Achieving High Availability in a Strongly Consistent System

Achieving high availability in modern, stateful distributed systems is a critical yet complex engineering challenge. This paper examines the core strategies required to tolerate infrastructural failures and maintain service continuity. By exploring everything from single points of failure to multi-region architectures, we provide a practical framework for identifying the right trade-offs whilst building resilient, “always-up” services.

Table of Contents

Introduction

If you're designing a software service today, high availability is likely a primary objective. Whether they are consumer-facing or not, systems these days are expected to be "always up". A system's "availability'' is simply the ability for the system to continue to operate — perhaps in a slightly lesser capacity — in the face of infrastructural failures. A highly available system, then, is one that implements strategies to tolerate a limited number of common failures without experiencing catastrophic failure.

The technical challenge with high availability lies in achieving it in a distributed, *stateful* system.

How can we quantify high availability?

What types of failures should be factored in for modern architectures?

What trade-offs should we make to maximize high availability?

And how can we find the best strategy that works for us?

In this paper, we'll explore these questions as we walk through the process of building a highly available service. We will start by understanding single points of failure, based on which we will define high availability and uptime. We will then examine different strategies for implementing high availability in the context of a single data center and expand out to different zones and regions.

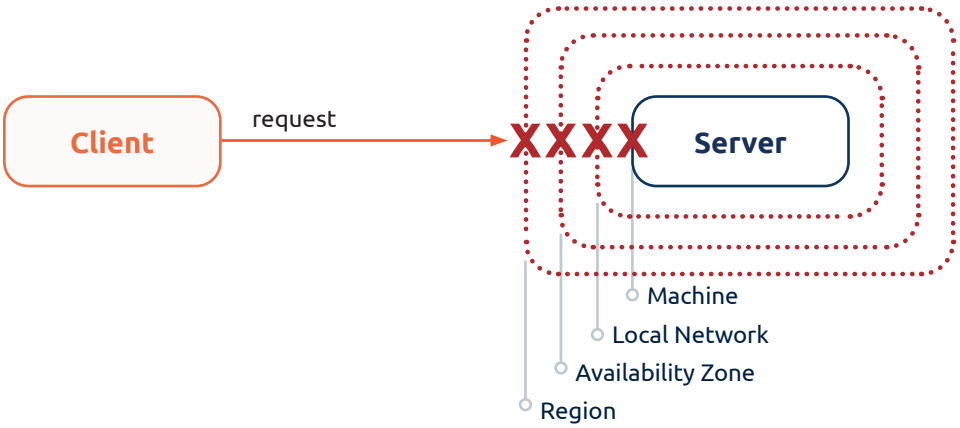
The Single Point of Failure

The single point of failure lies at the root of high-availability challenges.

Consider the classic distributed architecture of client-server. The client sends requests to the server and expects a response back.

Per the simple client-server design above, if we are deploying our entire service or a critical subsystem on a single machine, or if the client and server are connected via a single network link, our design suffers from a single point of failure and cannot be highly available. The failure of the network server will result in a loss of service to the client.

More broadly, this single point of failure can occur at different levels of the infrastructure, such as a single machine, a data center, or even an entire geography. In cloud computing terms, these failure boundaries can be as shown below:



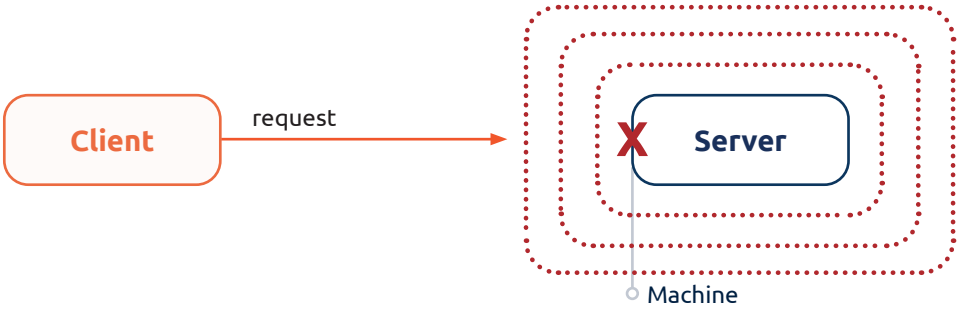
For the sake of this paper, we won't discuss the availability challenges in the network and ways to address them, but we do recognize that they do occur and must be accounted for at the system design level.

Before we look at strategies to fix the single point of failure and create a highly available service, let's first look at how to quantify availability, because you have to be able to measure it to know whether the solution is suitable for the problem in question.

Defining High Availability and Uptime

If uptime is defined as the percentage of time the infrastructure has historically been available or the percentage of time the system will likely be available in the future, high availability refers to the system's ability to support a higher uptime than that of the underlying infrastructure elements. When defined this way, we can use the infrastructure's uptime expectations to calculate the uptime probabilities achievable with different design choices.

Let us then discuss the system's uptime, assuming everything is perfect except the server machines themselves, which have an expected uptime less than 100%.



$$Let\ U_m \quad = \quad Expected\ uptime\ of\ a\ machine\ over\ a\ period\ of\ time$$

This number could have been projected from experience or possibly promised by the infrastructure provider. This is the percentage of time the machine is fully operational, such as 99.9% or 99.99%. It's frequently denoted simply by the count of 9s in the number, as in three 9s and four 9s availability. Let's scale this number to the range 0 to 1 for ease of calculation.

$$(Machine\ Uptime\ = four\ 9s) \equiv (Machine\ Uptime = 99.99\%) \equiv (U_m = 0.9999)$$

$$Let\ U_s \quad = \quad Expected\ uptime\ of\ the\ service\ over\ the\ same\ period\ of\ time$$

Under our ideal-world assumption, the system uptime depends only on the machine uptime. In the trivial case of a system with just a single machine, the system can only be up only as long as the underlying machine is up. If we ignore all the other factors for now, the uptime of a system that is deployed on a single machine would be equal to the machine's uptime.

$$U_s(U_m) \quad = \quad U_m$$

$$Machine\ uptime = 90\%\ (U_m = 0.9) \Rightarrow Service\ uptime = 90\%(U_s = 0.9)$$

When designing a highly available service, architects can start with the service's uptime requirement and the uptime expectations of each underlying infrastructure element, then work backward to identify the design parameters that will enable them to build a system that meets the given requirements.

If the service's uptime requirement is lower than that of the machine in the above example, a single server would meet it. Since, by definition, high availability means providing availability that is higher than that of the underlying infrastructure elements, we will assume for our discussion that $U_s > U_m$

Building a Highly Available Service

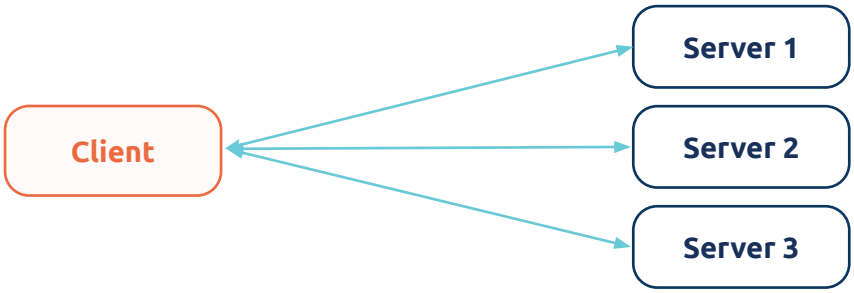
Continuing to peel back the layers of the high availability challenge, the question now becomes:

How do we actually build a highly available system? What choices should we make architecturally and tool-wise to make our systems highly available without compromising on consistency?

Stateless Distributed Service

Every high availability strategy invariably involves incorporating redundancy and failover. This is accomplished by augmenting the service with additional machines capable of executing the same tasks as the server in what's known as "redundancy."

So, instead of a single server, the service is now implemented by a *cluster of nodes*:



Setting aside the complexities of load balancing and routing requests to the servers, this is a simple design to increase the availability of stateless services — i.e., services that do not need any modifiable state to respond to a request. Examples of such services could be an encryption/encoding service or a service that serves images from a file system by name. Simply increasing redundancy increases the availability and, potentially, the throughput (if there aren't any other bottlenecks) your service can handle.

You can then calculate the cluster's uptime as the probability that the different combinations of nodes being up or down can keep the service available. For example, the service uptime for a two-node cluster would be:

$$Probability\ that\ both\ the\ nodes\ are\ up + \\ Probability\ that\ one\ of\ the\ nodes\ is\ up\ and\ the\ other\ is\ down$$

We can now quantify the service's uptime as a function of the machine's uptime U_m and the number of nodes in the service.

This function can be defined in terms of discrete probability. We can determine the service uptime by simply calculating the probability that the required number of nodes is available.

In the initial stateless service case, only one node needs to be available for the service to be available. Hence, the function to calculate the uptime would be -

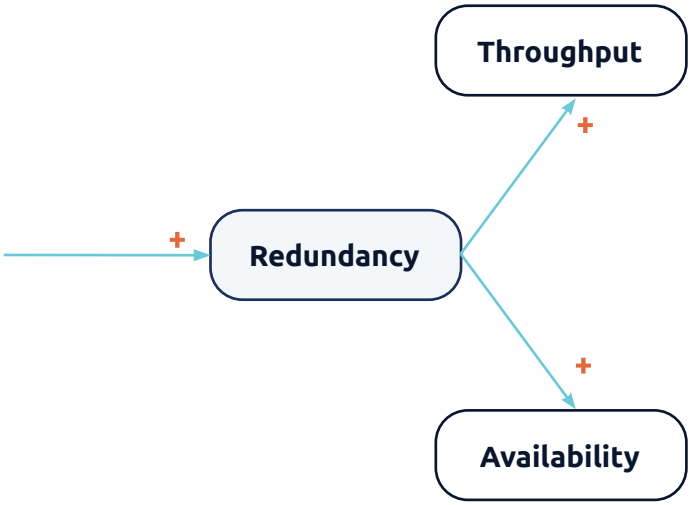
$$U_s(U_m, n) = \sum_{i=0}^{n-1} {}^{n}C_i \times (1 - U_m)^i \times U_m^{n-i}$$

The cluster's uptime will only increase as we add more nodes.

Given U_m = **0.9**

n	1	2	3	4	5	6
U_s	0.9	0.99	0.999	0.9999	0.99999	0.9999999999

We can see in this case that adding a new machine increases the service uptime by an extra '9'.

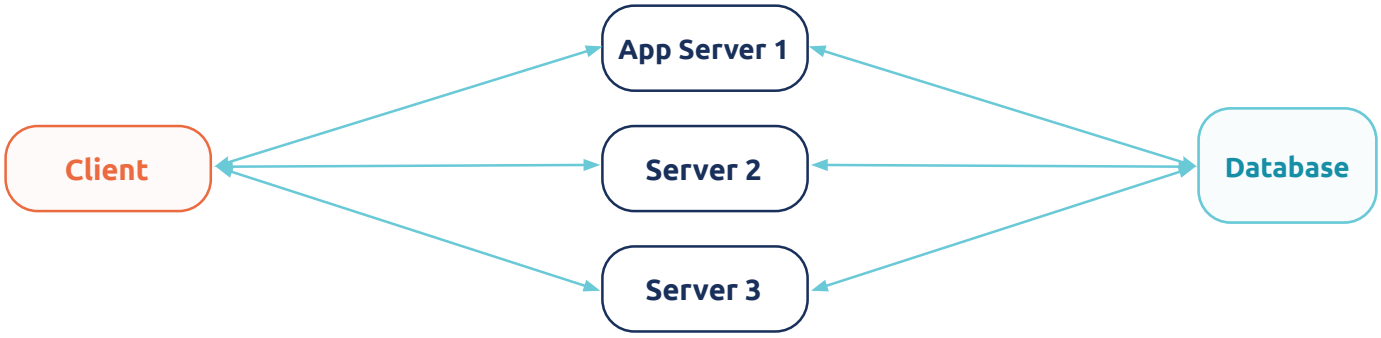


More servers, or more nodes in the cluster, not only allow more failures to be tolerated but also increase the service's throughput.

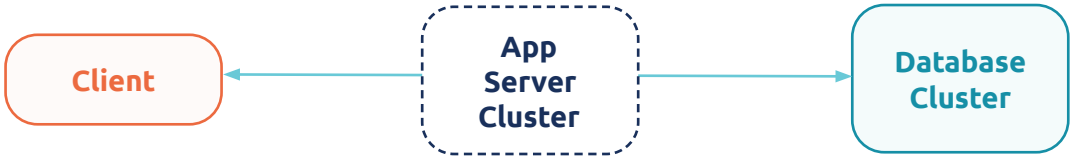
Stateful Distributed Service

Redundancy is pretty much the final answer to implementing high availability in stateless services, but the interesting challenges come with implementing stateful services.

So let's switch our context now to stateful services — services that require maintaining and controlling data necessary to respond to requests. In an e-commerce application, for example, the cart service, which allows users to add or update items in the shopping cart, would be stateful since it holds the items. Requests to the cart service use the state of the user's cart to respond appropriately. Non-trivial implementations of such services would almost always use a database to maintain the state.



Distributed databases, therefore, present themselves as an ideal candidate for discussing high-availability strategies for stateful systems. If we use a distributed database to implement our service, the problem can then be scoped to the system responsible for maintaining the data. Henceforth, we will use a system built to handle state management for services (such as a database or data processing platform) to discuss high availability.



Availability in distributed stateful systems would mean that the entire data set is available for reads and writes. For such a system to be strongly consistent, all nodes in the cluster must be able to participate in transactions. To define the uptime of a system of n nodes now,

$$U_s(n) \quad = \quad U_m^n$$

The uptime is now lower than in a stateless system because it is exposed to failures of each of the n machines instead of just one.

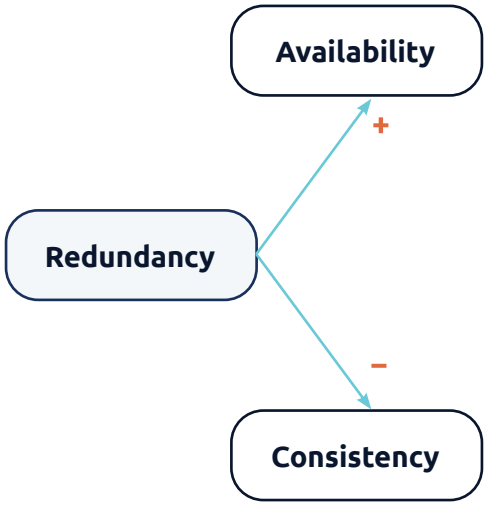
Let's calculate the uptime as we increase the node counts to see how it decays:

Given U_m = **0.9**

n	1	2	3	4	5	6
U_s	**0.9**	**0.81**	**0.729**	**0.6561**	**0.5904**	**0.3486**

Maintaining Consistency

While redundancy can be used to achieve the desired levels of availability, it must be considered carefully, as it will affect the consistency of the state the system manages. The relationship between redundancy and consistency is that creating more copies of data increases the likelihood that the state will diverge. If all the data were held as a single copy on a single server, there would be no risk of state divergence. Hence, redundancy reinforces availability and balances consistency.



If we employ strategies (independent of availability) to improve consistency, such as ensuring agreement across all copies of the data to answer a query, it becomes harder to maintain throughput, resulting in a balancing effect again.

Hence, it's important to understand the tradeoffs a stateful system makes to provide high availability. Good system software implements the optimal trade-offs for its target use cases and makes the right design decisions to achieve the best possible performance within those trade-offs.

Choosing the Consistency Level

To better understand the choices for high availability, let's first choose the consistency level our service requires, then evaluate strategies to increase availability.

Linearizability is a strong consistency model in distributed systems. This consistency level guarantees that all operations on single objects in the database appear to execute in the order consistent with their real-time execution.

Strong serializability offers the same guarantees across multiple objects. Still, the availability challenges are the same for both, since we define availability as the ability to respond to all valid requests, which means at least one copy of the entire data set is available. Lower consistency levels offer more opportunities to increase system availability but are outside the scope of this paper, as we focus on high availability only in strongly consistent systems.

Since our goal in this paper is to discuss high availability, let us treat consistency as an invariant maintained at the highest possible level for single-object operations (linearizability) and continue our journey toward maximizing high availability without compromising performance.

Many use cases benefit from linearizability or strict serializability in stateful systems, or require implementing those isolation levels in custom code. Applications such as ticketing systems need to serialize requests, since only one seat on a plane or in a theater, for example, can be assigned to a customer.

Data Persistence

Although disk and network access latencies are declining and throughput is increasing exponentially, RAM still has the lowest latency and remains the best place to store data for high-performance use cases. We will hence consider disk persistent databases out of scope for this paper. Main memory databases offer the best latency and throughput by eliminating disk-access latency. Let us use them to discuss further, since we can also avoid getting tangled up in disk-based system resiliency, which is arguably adjacent but not part of high availability.

Partitioning

Starting with the basic architecture of a stateful, scalable system, data partitioning is critical for distributing workloads across cores.

Most, if not all, databases created in the last decade use partitioning in one way or another. Transactions, however, are implemented in different ways, either using some form of concurrency control or a single-threaded execution model. Concurrency control strategies primarily aim to avoid a 2-phase commit by trading long-tail latency for better median performance.

Use a single-execution-thread model to support applications that require high consistency and performance.

Above, we represent a 3-node cluster with 4 sites per node, resulting in 12 partitions. Each of the 12 partitions contains a unique slice of the data and a dedicated thread to execute transactions on that slice. This design ensures there is no resource contention within the cluster and establishes a simple yet effective model on which to build the rest of the features.

Replication

To enable applications to scale horizontally without such deterioration in the expected uptime, we would use replication to maintain multiple consistent copies of the data.

The replication factor (k) indicates the number of *additional* copies of the data maintained in the system. Consequently, the k-factor also guarantees the number of node failures the cluster can now tolerate. The previous section could be considered a special case in which the replication factor equals 0.

Replication Granularity

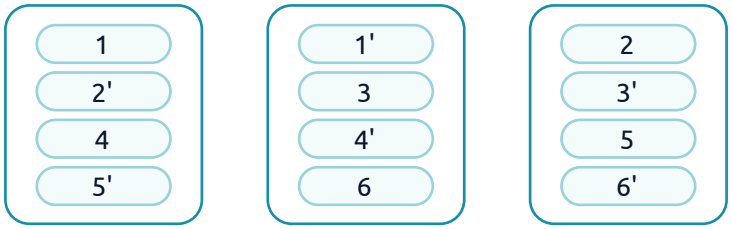
While you can establish replication at the individual node level, partitioning the data with a single thread of execution per partition allows you to distribute the data and workload more evenly across cores, provide more consistent performance, and enable horizontal scalability. It also provides greater flexibility in high-availability strategies by enabling you to simulate node-level replication.

Transaction Isolation

Achieving consistency is easier on basic database operations since they don't have to deal with the isolation in ACID. Operations are, by definition, atomic and are simply executed on the database without concern for interleaving. The execution of transactions, however, must be isolated to ensure a consistent data view.

The Volt Active Data platform, for example, implements the strongest isolation level in a performant way using a simple yet elegant design that leverages core principles that support both consistency and performance. The design uses a single-threaded execution model that runs deterministic transactions, providing strong consistency and optimal performance by avoiding the expensive commit mechanisms used in synchronous replication.

Given 3 nodes in a cluster hosting 4 sites, each with a replication factor of 1, would give you the partition distribution as below: 12 sites hosting 6 partitions with the leadership distributed evenly among the nodes. The partitions are also distributed so that a copy of each partition is available in the event of a single-node failure.



Atomic procedure invocations are submitted to the leader sites, which then replicate the invocations to the followers, ensuring all the copies of the partition are kept in sync with just a single round trip between them.

Here, the database can survive the loss of a single node. Generalizing on the k-factor to quantify the uptime,

$$U_s(U_m, n, k) = \sum_{i=0}^{k} {}^{n}C_i \times (1 - U_m)^i \times U_m^{n-i}$$

So let's calculate some uptime percentages on increasing node counts and see how the progression goes.

Given U_m = **0.9**, k = **1**

n	1	2	3	4	5	6
U_s	0.9	0.99				

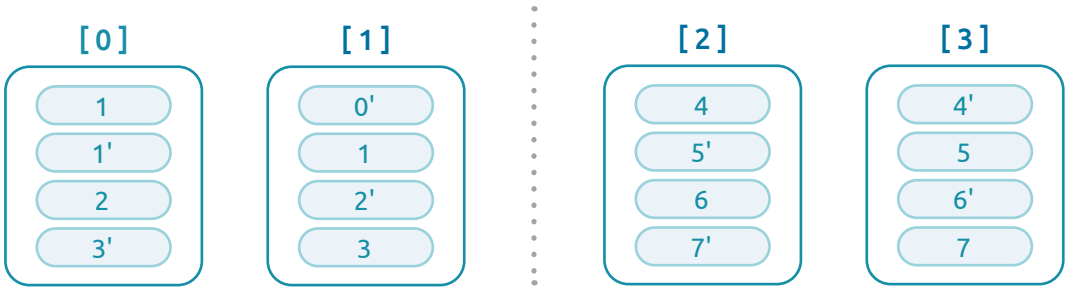
Given U_m = **0.9**, k = **1**

n	1	2	3	4	5	6
U_s	0.9					

These numbers show that you can increase the availability of a stateful system by replicating beyond what is promised per machine. The system's uptime increases, allowing you to scale by adding more machines.

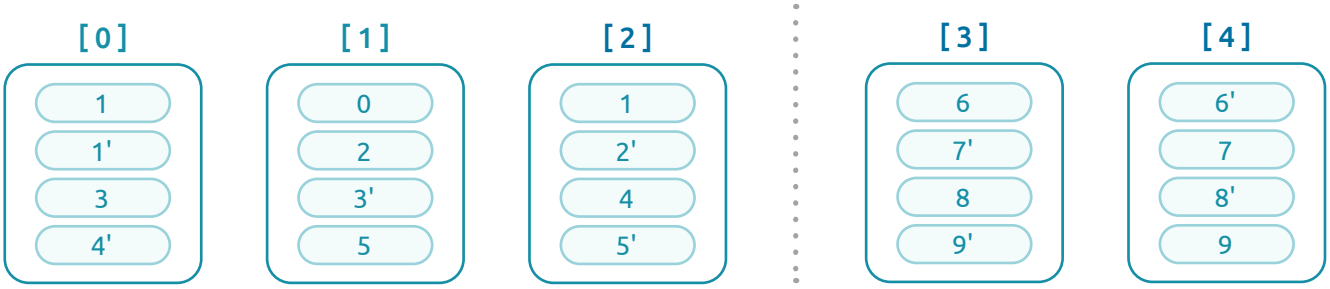
Node Pairing

Replication gives you higher uptime, but you can also optimize uptime by creating pairs of nodes and mirroring the partitions between the nodes in the pair.



Sets that can be lost { {**0**},{**1**},{**2**},{**3**},{**0,2**},{**0,3**},{**1,2**},{**1,3**} }

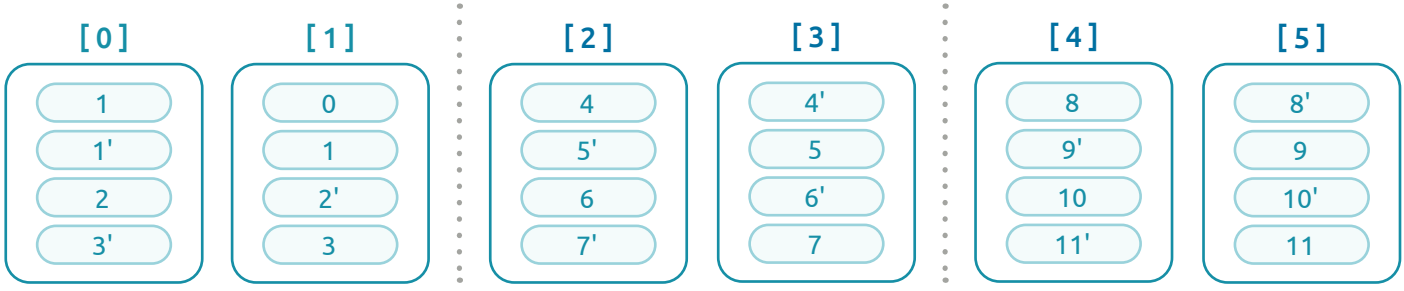
In a 4-node cluster, two pairs of nodes can be formed as above. Each node hosts exactly the same partitions as its pair, but the leadership is distributed evenly. In the case of a cluster of 4 nodes, the system can survive the loss of two nodes as long as they're not from the same pair. This gives us some extra availability.



Sets that can be lost { {**0**},{**1**},{**2**},{**3**},{**3**},{**0,3**},{**0,3**},{**1,3**},{**1,4**},{**2,3**},{**2,3**} }

In a 5-node cluster, at least one node pair can be constructed so that there is a copy of every partition on a different node, and we also have a pair of nodes with the same subset of partitions. This allows us not only to lose any single node but also to lose a node from each node pair.

And with 6 nodes, we can lose up to 3 and remain available.



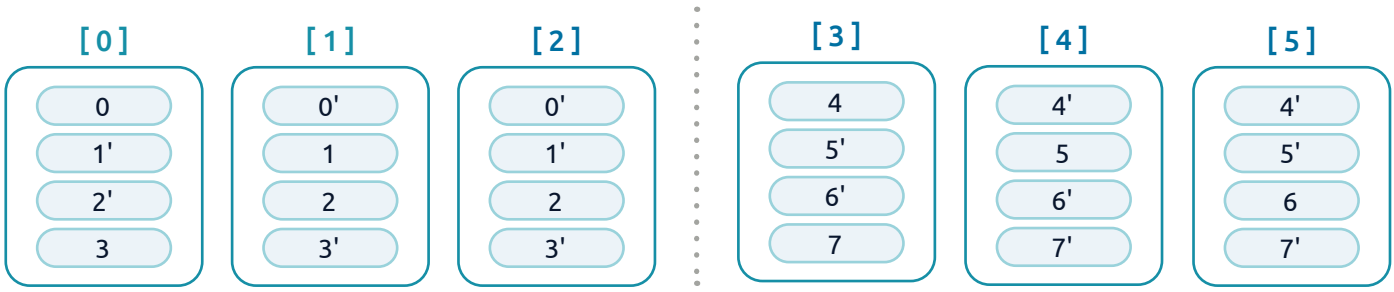
We can calculate the increased uptime due to this arrangement using the formulae below.

$$n >= 4 \; and \; n \; is \; even$$
$$U_s = U_m^n + \sum_{i=1}^{n/2} {}^{n/2}C_i * 2^i * (1 - U_m)^i U_m^{n-i}$$

$$n >= 5 \; and \; n \; is \; odd$$
$$U_s = U_m^n + n * (1 - U_m) * U_m^{n-1} + \sum_{i=2}^{n/2} {}^{n/2-1}C_i * 2^i * (1 - U_m)^i U_m^{n-i} + {}^{n/2-1}C_{i-1} \times 3 \times 2^{i-1} \times (1 - U_m)^i U^{n-i}$$

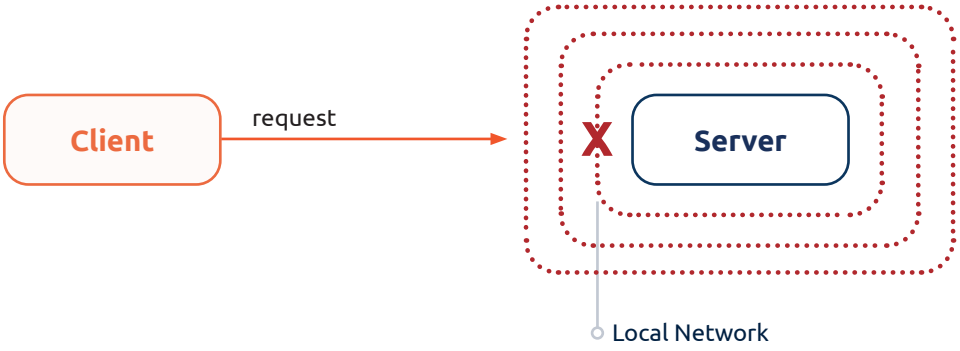
The trade-off here is that recovering a node from a crash would depend on the pair node and might introduce an imbalance in load handling across partitions on the failed node.

The calculations remain unchanged for the higher k-factor as well, since uptime now depends only on node count (and machine uptime). Even with k-factor=2, the 6-node cluster can survive the loss of 3 nodes.



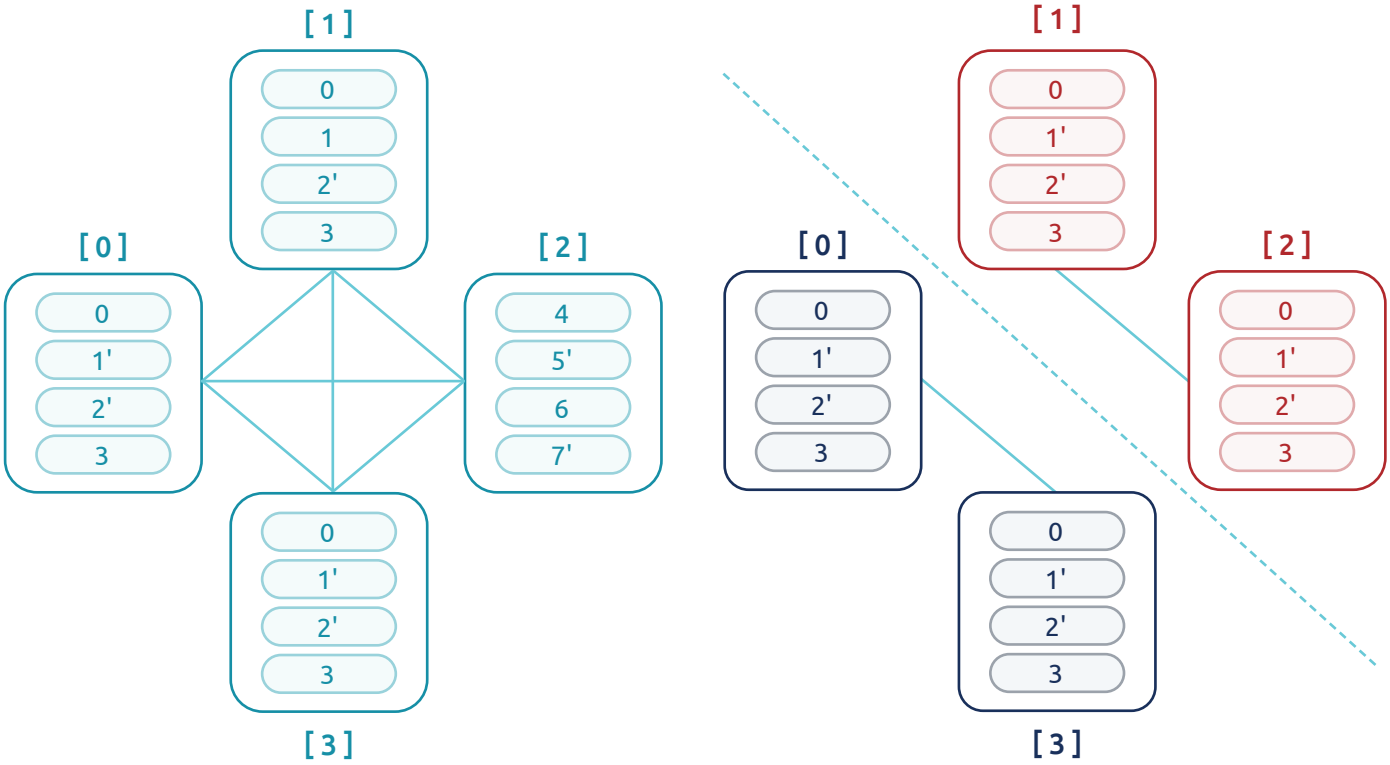
Local Network Failures

Let's now consider failures in the local network that can affect service availability.



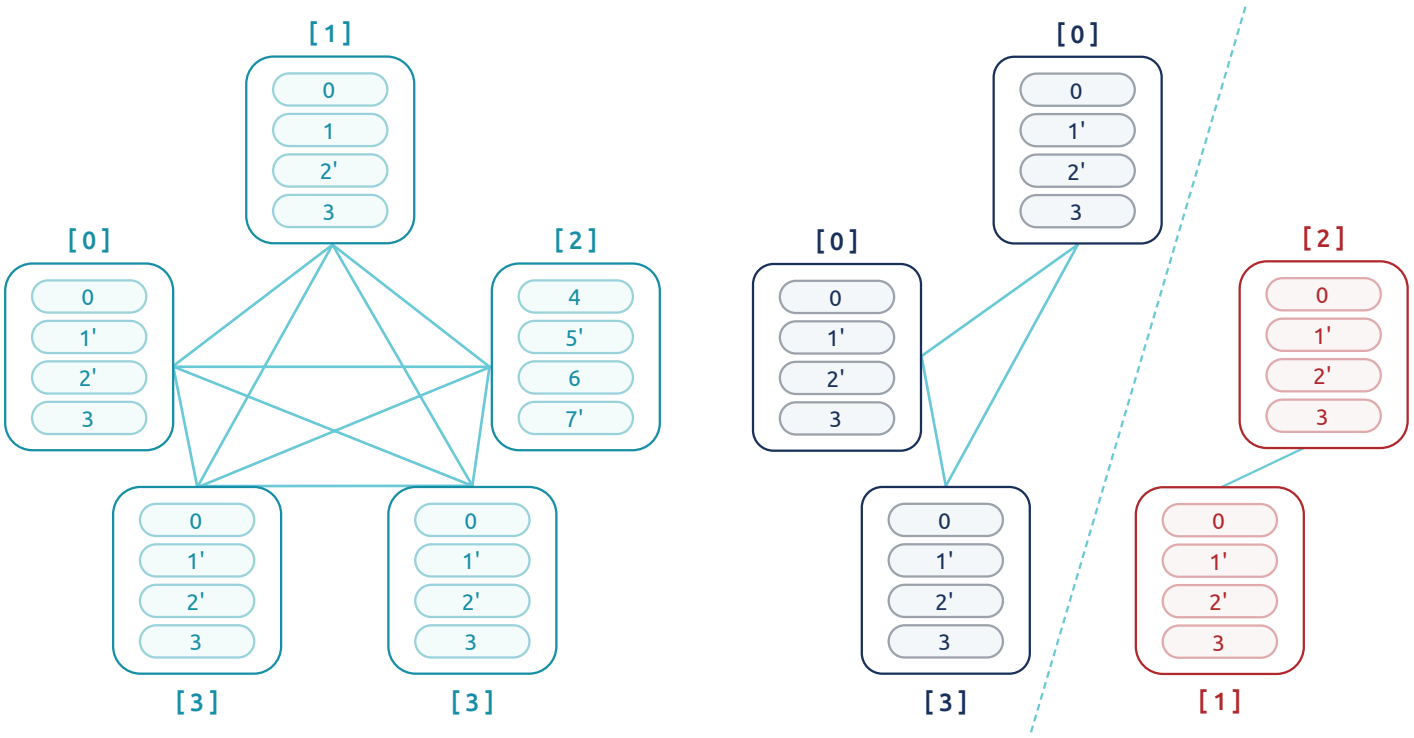
At any point in time, the network connectivity of an individual node can be lost. If the node itself recognizes that it is disconnected from the rest of the cluster, actions can be taken to ensure the node will not continue to operate in a way that violates data consistency. If such nodes simply terminate themselves, such failures will then be equivalent to the loss of a machine and hence be considered under the machine uptime expectation in the previous section.

However, certain scenarios of network partitioning among the nodes of a cluster can result in the cluster being divided into two or more independently functioning clusters. A network partition occurs when parts of the network mesh that connect all nodes are broken, dividing the cluster into distinct sets of nodes that can communicate only with one another and not with nodes outside their set. This cluster condition is also referred to as a split-brain.



In the above configuration with 4 nodes connected in a mesh network, it is possible to imagine a scenario in which the nodes are divided into disjoint clusters, such that cluster {0,3} has a full set of data, and cluster {1,2} has a full set as well. Both clusters can then assume leadership for all the partitions. This will cause the two smaller clusters to act as full databases themselves and serve clients, thereby diverging in their states.

A system like Volt Active Data, built to provide the highest level of consistency, will shut itself down under such conditions to avoid split-brain syndrome. Architects should use an odd number of nodes in their cluster for this reason. In the case of a network partition, a cluster with an odd number of nodes will prevent competing claims for leadership since the only disjoint cluster that can have a rightful claim to leadership is the one with the n/2+1 majority. The nodes in the minority clusters can then take themselves out of service to avoid divergence.

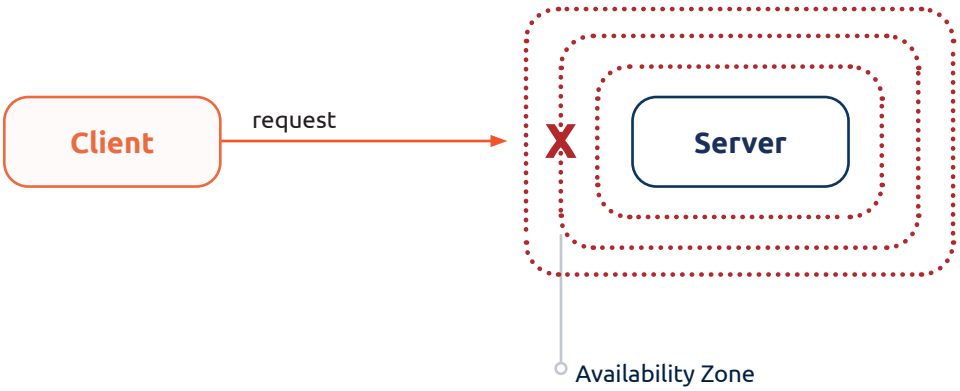


In the above scenario, only the disjoint cluster {0,1,3} can claim leadership and survive, since it's the only one that has copies of all the partitions.

Availability Zones and Regions

Loss of an Availability Zone

Let's expand our fault domain to the entire availability zone since a data center losing power or connectivity is a valid concern for organizations. We can only address this issue if the service's clients are deployed on other availability zones and are not affected by the failure under discussion.



How does this affect the availability calculations?

$Given\ U_s$ = Uptime expectation of the system given the expected uptime of the machines U_m
$Let\ U_z$ = Uptime expectation of the availability zone
$Let\ U_{sz}$ = Uptime probability of the system in the availability zone

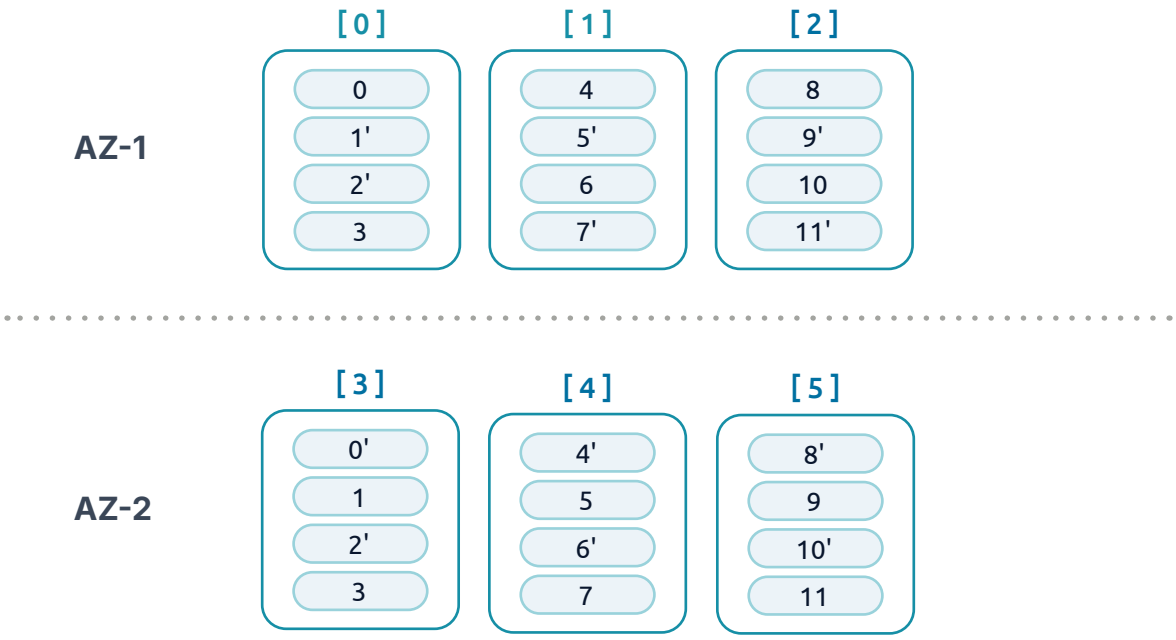
$$Then\ U_{sz} = U_s * U_z$$

For example, if we had calculated $U_s = 0.99$ and $U_z = 0.99$ then $U_{sz} = 0.9801$

So, if your service has two 9s availability and the availability zone has two 9s availability, it drops the overall availability of your service to a single 9.

Placement Groups

To address this loss of availability, you can expand the usage of redundancy to cover entire availability zones. You can assign nodes to placement groups to gain higher-level control over partition distribution. This will create a service that remains highly available even in the event of the loss of entire availability zones, while continuing to satisfy all the consistency properties that would be satisfied if the cluster were deployed in a single zone. On-prem deployments can take advantage of placement groups to provide resilience against failures larger than those of individual nodes, such as racks or rows, or even separate rooms within the data center.



The first downside of this design is that transactions will now have to contend with network latency between two availability zones. While that is a fact, public cloud availability zones often have round-trip latency in the single-digit milliseconds, which makes them close enough to use synchronous replication.

Considering the uptime of the zone as a factor now, we can update the system uptime calculations as follows:

The uptime of the system in the above arrangement is the same as that of the system with partition grouping, but with the added tolerance for the loss of an entire data center.

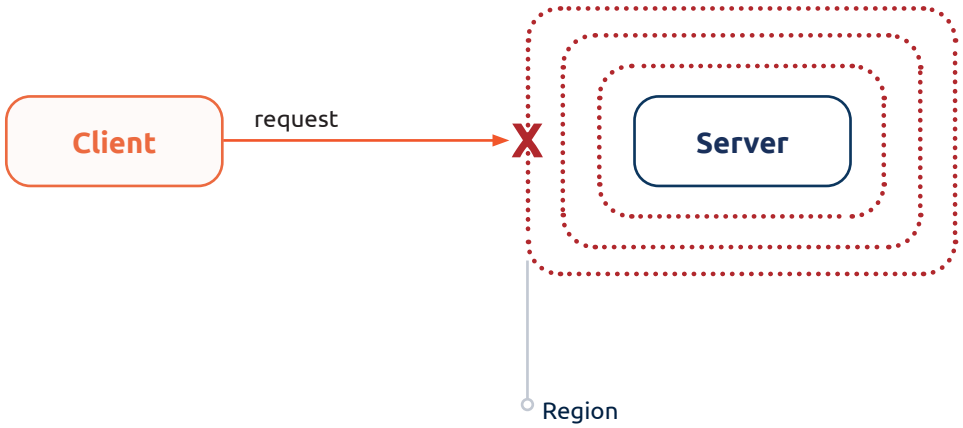
$Let\ x$	$=$	Number of availability zones the application is deployed in
$Let\ U_{sz}(U_z, U_m, n, x)$	$=$	Expected uptime of a cluster of *n* nodes divided equally among *x* availability zones
$Then\ kfactor$	$= x$	(Having at least as many copies of the data as zones is the required minimum)
$Let\ U_z$	$=$	Expected uptime for each zone
$Then\ U_{sz}$	$= \sum_{j=0}^{x-1} \left({}^{x}C_j \times U_z^{x-j}(1-U_z)^j \times \sum_{i=0}^{n/x} {}^{n/x}C_i \times (1-U_m)^i \times U_m^{n/x-i} \right)$	

Given U_Z = **0.99**, U_m = **0.9**, k = **1**

n	1	2	3	4	5	10
U_{sz}	**0.9**					

Loss of a Region

Let's now consider the case where the data centers are not in physical proximity to each other and when the network lag becomes too high for synchronous replication. Services with even more critical uptime requirements are deployed across multiple regions to withstand the loss of connectivity or functionality of all data centers in a geography.



The characteristics of lag and jitter of networks between geographic regions do not allow for synchronous replication with a fair or dependable performance. Since inter-region clusters, even active-active ones, are primarily deployed to provide resilience against the loss of entire data centers, consistency can be traded off to that end. Volt Active Data implements robust conflict detection and resolution mechanisms to detect and correct issues and prevent state divergence.

With this relaxed consistency, the system's uptime can be calculated as a function of the uptime of the system deployed across the different regions.

$Let\ U_{sr}(U_s, x)$ = Expected uptime of service deployed over x clusters each with an uptime of U_s deployed in x regions

$$U_{sr}(U_s, x) = \sum_{i=0}^{x-1} {}^{x}C_i \times (1 - U_r)^i \times U_r^{n-i} \times U_s$$

Given U_r = **0.99**, U_m = **0.9**, k = **1**

n	1	2	3	4	5	10
U_{sr}	**0.9**					

Conclusion

Availability cannot be treated independently of consistency and performance factors. It's not only important to understand the availability your business demands, but also the design of the data products to ensure the right trade-offs are made, and the design is optimal.

We have shown here how a data platform like Volt can support high availability without any meaningful tradeoffs in consistency.

Volt is unique in this respect, as it was designed from the ground up to support these types of endeavors, which are becoming increasingly common in the age of AI, 5G, and IoT.

Try Volt Today to Get Started.