on the actions that will be taken.
Two major families of consensus protocols can be identified: Shared state (between instances) and unshared state.

Shared state better suits the replicas coordination use case while unshared state is preferred by the shard coordination use case.

In shared state consensus protocol, only the strict majority of the instances need to agree on the proposed action, while in unshared state consensus protocol all of the instances need to agree or else the transaction fails.

Consensus protocols should be treated in a similar manner as encryption algorithms; only those that have been thoroughly tested and validated should be trusted.

Shared State Consensus Protocols

A component of shared state consensus protocols is a leader election process. After an agreement from a strict majority of the members of a stateful workload cluster, a leader is designated as the ultimate and undiscussed owner of the state.

As long as the strict majority of the elements of the cluster can communicate with each other, the cluster can continue to operate in a non-degraded state (without violating the CAP theorem). This results in a stateful system that is both consistent and available, while sustaining a number of failures.

In a cluster of two, if a member is lost, the remaining member does not represent the strict majority. In a cluster of three, if a member is lost, the two remaining members do represent the strict majority. As a consequence, for a stateful workload that implements a leader election protocol, there must be at least three nodes to preserve availability and consistency in the presence of one failure (HA-1).

As of today, there are two main generally accepted and formally proven consensus algorithms based on leader election:

- Paxos - Generally considered very efficient, but can be difficult to understand and is challenged by several real world corner cases.
- Raft - Generally considered easy to understand for most real life scenarios, even though it is less efficient.

Most of the new stateful software tends to be based on Raft as it is simpler to implement.

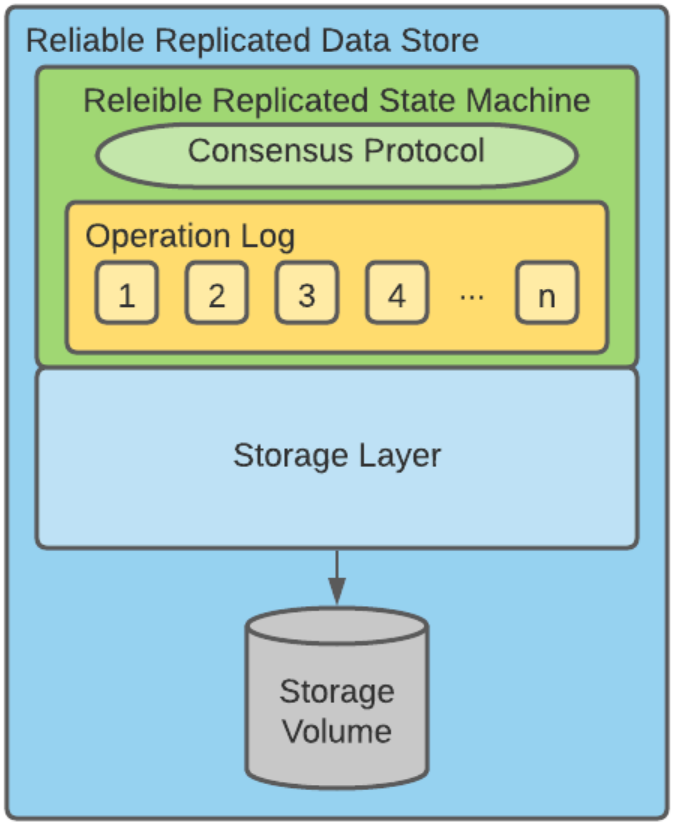
Reliable Replicated State Machines

A replicated state machine (RSM) is a system that executes the same set of operations, in the same order, on several processes. A reliable replicated state machine relies on a consensus protocol to ensure that a set of operations are agreed upon and executed in absolute order by all the instances of a stateful workload.

Notice that given the concept of log of operations in the Raft consensus protocol, with Raft it is easier to implement a Reliable Replicated State Machine.

Reliable Replicated Data Store

Reliable Replicated Data Store builds on the concept of reliably replicated state machines. The goal of the replicated state machine is to store data in datastores.
Reliably replicated data stores are a foundational building block of modern stateful workloads and govern how replicas are synchronized.



The previous diagram depicts how a Reliable Replicated Data store can be created by combining a reliable Replicated State Machine and a Storage Layer.

Unshared State Consensus Protocols

Unshared state consensus protocols can be used to coordinate processes by agreeing on some action to perform. Notice that the action can be different for each of the processes involved. For this reason a coordinator is needed to orchestrate the involved processes and keep track of

what action each process needs to perform. Unshared state consensus protocols are apt at coordinating cross-shard requests.

2PC

2PC (two-phase commit) is a specialized form of consensus protocol used for coordination between participants in a distributed atomic transaction to decide on whether to commit or abort (roll back) the transaction. 2PC is not resilient to all possible failures, and in some cases, outside (e.g. human) intervention is needed to remedy failures. Also, it is a blocking protocol. All participants block between sending in their vote (see below), and receiving the outcome of the transaction from the co-ordinator. If the co-ordinator fails permanently, participants may block indefinitely, without outside intervention. In normal, non-failure cases, the protocol consists of two phases, whence it derives its name:

1. The commit-request phase (or voting phase), in which a coordinator requests all participants to take the necessary steps for either committing or aborting the transaction and to vote, either "Yes" (on success) , or "No" (on failure)
2. The commit phase, in which case the coordinator decides whether to commit (if all participants have voted "Yes") or abort, and notifies all participants accordingly.

3PC

3PC adds an additional phase to the 2PC protocol to address the indefinite blocking issue mentioned above. But 3PC still cannot recover from network segmentation, and due to the additional phase, requires more network round-trips, resulting in higher transaction latency

Examples of consensus protocol used by stateful workloads

The following table illustrates several stateful workloads products and their choices in terms of consensus protocols.

Product	**Replica consensus protocol**	**Shard consensus protocol**
Etcd	Raft	N/A (no support for shards)
Consul	Raft	N/A (no support for shards)
Zookeeper	Atomic Broadcast (a derivative of Paxos)	N/A (no support for shards)
ElasticSearch	Paxos	N/A (No support for transactions)
Cassandra	Paxos	Supported, but details are not available.
MongoDB	Paxos	Homegrown protocol.

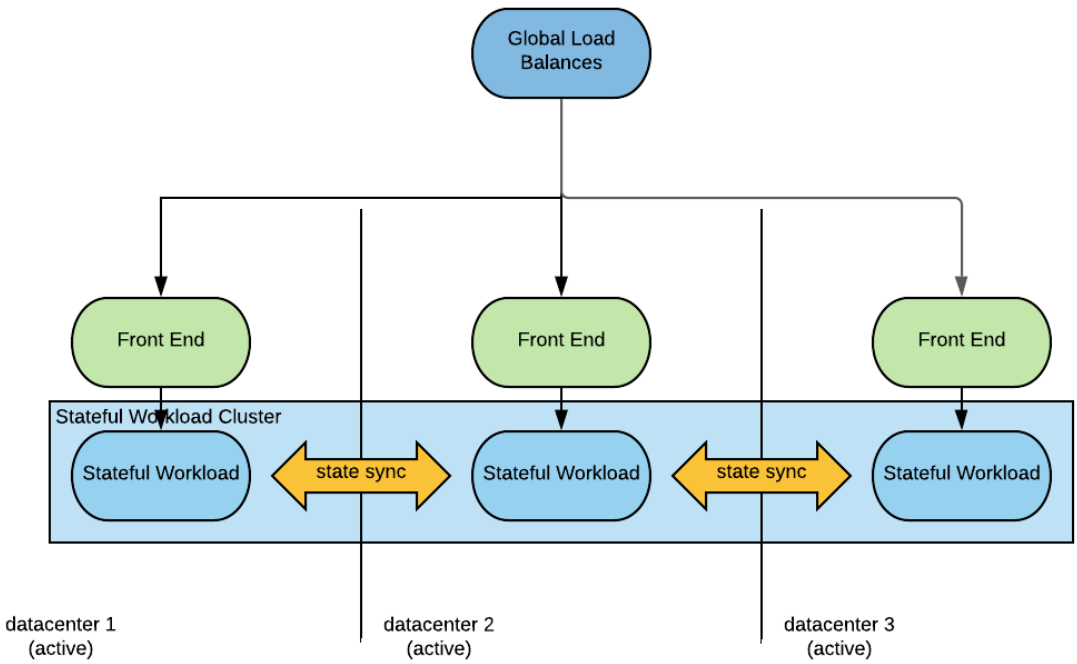
CockroachDB	Raft	2PC
YugabyteDB	Raft	2PC
TiKV	Raft	Percolator
Spanner	Raft	2PC+high-precision time service
Kafka	A custom derivative of PacificA	Custom Implementation of 2PC

Cloud Native Disaster Recovery - Reference Design

This section describes two reference implementation approaches to cloud native disaster recovery as defined at the beginning of this document. The first approach features strong consistency, while the second is an eventual consistency approach.

Strong Consistency

A strong consistency cloud native disaster recovery deployment can be built by picking a stateful workload that favors consistency in the CAP theorem.
The high-level architecture is displayed in the following diagram:



As we can see a global load balancer distributes traffic to the datacenters. The global load balancer should be able to sense the application health in each datacenter with the use of health checks. The global load balancer should also be able to implement different load balancing policies. A common load balancing policy in these scenarios is low latency, where a consumer is redirected to the closest data center, using latency as the metrics for distance.

The traffic may reach directly the stateful workload after being load balanced, but more typically it will reach some stateless front-end tier. The front-end tier will access the stateful workload in the same locality.

The stateful workload can communicate in an east-west fashion with the other instances deployed in the other region/datacenters in order to sync the state.

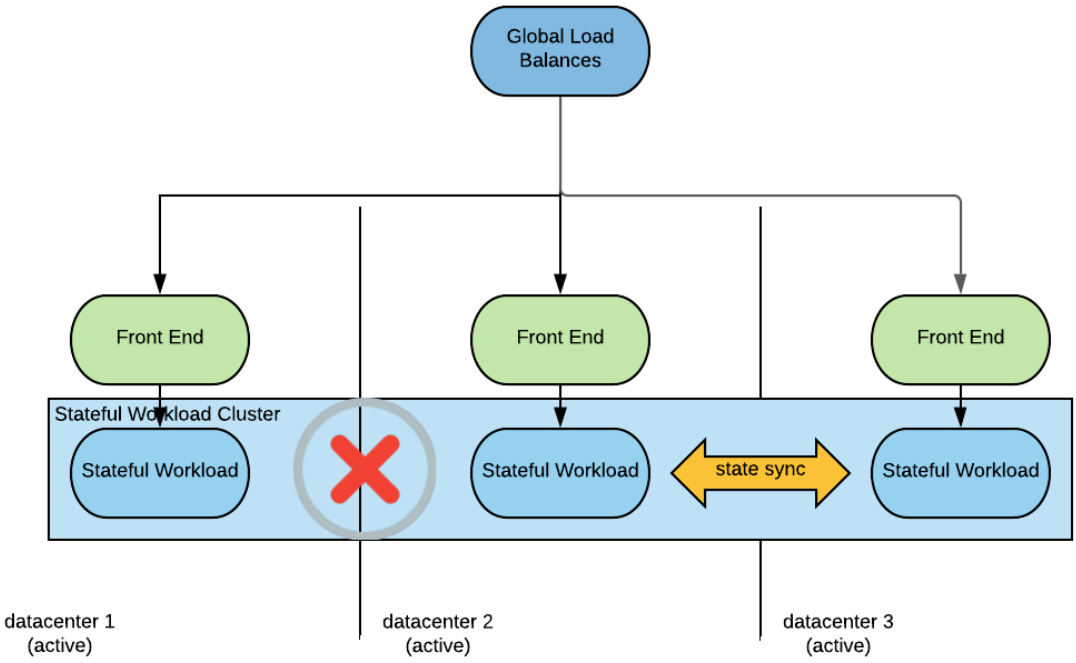
When a disaster occurs, the global load balancer will detect the unavailability of one of the data centers and redirect all traffic to the remaining active datacenters. No action needs to occur on the stateful workload as it will manage the loss of a cluster member. Likewise when normal operations are resumed the stateful workload will reorganize itself and the recovered instances will become active after catching up with any state loss they may have incurred into. Once the recovered instances become active again the global load balancer will sense that and will resume serving traffic to the recovered data center or regions. No human intervention is needed in either case.

Strongly consistent deployments guarantee an RPO of exactly zero.

Given that in order to guarantee consistency messages have to be replicated across datacenters which have possibly high-latency between them, these architectures may not be suitable for all the applications, especially not for very latency-sensitive applications.

Considerations on network partitioning

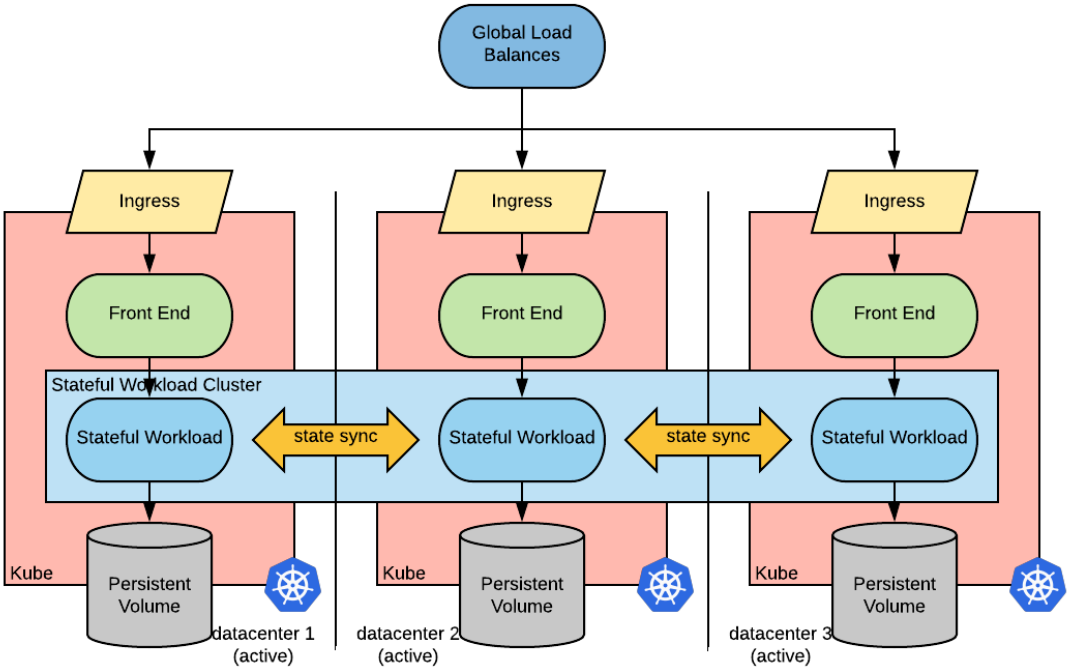
Network partitioning is a situation that requires some attention in this kind of deployments. Network partitioning is different from a disaster situation that takes down an entire datacenter as we have described previously. Here is what a network partition might look like:



In this situation connectivity between datacenter one and the other datacenters is not possible. Notice that connectivity from outside the data centers may still be possible so from the global load balancer perspective all data centers are still available.
In this situation the because the stateful workload instances in the datacenter 1 cannot reach quorum they will make themselves unavailable. If the global load balancer health check is sophisticated enough to detect that the stateful workload instances are not available, connections will be redirected to the available data centers and the system will behave as when in a disaster situation. If the health checks are not sophisticated enough, consumers connecting to datacenter one will receive an error. In either case consistency of data is guaranteed.

Kubernetes implementation considerations

A possible implementation of the described above active/active strongly consistent strategy in kubernetes is depicted below:



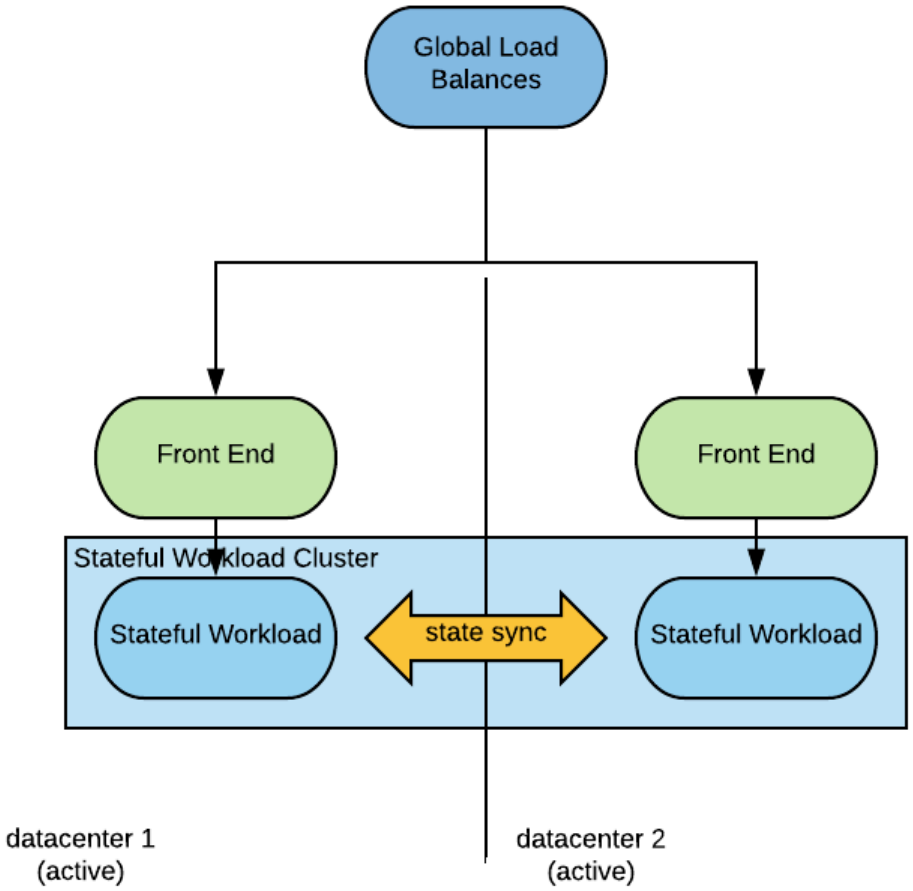
In order to implement this architecture we need the following capabilities:

1. A global load balancer with the ability to define health checks. The global load balancer should be configured based on the state of kubernetes clusters. Ideally an operator would do that.
2. The ability for the instances of the stateful workload to communicate in an east-west fashion between the clusters. This can be achieved in many ways depending on the CNI implementations. For some CNI implementations, pods are directly routable from outside the pod's network, in this case cross-cluster discoverability is needed. Other CNI implementations define an overlay network for the pods, in this case an overlay network to overlay network routability is needed. This can be implemented via a network tunnel.

Surprisingly, the capabilities needed for cloud native disaster recovery fall in the networking area rather than in the storage area as one might have expected.rea as one might have expected.

Eventual Consistency

An eventually consistent cloud native disaster recovery deployment can be built by picking a stateful workload that favors availability in the CAP theorem.
The architecture will look as follows:

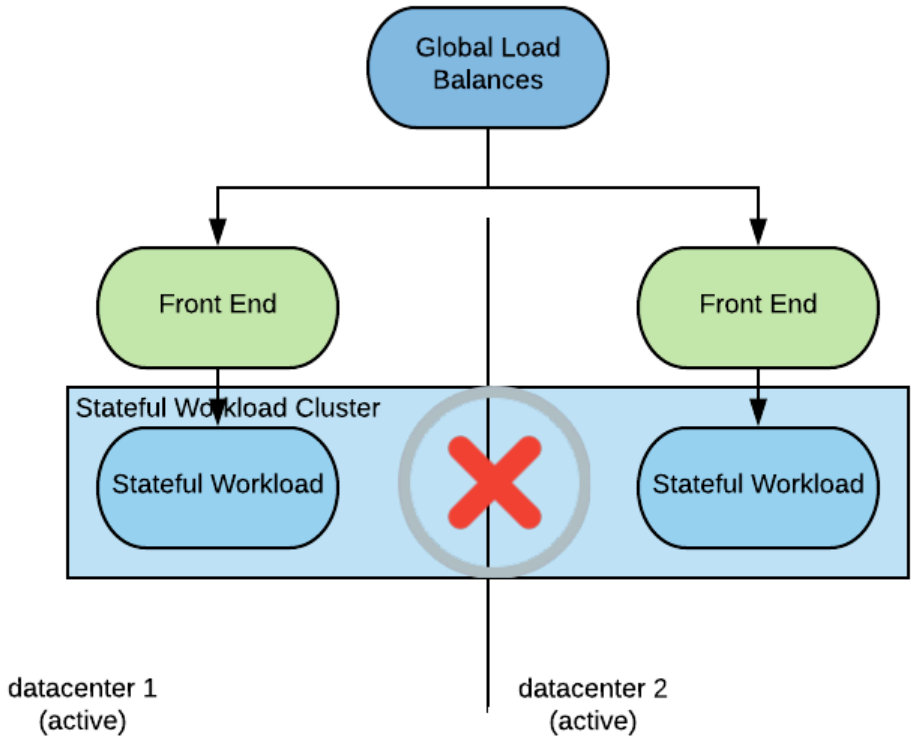


For this discussion, we define as an eventual consistent workload a workload that persists data locally first and then propagates the changes to its peers. This simplification is needed to make the discussion tractable and does not change the conclusions. Many eventually consistent workloads allow you to define the number of copies of the data that have to have been persisted before the transaction can be considered successful. As long as the number of copies is lesser than the strict majority, we still have an eventual consistent behavior, if the number of consistent copies is equal or higher than the strict majority of the instances, then we fall in the strongly consistent camp (see paragraph above).
Differently from a strongly consistent deployment, here we need only two datacenters.
When a disaster occurs, the global load balancer will detect the unavailability of one of the data centers and start direct connections to the other one. This is similar to what happens with strongly consistent deployments, resulting in a RTO close to zero. The main difference from a strongly consistent deployment is that there can be some transactions that have been persisted locally in the datacenter that is hit by the disaster and have not been synched with the other datacenter. The consequence of this is that the RPO of this architecture is not zero. Under normal circumstances the RPO will be very small, likely a multiple of the latency between the two datacenters. But if the system is under stress unsynced transactions will accumulate on one

side with no upper bound, yielding a theoretically unbounded RPO (however this is an unlikely situation).
When the disaster situation is recovered, the instances of the stateful workload running in the restored site will sync back automatically and, when ready, the global load balancer will start distributing traffic to both datacenters. No human intervention is required.

Considerations on network partitioning

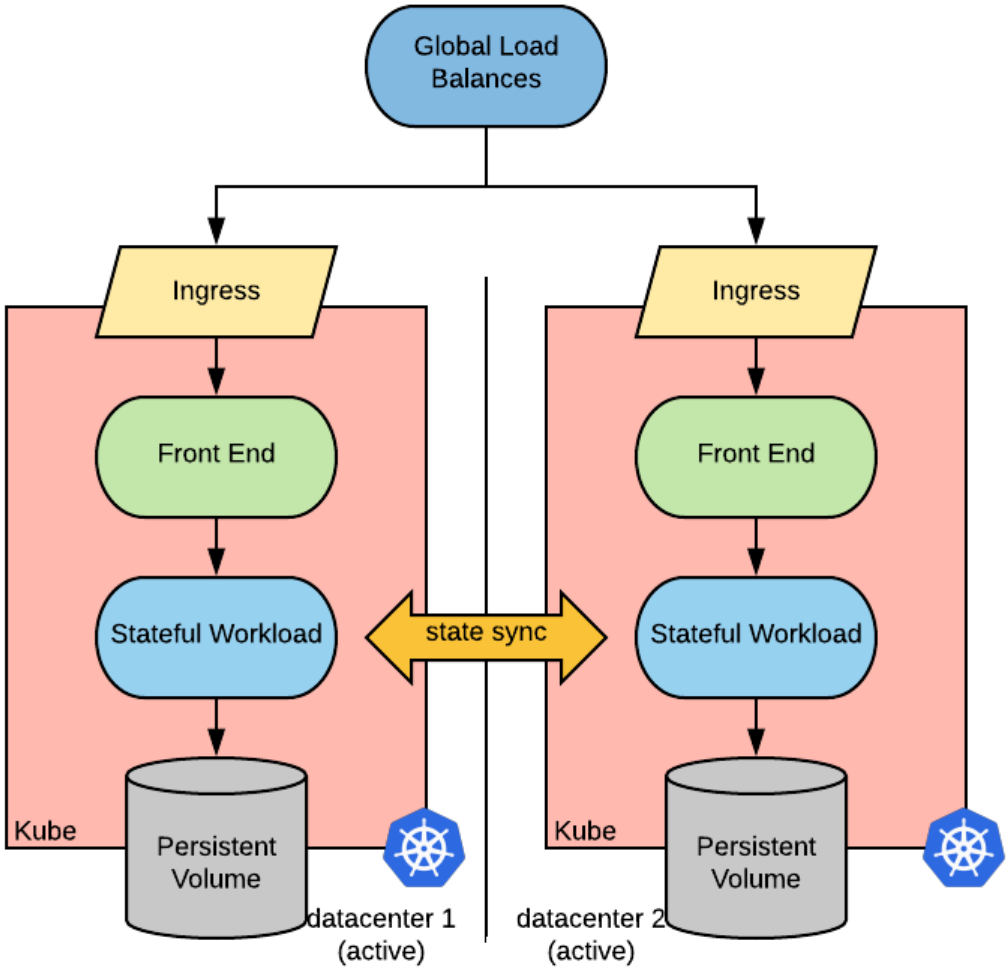
A network partition scenario for an eventual consistent deployment looks as the following diagram:



As shown in the picture, connectivity between datacenter one and two is interrupted. From a consumer and load balancer perspective though, both data centers are still available. Consumers of this service will be able to connect and operate normally, but the state of the stateful workload will diverge between the two sites.
When the partition condition is removed, the state will converge based on a state reconciliation logic. Notice that this logic does not guarantee that the final state will be correct in the application-specific business logic sense.

Kubernetes implementation considerations

A possible implementation of the described above active/active eventual consistent strategy in kubernetes is depicted below:



The same implementation-related considerations as for the strongly consistent deployment apply here, the main difference is that we need only two data centers/regions.