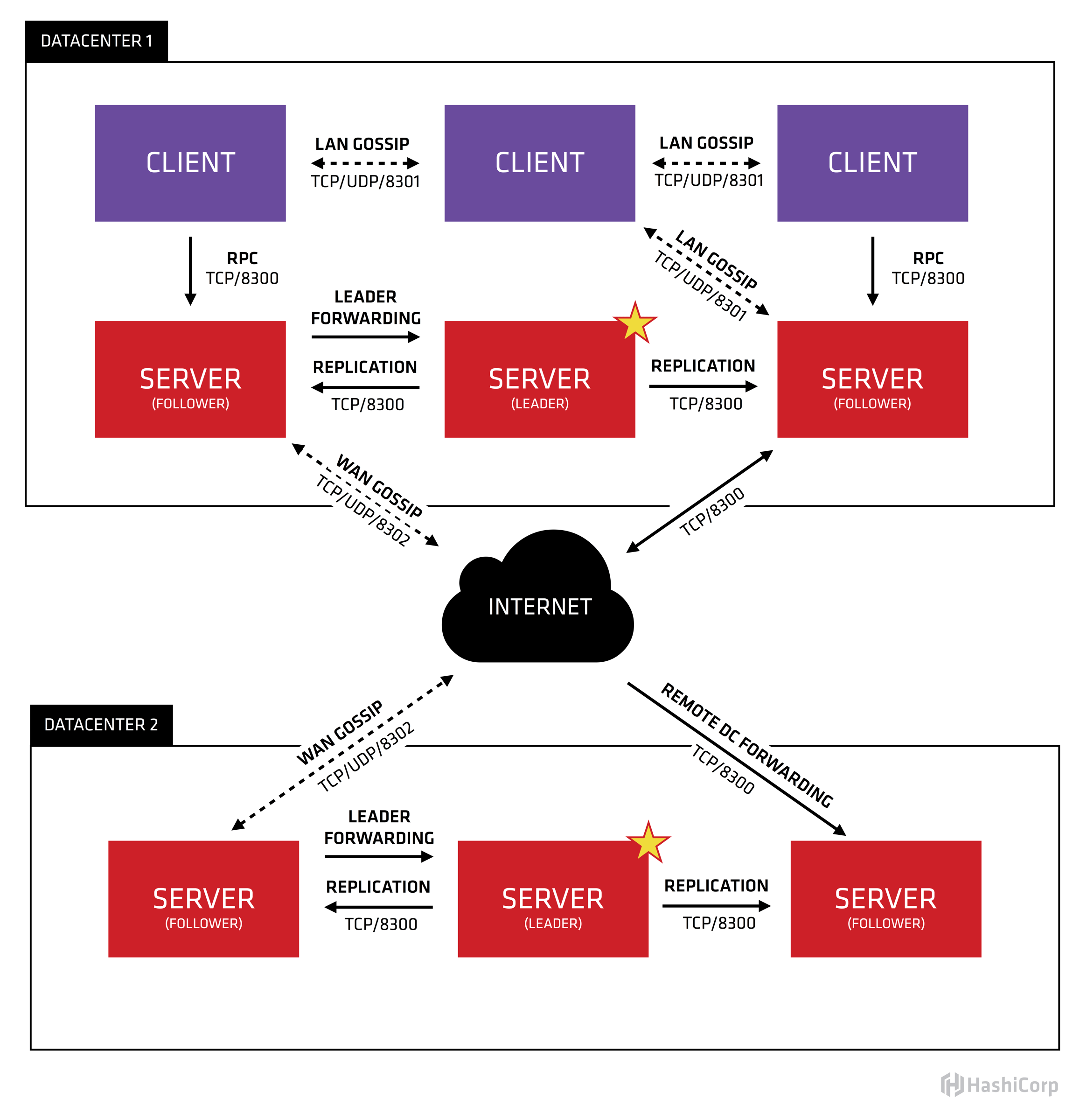
# Consul Architecture

## [»](https://www.consul.io/docs/internals/architecture.html" \l "10-000-foot-view)10,000 foot view

From a 10,000 foot altitude the architecture of Consul looks like this:

[](https://www.consul.io/assets/images/consul-arch-420ce04a.png)

Let's break down this image and describe each piece. First of all, we can see that there are two datacenters, labeled "one" and "two". Consul has first class support for [multiple datacenters](https://learn.hashicorp.com/consul/security-networking/datacenters) and expects this to be the common case.

Within each datacenter, we have a mixture of clients and servers. It is expected that there be between three to five servers. This strikes a balance between availability in the case of failure and performance, as consensus gets progressively slower as more machines are added. However, there is no limit to the number of clients, and they can easily scale into the thousands or tens of thousands.

All the agents that are in a datacenter participate in a [gossip protocol](https://www.consul.io/docs/internals/gossip.html). This means there is a gossip pool that contains all the agents for a given datacenter. This serves a few purposes: first, there is no need to configure clients with the addresses of servers; discovery is done automatically. Second, the work of detecting agent failures is not placed on the servers but is distributed. This makes failure detection much more scalable than naive heartbeating schemes. It also provides failure detection for the nodes; if the agent is not reachable, then the node may have experienced a failure. Thirdly, it is used as a messaging layer to notify when important events such as leader election take place.

The servers in each datacenter are all part of a single Raft peer set. This means that they work together to elect a single leader, a selected server which has extra duties. The leader is responsible for processing all queries and transactions. Transactions must also be replicated to all peers as part of the [consensus protocol](https://www.consul.io/docs/internals/consensus.html). Because of this requirement, when a non-leader server receives an RPC request, it forwards it to the cluster leader.

The server agents also operate as part of a WAN gossip pool. This pool is different from the LAN pool as it is optimized for the higher latency of the internet and is expected to contain only other Consul server agents. The purpose of this pool is to allow datacenters to discover each other in a low-touch manner. Bringing a new datacenter online is as easy as joining the existing WAN gossip pool. Because the servers are all operating in this pool, it also enables cross-datacenter requests. When a server receives a request for a different datacenter, it forwards it to a random server in the correct datacenter. That server may then forward to the local leader.

This results in a very low coupling between datacenters, but because of failure detection, connection caching and multiplexing, cross-datacenter requests are relatively fast and reliable.

In general, data is not replicated between different Consul datacenters. When a request is made for a resource in another datacenter, the local Consul servers forward an RPC request to the remote Consul servers for that resource and return the results. If the remote datacenter is not available, then those resources will also not be available, but that won't otherwise affect the local datacenter. There are some special situations where a limited subset of data can be replicated, such as with Consul's built-in [ACL replication](https://learn.hashicorp.com/consul/day-2-operations/acl-replication) capability, or external tools like [consul-replicate](https://github.com/hashicorp/consul-replicate).

In some places, client agents may cache data from the servers to make it available locally for performance and reliability. Examples include Connect certificates and intentions which allow the client agent to make local decisions about inbound connection requests without a round trip to the servers. Some API endpoints also support optional result caching. This helps reliability because the local agent can continue to respond to some queries like service-discovery or Connect authorization from cache even if the connection to the servers is disrupted or the servers are temporarily unavailable.

# Consul Reference Architecture

As you migrate applications to dynamically provisioned infrastructure, you will encounter challenges scaling services and managing the communications between them. Consul helps the components of dynamic applications communicate with each other by providing service discovery. It monitors the health of each node and application so that it only exposes healthy instances as discoverable. Consul's distributed Key-Value store allows you to make runtime-configuration updates across global infrastructure.

This document recommends best practices and provides a reference architecture, including system requirements, datacenter design, networking, and performance optimizations for Consul production deployments.

## [»](https://learn.hashicorp.com/consul/datacenter-deploy/reference-architecture?utm_source=consul.io&utm_medium=docs#infrastructure-requirements)Infrastructure Requirements

### [»](https://learn.hashicorp.com/consul/datacenter-deploy/reference-architecture?utm_source=consul.io&utm_medium=docs#consul-servers)Consul Servers

Consul server agents maintain the cluster state, respond to RPC queries (read operations), and process all write operations. Because Consul server agents do most of the heavy lifting, their host sizing is critical for the overall performance, efficiency, and health of the Consul cluster.

The following table provides high-level server host guidelines. Of particular note is the strong recommendation to avoid non-fixed performance CPUs, or "Burstable CPU".

| **Type** | **CPU** | **Memory** | **Disk** | **Typical Cloud Instance Types** |
| --- | --- | --- | --- | --- |
| Small | 2 core | 8-16 GB RAM | 50GB | **AWS**: m5.large, m5.xlarge |
|  |  |  |  | **Azure**: Standard\_A4\_v2, Standard\_A8\_v2 |
|  |  |  |  | **GCE**: n1-standard-8, n1-standard-16 |
| Large | 4-8 core | 32-64 GB RAM | 100GB | **AWS**: m5.2xlarge, m5.4xlarge |
|  |  |  |  | **Azure**: Standard\_D4\_v3, Standard\_D5\_v3 |
|  |  |  |  | **GCE**: n1-standard-32, n1-standard-64 |

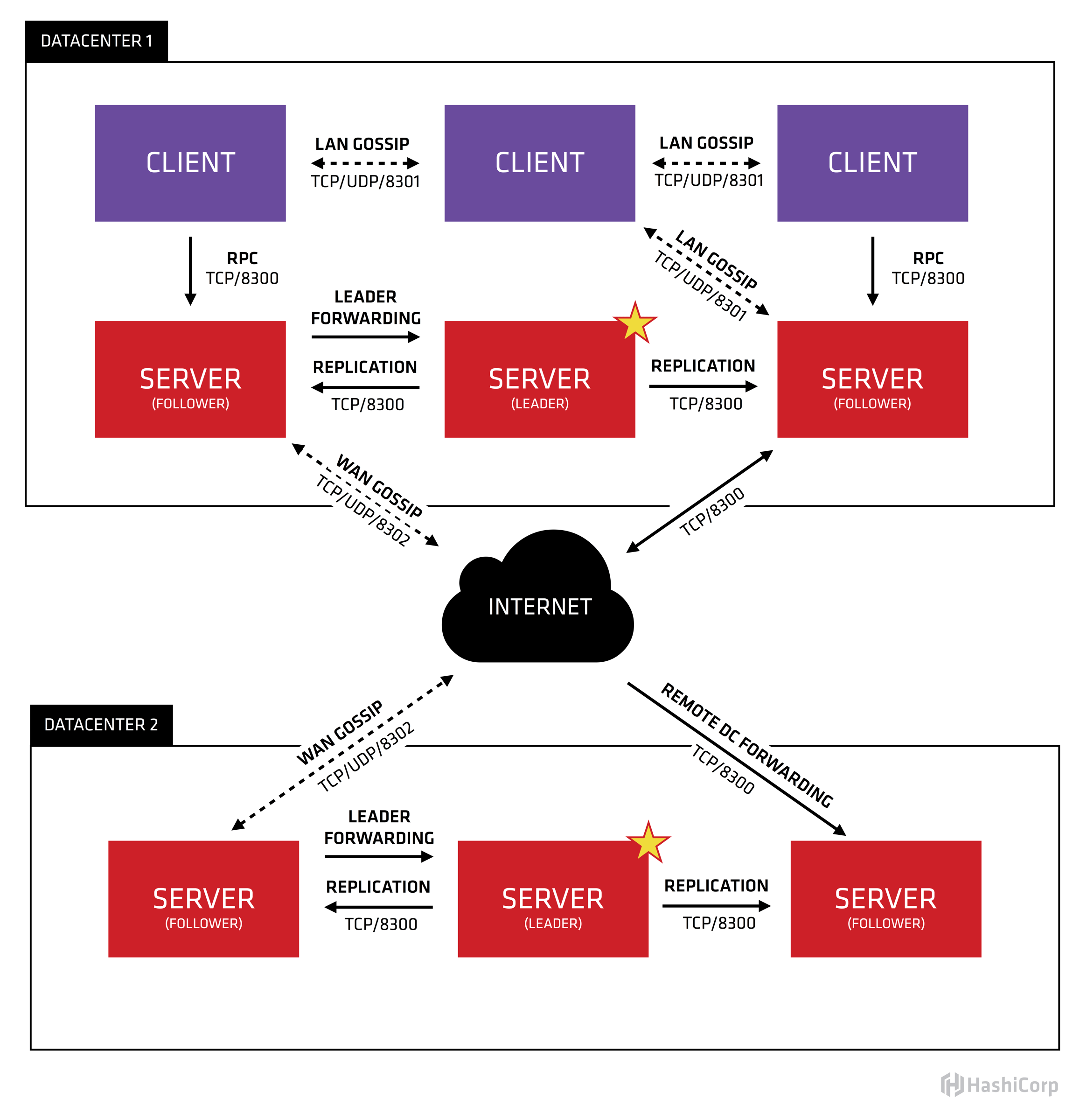
#### [»](https://learn.hashicorp.com/consul/datacenter-deploy/reference-architecture?utm_source=consul.io&utm_medium=docs#hardware-sizing-considerations)Hardware Sizing Considerations

* The small size would be appropriate for most initial production deployments, or for development/testing environments.
* The large size is for production environments where there is a consistently high workload.

**NOTE** For large workloads, ensure that the disks support a high number of IOPS to keep up with the rapid Raft log update rate.

For more information on server requirements, review the [server performance](https://www.consul.io/docs/install/performance.html) documentation.

## [»](https://learn.hashicorp.com/consul/datacenter-deploy/reference-architecture?utm_source=consul.io&utm_medium=docs#infrastructure-diagram)Infrastructure Diagram



## [»](https://learn.hashicorp.com/consul/datacenter-deploy/reference-architecture?utm_source=consul.io&utm_medium=docs#datacenter-design)Datacenter Design

You may deploy a Consul cluster (typically three or five servers plus client agents) in a single physical datacenter or across multiple datacenters. For a large cluster with high runtime reads and writes, deploying servers in the same physical location improves performance. In cloud environments, you may deploy a single datacenter across multiple availability zones, i.e., each server in a separate availability zone on a single host. Consul also supports multi-datacenter deployments via separate clusters joined by WAN links. In some cases, you may also deploy two or more Consul clusters in the same LAN environment.

### [»](https://learn.hashicorp.com/consul/datacenter-deploy/reference-architecture?utm_source=consul.io&utm_medium=docs#single-datacenter)Single Datacenter

We recommend a single Consul cluster for applications deployed in the same datacenter. Consul supports traditional three-tier applications as well as microservices.

Typically, you will need between three or five servers to balance between availability and performance. These servers together run the Raft-driven consistent state store for updating catalog, session, prepared query, ACL, and KV state.

The recommended maximum cluster size for a single datacenter is 5,000 nodes. For a write-heavy and/or a read-heavy cluster, you may need to reduce the maximum number of nodes further, depending on the number and the size of KV pairs and the number of watches. As you add more client machines it takes more time for gossip to converge. Similarly, when a new server joins an existing multi-thousand node cluster with a large KV store it may take more time to replicate the store to the new server's log, and the update rate may increase.

**TIP** For write-heavy clusters, consider scaling vertically with larger machine instances and lower latency storage.

[Service tags](https://www.consul.io/docs/agent/services.html#service-definition) help you make necessary queries against your cluster. They can help you distinguish between different services, or different versions of the same service. Without them, node searches based on a a specific service are impossible.

In cases where agents can't all contact each other due to network segmentation, you can use Consul's [network segments](https://learn.hashicorp.com/consul/day-2-operations/network-segments) (Consul Enterprise only) to create multiple tenants that share Raft servers in the same cluster. Each tenant has its own gossip pool and doesn't communicate with the agents outside this pool. All the tenants, however, do share the KV store. If you don't have access to Consul network segments you can create discrete [Consul datacenters](https://learn.hashicorp.com/consul/security-networking/datacenters) to isolate agents from each other.

### [»](https://learn.hashicorp.com/consul/datacenter-deploy/reference-architecture?utm_source=consul.io&utm_medium=docs#multiple-datacenters)Multiple Datacenters

You can join Consul clusters running the same service in different datacenters by WAN links. The clusters operate independently and only communicate over the WAN on port 8302. Unless explicitly configured via CLI or API, Consul servers will only return results from their local datacenter. Consul does not replicate data between multiple datacenters, but you can use the [consul-replicate](https://github.com/hashicorp/consul-replicate) tool to replicate the KV data periodically.

It is good practice to enable TLS server name checking in order to avoid accidental cross-joining of agents.

The [network areas](https://www.consul.io/api/operator/area.html) feature in Consul Enterprise provides advanced federation. For example, imagine that datacenter1 (dc1) hosts services like LDAP (or an ACL database) and shares them with datacenter2 (dc2) and datacenter3 (dc3). However, due to compliance issues, servers in dc2 must not connect with servers in dc3. Basic WAN federation can't isolate dc2 from dc3; it requires that all the servers in dc1, dc2 and dc3 are connected in a full mesh and opens both gossip (8302 tcp/udp) and RPC (8300) ports for communication.

Network areas allows peering between datacenters to make shared services discoverable over WAN. With network areas, servers in dc1 can communicate with those in dc2 and dc3, without a connection between dc2 and dc3. This meets the compliance requirement of the organization in our example use case. Servers that are part of the network area communicate over RPC only. This removes the overhead of sharing and maintaining the symmetric key used by the gossip protocol across datacenters. It also reduces the attack surface at the gossip ports since they no longer need to be opened in security gateways or firewalls.

Consul's [prepared queries](https://www.consul.io/api/query.html) allow clients to failover to another datacenter for service discovery. For example, if a service payment in the local datacenter dc1 goes down, a prepared query lets users define a geographic fallback order to the nearest datacenter to check for healthy instances of the same service.

**NOTE** Consul clusters must be WAN linked for a prepared query to work across datacenters.

Prepared queries, by default, resolve the query in the local datacenter first. They don't support querying KV store features, and do work with ACLs. Prepared query config/templates are maintained consistently in Raft and are executed on the servers.

## [»](https://learn.hashicorp.com/consul/datacenter-deploy/reference-architecture?utm_source=consul.io&utm_medium=docs#network-connectivity)Network Connectivity

LAN gossip occurs between all agents in a single datacenter with each agent sending a periodic probe to random agents from its member list. Both client and server agents participate in the gossip. The initial probe is sent over UDP every second. If a node fails to acknowledge within 200ms, the agent pings over TCP. If the TCP probe fails (10 second timeout), it asks a configurable number of random nodes to probe the same node (also known as an indirect probe). If there is no response from the peers regarding the status of the node, that agent is marked as down.

The agent's status directly affects the service discovery results. If an agent is down, the services it is monitoring will also be marked as down.

In addition, the agent also periodically performs a full state sync over TCP which gossips each agent's understanding of the member list around it (node names, IP addresses, and health status). These operations are expensive relative to the standard gossip protocol mentioned above and are performed at a rate determined by cluster size to keep overhead low. It's typically between 30 seconds and 5 minutes. For more details, refer to [Serf Gossip docs](https://www.serf.io/docs/internals/gossip.html).

In a larger network that spans L3 segments, traffic typically traverses through a firewall and/or a router. You must update your ACL or firewall rules to allow the following ports:

| **Name** | **Port** | **Flag** | **Description** |
| --- | --- | --- | --- |
| Server RPC | 8300 |  | Used by servers to handle incoming requests from other agents. TCP only. |
| Serf LAN | 8301 |  | Used to handle gossip in the LAN. Required by all agents. TCP and UDP. |
| Serf WAN | 8302 | -1 to disable (available in Consul 1.0.7) | Used by servers to gossip over the LAN and WAN to other servers. TCP and UDP. |
| HTTP API | 8500 | -1 to disable | Used by clients to talk to the HTTP API. TCP only. |
| DNS Interface | 8600 | -1 to disable |  |

As mentioned in the [datacenter design section](https://learn.hashicorp.com/consul/datacenter-deploy/reference-architecture?utm_source=consul.io&utm_medium=docs#datacenter-design), network areas and network segments can be used to prevent opening up firewall ports between different subnets.

By default agents will only listen for HTTP and DNS traffic on the local interface.

For more information about the ports that Consul uses, see the ports section of the [agent configuration documentation](https://www.consul.io/docs/agent/options.html#ports).

## [»](https://learn.hashicorp.com/consul/datacenter-deploy/reference-architecture?utm_source=consul.io&utm_medium=docs#summary)Summary

In this guide we've discussed considerations for deploying Consul, including hardware sizing, datacenter design, and network connectivity. Next, review the Deployment Guide to learn the steps required to install and configure a single HashiCorp Consul cluster.

# Consensus Protocol

Consul uses a [consensus protocol](https://en.wikipedia.org/wiki/Consensus_(computer_science)) to provide [Consistency (as defined by CAP)](https://en.wikipedia.org/wiki/CAP_theorem). The consensus protocol is based on ["Raft: In search of an Understandable Consensus Algorithm"](https://raft.github.io/raft.pdf). For a visual explanation of Raft, see [The Secret Lives of Data](http://thesecretlivesofdata.com/raft).

## [»](https://www.consul.io/docs/internals/consensus.html" \l "raft-protocol-overview)Raft Protocol Overview

Raft is a consensus algorithm that is based on [Paxos](https://en.wikipedia.org/wiki/Paxos_%28computer_science%29). Compared to Paxos, Raft is designed to have fewer states and a simpler, more understandable algorithm.

There are a few key terms to know when discussing Raft:

* Log - The primary unit of work in a Raft system is a log entry. The problem of consistency can be decomposed into a replicated log. A log is an ordered sequence of entries. Entries includes any cluster change: adding nodes, adding services, new key-value pairs, etc. We consider the log consistent if all members agree on the entries and their order.
* FSM - [Finite State Machine](https://en.wikipedia.org/wiki/Finite-state_machine). An FSM is a collection of finite states with transitions between them. As new logs are applied, the FSM is allowed to transition between states. Application of the same sequence of logs must result in the same state, meaning behavior must be deterministic.
* Peer set - The peer set is the set of all members participating in log replication. For Consul's purposes, all server nodes are in the peer set of the local datacenter.
* Quorum - A quorum is a majority of members from a peer set: for a set of size n, quorum requires at least (n/2)+1 members. For example, if there are 5 members in the peer set, we would need 3 nodes to form a quorum. If a quorum of nodes is unavailable for any reason, the cluster becomes unavailable and no new logs can be committed.
* Committed Entry - An entry is considered committed when it is durably stored on a quorum of nodes. Once an entry is committed it can be applied.
* Leader - At any given time, the peer set elects a single node to be the leader. The leader is responsible for ingesting new log entries, replicating to followers, and managing when an entry is considered committed.

Raft is a complex protocol and will not be covered here in detail (for those who desire a more comprehensive treatment, the full specification is available in this [paper](https://raft.github.io/raft.pdf)). We will, however, attempt to provide a high level description which may be useful for building a mental model.

Raft nodes are always in one of three states: follower, candidate, or leader. All nodes initially start out as a follower. In this state, nodes can accept log entries from a leader and cast votes. If no entries are received for some time, nodes self-promote to the candidate state. In the candidate state, nodes request votes from their peers. If a candidate receives a quorum of votes, then it is promoted to a leader. The leader must accept new log entries and replicate to all the other followers. In addition, if stale reads are not acceptable, all queries must also be performed on the leader.

Once a cluster has a leader, it is able to accept new log entries. A client can request that a leader append a new log entry (from Raft's perspective, a log entry is an opaque binary blob). The leader then writes the entry to durable storage and attempts to replicate to a quorum of followers. Once the log entry is considered committed, it can be applied to a finite state machine. The finite state machine is application specific; in Consul's case, we use [MemDB](https://github.com/hashicorp/go-memdb) to maintain cluster state. Consul's writes block until it is both committed and applied. This achieves read after write semantics when used with the [consistent](https://www.consul.io/api/features/consistency.html#consistent) mode for queries.

Obviously, it would be undesirable to allow a replicated log to grow in an unbounded fashion. Raft provides a mechanism by which the current state is snapshotted and the log is compacted. Because of the FSM abstraction, restoring the state of the FSM must result in the same state as a replay of old logs. This allows Raft to capture the FSM state at a point in time and then remove all the logs that were used to reach that state. This is performed automatically without user intervention and prevents unbounded disk usage while also minimizing time spent replaying logs. One of the advantages of using MemDB is that it allows Consul to continue accepting new transactions even while old state is being snapshotted, preventing any availability issues.

Consensus is fault-tolerant up to the point where quorum is available. If a quorum of nodes is unavailable, it is impossible to process log entries or reason about peer membership. For example, suppose there are only 2 peers: A and B. The quorum size is also 2, meaning both nodes must agree to commit a log entry. If either A or B fails, it is now impossible to reach quorum. This means the cluster is unable to add or remove a node or to commit any additional log entries. This results in unavailability. At this point, manual intervention would be required to remove either A or B and to restart the remaining node in bootstrap mode.

A Raft cluster of 3 nodes can tolerate a single node failure while a cluster of 5 can tolerate 2 node failures. The recommended configuration is to either run 3 or 5 Consul servers per datacenter. This maximizes availability without greatly sacrificing performance. The [deployment table](https://www.consul.io/docs/internals/consensus.html#deployment_table) below summarizes the potential cluster size options and the fault tolerance of each.

In terms of performance, Raft is comparable to Paxos. Assuming stable leadership, committing a log entry requires a single round trip to half of the cluster. Thus, performance is bound by disk I/O and network latency. Although Consul is not designed to be a high-throughput write system, it should handle on the order of hundreds to thousands of transactions per second depending on network and hardware configuration.

## [»](https://www.consul.io/docs/internals/consensus.html" \l "raft-in-consul)Raft in Consul

Only Consul server nodes participate in Raft and are part of the peer set. All client nodes forward requests to servers. Part of the reason for this design is that, as more members are added to the peer set, the size of the quorum also increases. This introduces performance problems as you may be waiting for hundreds of machines to agree on an entry instead of a handful.

When getting started, a single Consul server is put into "bootstrap" mode. This mode allows it to self-elect as a leader. Once a leader is elected, other servers can be added to the peer set in a way that preserves consistency and safety. Eventually, once the first few servers are added, bootstrap mode can be disabled. See [this document](https://www.consul.io/docs/install/bootstrapping.html) for more details.

Since all servers participate as part of the peer set, they all know the current leader. When an RPC request arrives at a non-leader server, the request is forwarded to the leader. If the RPC is a query type, meaning it is read-only, the leader generates the result based on the current state of the FSM. If the RPC is a transaction type, meaning it modifies state, the leader generates a new log entry and applies it using Raft. Once the log entry is committed and applied to the FSM, the transaction is complete.

Because of the nature of Raft's replication, performance is sensitive to network latency. For this reason, each datacenter elects an independent leader and maintains a disjoint peer set. Data is partitioned by datacenter, so each leader is responsible only for data in their datacenter. When a request is received for a remote datacenter, the request is forwarded to the correct leader. This design allows for lower latency transactions and higher availability without sacrificing consistency.

## [»](https://www.consul.io/docs/internals/consensus.html" \l "consistency-modes)Consistency Modes

Although all writes to the replicated log go through Raft, reads are more flexible. To support various trade-offs that developers may want, Consul supports 3 different consistency modes for reads.

The three read modes are:

* [default](https://www.consul.io/docs/internals/consensus.html" \l "default) - Raft makes use of leader leasing, providing a time window in which the leader assumes its role is stable. However, if a leader is partitioned from the remaining peers, a new leader may be elected while the old leader is holding the lease. This means there are 2 leader nodes. There is no risk of a split-brain since the old leader will be unable to commit new logs. However, if the old leader services any reads, the values are potentially stale. The default consistency mode relies only on leader leasing, exposing clients to potentially stale values. We make this trade-off because reads are fast, usually strongly consistent, and only stale in a hard-to-trigger situation. The time window of stale reads is also bounded since the leader will step down due to the partition.

* [consistent](https://www.consul.io/docs/internals/consensus.html" \l "consistent) - This mode is strongly consistent without caveats. It requires that a leader verify with a quorum of peers that it is still leader. This introduces an additional round-trip to all server nodes. The trade-off is always consistent reads but increased latency due to the extra round trip.

* [stale](https://www.consul.io/docs/internals/consensus.html" \l "stale) - This mode allows any server to service the read regardless of whether it is the leader. This means reads can be arbitrarily stale but are generally within 50 milliseconds of the leader. The trade-off is very fast and scalable reads but with stale values. This mode allows reads without a leader meaning a cluster that is unavailable will still be able to respond.

For more documentation about using these various modes, see the [HTTP API](https://www.consul.io/api/features/consistency.html).

## [»](https://www.consul.io/docs/internals/consensus.html" \l "deployment-table)Deployment Table

Below is a table that shows quorum size and failure tolerance for various cluster sizes. The recommended deployment is either 3 or 5 servers. A single server deployment is ***highly*** discouraged as data loss is inevitable in a failure scenario.

|  |  |  |
| --- | --- | --- |
| **Servers** | **Quorum Size** | **Failure Tolerance** |
| 1 | 1 | 0 |
| 2 | 2 | 0 |
| 3 | 2 | 1 |
| 4 | 3 | 1 |
| 5 | 3 | 2 |
| 6 | 4 | 2 |
| 7 | 4 | 3 |

# Gossip Protocol

Consul uses a [gossip protocol](https://en.wikipedia.org/wiki/Gossip_protocol) to manage membership and broadcast messages to the cluster. All of this is provided through the use of the [Serf library](https://www.serf.io/). The gossip protocol used by Serf is based on ["SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol"](http://www.cs.cornell.edu/info/projects/spinglass/public_pdfs/swim.pdf), with a few minor adaptations. There are more details about [Serf's protocol here](https://www.serf.io/docs/internals/gossip.html).

## [»](https://www.consul.io/docs/internals/gossip.html" \l "gossip-in-consul)Gossip in Consul

Consul makes use of two different gossip pools. We refer to each pool as the LAN or WAN pool respectively. Each datacenter Consul operates in has a LAN gossip pool containing all members of the datacenter, both clients and servers. The LAN pool is used for a few purposes. Membership information allows clients to automatically discover servers, reducing the amount of configuration needed. The distributed failure detection allows the work of failure detection to be shared by the entire cluster instead of concentrated on a few servers. Lastly, the gossip pool allows for reliable and fast event broadcasts.

The WAN pool is globally unique, as all servers should participate in the WAN pool regardless of datacenter. Membership information provided by the WAN pool allows servers to perform cross datacenter requests. The integrated failure detection allows Consul to gracefully handle an entire datacenter losing connectivity, or just a single server in a remote datacenter.

All of these features are provided by leveraging [Serf](https://www.serf.io/). It is used as an embedded library to provide these features. From a user perspective, this is not important, since the abstraction should be masked by Consul. It can be useful however as a developer to understand how this library is leveraged.

## [»](https://www.consul.io/docs/internals/gossip.html" \l "lifeguard-enhancements)Lifeguard Enhancements

SWIM makes the assumption that the local node is healthy in the sense that soft real-time processing of packets is possible. However, in cases where the local node is experiencing CPU or network exhaustion this assumption can be violated. The result is that the serfHealth check status can occasionally flap, resulting in false monitoring alarms, adding noise to telemetry, and simply causing the overall cluster to waste CPU and network resources diagnosing a failure that may not truly exist.

Lifeguard completely resolves this issue with novel enhancements to SWIM.

For more details about Lifeguard, please see the [Making Gossip More Robust with Lifeguard](https://www.hashicorp.com/blog/making-gossip-more-robust-with-lifeguard/) blog post, which provides a high level overview of the HashiCorp Research paper [Lifeguard : SWIM-ing with Situational Awareness](https://arxiv.org/abs/1707.00788). The [Serf gossip protocol guide](https://www.serf.io/docs/internals/gossip.html#lifeguard) also provides some lower-level details about the gossip protocol and Lifeguard.

# Network Coordinates

Consul uses a [network tomography](https://en.wikipedia.org/wiki/Network_tomography) system to compute network coordinates for nodes in the cluster. These coordinates allow the network round trip time to be estimated between any two nodes using a very simple calculation. This allows for many useful applications, such as finding the service node nearest a requesting node, or failing over to services in the next closest datacenter.

All of this is provided through the use of the [Serf library](https://www.serf.io/). Serf's network tomography is based on ["Vivaldi: A Decentralized Network Coordinate System"](http://www.cs.ucsb.edu/~ravenben/classes/276/papers/vivaldi-sigcomm04.pdf), with some enhancements based on other research. There are more details about [Serf's network coordinates here](https://www.serf.io/docs/internals/coordinates.html).

## [»](https://www.consul.io/docs/internals/coordinates.html" \l "network-coordinates-in-consul)Network Coordinates in Consul

Network coordinates manifest in several ways inside Consul:

* The [consul rtt](https://www.consul.io/docs/commands/rtt.html) command can be used to query for the network round trip time between any two nodes.
* The [Catalog endpoints](https://www.consul.io/api/catalog.html) and [Health endpoints](https://www.consul.io/api/health.html) can sort the results of queries based on the network round trip time from a given node using a "?near=" parameter.
* [Prepared queries](https://www.consul.io/api/query.html) can automatically fail over services to other Consul datacenters based on network round trip times. See the [Geo Failover](https://learn.hashicorp.com/consul/developer-discovery/geo-failover) for some examples.
* The [Coordinate endpoint](https://www.consul.io/api/coordinate.html) exposes raw network coordinates for use in other applications.

Consul uses Serf to manage two different gossip pools, one for the LAN with members of a given datacenter, and one for the WAN which is made up of just the Consul servers in all datacenters. It's important to note that **network coordinates are not compatible between these two pools**. LAN coordinates only make sense in calculations with other LAN coordinates, and WAN coordinates only make sense with other WAN coordinates.

## [»](https://www.consul.io/docs/internals/coordinates.html" \l "working-with-coordinates)Working with Coordinates

Computing the estimated network round trip time between any two nodes is simple once you have their coordinates. Here's a sample coordinate, as returned from the [Coordinate endpoint](https://www.consul.io/api/coordinate.html).

"Coord": {

"Adjustment": 0.1,

"Error": 1.5,

"Height": 0.02,

"Vec": [0.34,0.68,0.003,0.01,0.05,0.1,0.34,0.06]

}

All values are floating point numbers in units of seconds, except for the error term which isn't used for distance calculations.

Here's a complete example in Go showing how to compute the distance between two coordinates:

import (

"math"

"time"

"github.com/hashicorp/serf/coordinate"

)

func dist(a \*coordinate.Coordinate, b \*coordinate.Coordinate) time.Duration {

// Coordinates will always have the same dimensionality, so this is

// just a sanity check.

if len(a.Vec) != len(b.Vec) {

panic("dimensions aren't compatible")

}

// Calculate the Euclidean distance plus the heights.

sumsq := 0.0

for i := 0; i < len(a.Vec); i++ {

diff := a.Vec[i] - b.Vec[i]

sumsq += diff \* diff

}

rtt := math.Sqrt(sumsq) + a.Height + b.Height

// Apply the adjustment components, guarding against negatives.

adjusted := rtt + a.Adjustment + b.Adjustment

if adjusted > 0.0 {

rtt = adjusted

}

// Go's times are natively nanoseconds, so we convert from seconds.

const secondsToNanoseconds = 1.0e9

return time.Duration(rtt \* secondsToNanoseconds)

}

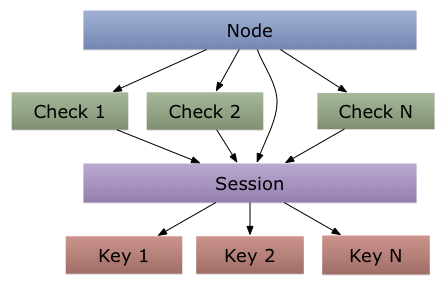
# Sessions

Consul provides a session mechanism which can be used to build distributed locks. Sessions act as a binding layer between nodes, health checks, and key/value data. They are designed to provide granular locking and are heavily inspired by [The Chubby Lock Service for Loosely-Coupled Distributed Systems](http://research.google.com/archive/chubby.html).

## [»](https://www.consul.io/docs/internals/sessions.html" \l "session-design)Session Design

A session in Consul represents a contract that has very specific semantics. When a session is constructed, a node name, a list of health checks, a behavior, a TTL, and a lock-delay may be provided. The newly constructed session is provided with a named ID that can be used to identify it. This ID can be used with the KV store to acquire locks: advisory mechanisms for mutual exclusion.

Below is a diagram showing the relationship between these components:



The contract that Consul provides is that under any of the following situations, the session will be invalidated:

* Node is deregistered
* Any of the health checks are deregistered
* Any of the health checks go to the critical state
* Session is explicitly destroyed
* TTL expires, if applicable

When a session is invalidated, it is destroyed and can no longer be used. What happens to the associated locks depends on the behavior specified at creation time. Consul supports a release and delete behavior. The release behavior is the default if none is specified.

If the release behavior is being used, any of the locks held in association with the session are released, and the ModifyIndex of the key is incremented. Alternatively, if the delete behavior is used, the key corresponding to any of the held locks is simply deleted. This can be used to create ephemeral entries that are automatically deleted by Consul.

While this is a simple design, it enables a multitude of usage patterns. By default, the [gossip based failure detector](https://www.consul.io/docs/internals/gossip.html) is used as the associated health check. This failure detector allows Consul to detect when a node that is holding a lock has failed and to automatically release the lock. This ability provides **liveness** to Consul locks; that is, under failure the system can continue to make progress. However, because there is no perfect failure detector, it's possible to have a false positive (failure detected) which causes the lock to be released even though the lock owner is still alive. This means we are sacrificing some **safety**.

Conversely, it is possible to create a session with no associated health checks. This removes the possibility of a false positive and trades liveness for safety. You can be absolutely certain Consul will not release the lock even if the existing owner has failed. Since Consul APIs allow a session to be force destroyed, this allows systems to be built that require an operator to intervene in the case of a failure while precluding the possibility of a split-brain.

A third health checking mechanism is session TTLs. When creating a session, a TTL can be specified. If the TTL interval expires without being renewed, the session has expired and an invalidation is triggered. This type of failure detector is also known as a heartbeat failure detector. It is less scalable than the gossip based failure detector as it places an increased burden on the servers but may be applicable in some cases. The contract of a TTL is that it represents a lower bound for invalidation; that is, Consul will not expire the session before the TTL is reached, but it is allowed to delay the expiration past the TTL. The TTL is renewed on session creation, on session renew, and on leader failover. When a TTL is being used, clients should be aware of clock skew issues: namely, time may not progress at the same rate on the client as on the Consul servers. It is best to set conservative TTL values and to renew in advance of the TTL to account for network delay and time skew.

The final nuance is that sessions may provide a lock-delay. This is a time duration, between 0 and 60 seconds. When a session invalidation takes place, Consul prevents any of the previously held locks from being re-acquired for the lock-delay interval; this is a safeguard inspired by Google's Chubby. The purpose of this delay is to allow the potentially still live leader to detect the invalidation and stop processing requests that may lead to inconsistent state. While not a bulletproof method, it does avoid the need to introduce sleep states into application logic and can help mitigate many issues. While the default is to use a 15 second delay, clients are able to disable this mechanism by providing a zero delay value.

## [»](https://www.consul.io/docs/internals/sessions.html" \l "k-v-integration)K/V Integration

Integration between the KV store and sessions is the primary place where sessions are used. A session must be created prior to use and is then referred to by its ID.

The KV API is extended to support an acquire and release operation. The acquire operation acts like a Check-And-Set operation except it can only succeed if there is no existing lock holder (the current lock holder can re-acquire, see below). On success, there is a normal key update, but there is also an increment to the LockIndex, and the Session value is updated to reflect the session holding the lock.

If the lock is already held by the given session during an acquire, then the LockIndex is not incremented but the key contents are updated. This lets the current lock holder update the key contents without having to give up the lock and reacquire it.

Once held, the lock can be released using a corresponding release operation, providing the same session. Again, this acts like a Check-And-Set operation since the request will fail if given an invalid session. A critical note is that the lock can be released without being the creator of the session. This is by design as it allows operators to intervene and force-terminate a session if necessary. As mentioned above, a session invalidation will also cause all held locks to be released or deleted. When a lock is released, the LockIndex does not change; however, the Session is cleared and the ModifyIndex increments.

These semantics (heavily borrowed from Chubby), allow the tuple of (Key, LockIndex, Session) to act as a unique "sequencer". This sequencer can be passed around and used to verify if the request belongs to the current lock holder. Because the LockIndex is incremented on each acquire, even if the same session re-acquires a lock, the sequencer will be able to detect a stale request. Similarly, if a session is invalided, the Session corresponding to the given LockIndex will be blank.

To be clear, this locking system is purely advisory. There is no enforcement that clients must acquire a lock to perform any operation. Any client can read, write, and delete a key without owning the corresponding lock. It is not the goal of Consul to protect against misbehaving clients.

## [»](https://www.consul.io/docs/internals/sessions.html" \l "leader-election)Leader Election

The primitives provided by sessions and the locking mechanisms of the KV store can be used to build client-side leader election algorithms. These are covered in more detail in the [Leader Election guide](https://learn.hashicorp.com/consul/developer-configuration/elections).

## [»](https://www.consul.io/docs/internals/sessions.html" \l "prepared-query-integration)Prepared Query Integration

Prepared queries may be attached to a session in order to automatically delete the prepared query when the session is invalidated.

# Anti-Entropy

Consul uses an advanced method of maintaining service and health information. This page details how services and checks are registered, how the catalog is populated, and how health status information is updated as it changes.

### [»](https://www.consul.io/docs/internals/anti-entropy.html" \l "components)Components

It is important to first understand the moving pieces involved in services and health checks: the [agent](https://www.consul.io/docs/internals/anti-entropy.html#agent) and the [catalog](https://www.consul.io/docs/internals/anti-entropy.html#catalog). These are described conceptually below to make anti-entropy easier to understand.

#### [»](https://www.consul.io/docs/internals/anti-entropy.html" \l "agent)Agent

Each Consul agent maintains its own set of service and check registrations as well as health information. The agents are responsible for executing their own health checks and updating their local state.

Services and checks within the context of an agent have a rich set of configuration options available. This is because the agent is responsible for generating information about its services and their health through the use of [health checks](https://www.consul.io/docs/agent/checks.html).

#### [»](https://www.consul.io/docs/internals/anti-entropy.html" \l "catalog)Catalog

Consul's service discovery is backed by a service catalog. This catalog is formed by aggregating information submitted by the agents. The catalog maintains the high-level view of the cluster, including which services are available, which nodes run those services, health information, and more. The catalog is used to expose this information via the various interfaces Consul provides, including DNS and HTTP.

Services and checks within the context of the catalog have a much more limited set of fields when compared with the agent. This is because the catalog is only responsible for recording and returning information about services, nodes, and health.

The catalog is maintained only by server nodes. This is because the catalog is replicated via the [Raft log](https://www.consul.io/docs/internals/consensus.html) to provide a consolidated and consistent view of the cluster.

### [»](https://www.consul.io/docs/internals/anti-entropy.html" \l "anti-entropy-1)Anti-Entropy

Entropy is the tendency of systems to become increasingly disordered. Consul's anti-entropy mechanisms are designed to counter this tendency, to keep the state of the cluster ordered even through failures of its components.

Consul has a clear separation between the global service catalog and the agent's local state as discussed above. The anti-entropy mechanism reconciles these two views of the world: anti-entropy is a synchronization of the local agent state and the catalog. For example, when a user registers a new service or check with the agent, the agent in turn notifies the catalog that this new check exists. Similarly, when a check is deleted from the agent, it is consequently removed from the catalog as well.

Anti-entropy is also used to update availability information. As agents run their health checks, their status may change in which case their new status is synced to the catalog. Using this information, the catalog can respond intelligently to queries about its nodes and services based on their availability.

During this synchronization, the catalog is also checked for correctness. If any services or checks exist in the catalog that the agent is not aware of, they will be automatically removed to make the catalog reflect the proper set of services and health information for that agent. Consul treats the state of the agent as authoritative; if there are any differences between the agent and catalog view, the agent-local view will always be used.

### [»](https://www.consul.io/docs/internals/anti-entropy.html" \l "periodic-synchronization)Periodic Synchronization

In addition to running when changes to the agent occur, anti-entropy is also a long-running process which periodically wakes up to sync service and check status to the catalog. This ensures that the catalog closely matches the agent's true state. This also allows Consul to re-populate the service catalog even in the case of complete data loss.

To avoid saturation, the amount of time between periodic anti-entropy runs will vary based on cluster size. The table below defines the relationship between cluster size and sync interval:

|  |  |
| --- | --- |
| **Cluster Size** | **Periodic Sync Interval** |
| 1 - 128 | 1 minute |
| 129 - 256 | 2 minutes |
| 257 - 512 | 3 minutes |
| 513 - 1024 | 4 minutes |
| ... | ... |

The intervals above are approximate. Each Consul agent will choose a randomly staggered start time within the interval window to avoid a thundering herd.

### [»](https://www.consul.io/docs/internals/anti-entropy.html" \l "best-effort-sync)Best-effort sync

Anti-entropy can fail in a number of cases, including misconfiguration of the agent or its operating environment, I/O problems (full disk, filesystem permission, etc.), networking problems (agent cannot communicate with server), among others. Because of this, the agent attempts to sync in best-effort fashion.

If an error is encountered during an anti-entropy run, the error is logged and the agent continues to run. The anti-entropy mechanism is run periodically to automatically recover from these types of transient failures.

### [»](https://www.consul.io/docs/internals/anti-entropy.html" \l "enable-tag-override)Enable Tag Override

Synchronization of service registration can be partially modified to allow external agents to change the tags for a service. This can be useful in situations where an external monitoring service needs to be the source of truth for tag information. For example, the Redis database and its monitoring service Redis Sentinel have this kind of relationship. Redis instances are responsible for much of their configuration, but Sentinels determine whether the Redis instance is a primary or a secondary. Using the Consul service configuration item [enable\_tag\_override](https://www.consul.io/docs/agent/services.html) you can instruct the Consul agent on which the Redis database is running to NOT update the tags during anti-entropy synchronization. For more information see [Services](https://www.consul.io/docs/agent/services.html#enable-tag-override-and-anti-entropy) page.

# Security Model

Consul relies on both a lightweight gossip mechanism and an RPC system to provide various features. Both of the systems have different security mechanisms that stem from their designs. However, the security mechanisms of Consul have a common goal: to provide [confidentiality, integrity, and authentication](https://en.wikipedia.org/wiki/Information_security).

The [gossip protocol](https://www.consul.io/docs/internals/gossip.html) is powered by [Serf](https://www.serf.io/), which uses a symmetric key, or shared secret, cryptosystem. There are more details on the security of [Serf here](https://www.serf.io/docs/internals/security.html). For details on how to enable Serf's gossip encryption in Consul, see the [encryption doc here](https://www.consul.io/docs/agent/encryption.html).

The RPC system supports using end-to-end TLS with optional client authentication. [TLS](https://en.wikipedia.org/wiki/Transport_Layer_Security) is a widely deployed asymmetric cryptosystem and is the foundation of security on the Web.

This means Consul communication is protected against eavesdropping, tampering, and spoofing. This makes it possible to run Consul over untrusted networks such as EC2 and other shared hosting providers.

## [»](https://www.consul.io/docs/internals/security.html" \l "secure-configuration)Secure Configuration

The Consul threat model is only applicable if Consul is running in a secure configuration. Consul does not operate in a secure-by-default configuration. If any of the settings below are not enabled, then parts of this threat model are going to be invalid. Additional security precautions must also be taken for items outside of Consul's threat model as noted in sections below.

* **ACLs enabled with default deny.** Consul must be configured to use ACLs with a whitelist (default deny) approach. This forces all requests to have explicit anonymous access or provide an ACL token.
* **Encryption enabled.** TCP and UDP encryption must be enabled and configured to prevent plaintext communication between Consul agents. At a minimum, verify\_outgoing should be enabled to verify server authenticity with each server having a unique TLS certificate. verify\_server\_hostname is also required to prevent a compromised agent restarting as a server and being given access to all secrets.

verify\_incoming provides additional agent verification via mutual authentication, but isn't strictly necessary to enforce the threat model since requests must also contain a valid ACL token. The subtlety is that currently verify\_incoming = false will allow servers to still accept un-encrypted connections from clients (to allow for gradual TLS rollout). That alone doesn't violate the threat model, but any misconfigured client that chooses not to use TLS will violate the model. We recommend setting this to true. If it is left as false care must be taken to ensure all consul clients use verify\_outgoing = true as noted above, but also all external API/UI access must be via HTTPS with HTTP listeners disabled.

### [»](https://www.consul.io/docs/internals/security.html" \l "known-insecure-configurations)Known Insecure Configurations

In addition to configuring the non-default settings above, Consul has several non-default options that potentially present additional security risks.

* **Script checks enabled with network-exposed API.** If a Consul agent (client or server) exposes its HTTP API to the network beyond localhost, [enable\_script\_checks](https://www.consul.io/docs/agent/options.html" \l "_enable_script_checks) must be false otherwise, even with ACLs configured, script checks present a remote code execution threat. [enable\_local\_script\_checks](https://www.consul.io/docs/agent/options.html" \l "_enable_local_script_checks) provides a secure alternative if the HTTP API must be exposed and is available from 1.3.0 on. This feature was also back-ported to patch releases 0.9.4, 1.1.1, and 1.2.4 [as described here](https://www.hashicorp.com/blog/protecting-consul-from-rce-risk-in-specific-configurations).
* **Remote exec enabled.** Consul includes a [consul exec feature](https://www.consul.io/docs/commands/exec.html) allowing execution of arbitrary commands across the cluster. This is disabled by default since 0.8.0. We recommend leaving it disabled. If enabled, extreme care must be taken to ensure correct ACLs restrict access, for example any management token grants access to execute arbitrary code on the cluster.
* **Verify Server Hostname Used Alone.** From version 0.5.1 to 1.4.0 we documented that verify\_server\_hostname being true implied verify\_outgoing however due to a bug this was not the case so setting only verify\_server\_hostname results in plaintext communication between client and server. See [CVE-2018-19653](https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-19653) for more details. This is fixed in 1.4.1.

## [»](https://www.consul.io/docs/internals/security.html" \l "threat-model)Threat Model

The following are parts of the Consul threat model:

* **Consul agent-to-agent communication.** Communication between Consul agents should be secure from eavesdropping. This requires transport encryption to be enabled on the cluster and covers both TCP and UDP traffic.
* **Consul agent-to-CA communication.** Communication between the Consul server and the configured certificate authority provider for Connect is always encrypted.
* **Tampering of data in transit.** Any tampering should be detectable and cause Consul to avoid processing the request.
* **Access to data without authentication or authorization.** All requests must be authenticated and authorized. This requires that ACLs are enabled on the cluster with a default deny mode.
* **State modification or corruption due to malicious messages.** Ill-formatted messages are discarded and well-formatted messages require authentication and authorization.
* **Non-server members accessing raw data.** All servers must join the cluster (with proper authentication and authorization) to begin participating in Raft. Raft data is transmitted over TLS.
* **Denial of Service against a node.** DoS attacks against a node should not compromise the security stance of the software.
* **Connect-based Service-to-Service communication.** Communications between two Connect-enabled services (natively or by proxy) should be secure from eavesdropping and provide authentication. This is achieved via mutual TLS.

The following are not part of the Consul threat model for Consul server agents:

* **Access (read or write) to the Consul data directory.** All Consul servers, including non-leaders, persist the full set of Consul state to this directory. The data includes all KV, service registrations, ACL tokens, Connect CA configuration, and more. Any read or write to this directory allows an attacker to access and tamper with that data.
* **Access (read or write) to the Consul configuration directory.** Consul configuration can enable or disable the ACL system, modify data directory paths, and more. Any read or write of this directory allows an attacker to reconfigure many aspects of Consul. By disabling the ACL system, this may give an attacker access to all Consul data.
* **Memory access to a running Consul server agent.** If an attacker is able to inspect the memory state of a running Consul server agent the confidentiality of almost all Consul data may be compromised. If you're using an external Connect CA, the root private key material is never available to the Consul process and can be considered safe. Service Connect TLS certificates should be considered compromised; they are never persisted by server agents but do exist in-memory during at least the duration of a Sign request.

The following are not part of the Consul threat model for Consul client agents:

* **Access (read or write) to the Consul data directory.** Consul clients will use the data directory to cache local state. This includes local services, associated ACL tokens, Connect TLS certificates, and more. Read or write access to this directory will allow an attacker to access this data. This data is typically a smaller subset of the full data of the cluster.
* **Access (read or write) to the Consul configuration directory.** Consul client configuration files contain the address and port information of services, default ACL tokens for the agent, and more. Access to Consul configuration could enable an attacker to change the port of a service to a malicious port, register new services, and more. Further, some service definitions have ACL tokens attached that could be used cluster-wide to impersonate that service. An attacker cannot change cluster-wide configurations such as disabling the ACL system.
* **Memory access to a running Consul client agent.** The blast radius of this is much smaller than a server agent but the confidentiality of a subset of data can still be compromised. Particularly, any data requested against the agent's API including services, KV, and Connect information may be compromised. If a particular set of data on the server was never requested by the agent, it never enters the agent's memory since replication only exists between servers. An attacker could also potentially extract ACL tokens used for service registration on this agent, since the tokens must be stored in-memory alongside the registered service.
* **Network access to a local Connect proxy or service.** Communications between a service and a Connect-aware proxy are generally unencrypted and must happen over a trusted network. This is typically a loopback device. This requires that other processes on the same machine are trusted, or more complex isolation mechanisms are used such as network namespaces. This also requires that external processes cannot communicate to the Connect service or proxy (except on the inbound port). Therefore, non-native Connect applications should only bind to non-public addresses.
* **Improperly Implemented Connect proxy or service.** A Connect proxy or natively integrated service must correctly serve a valid leaf certificate, verify the inbound TLS client certificate, and call the Consul agent-local authorize endpoint. If any of this isn't performed correctly, the proxy or service may allow unauthenticated or unauthorized connections.

## [»](https://www.consul.io/docs/internals/security.html" \l "external-threat-overview)External Threat Overview

There are four components that affect the Consul threat model: the server agent, the client agent, the Connect CA, and Consul API clients (including proxies for Connect).

The server agent participates in leader election and data replication via Raft. All communications with other agents is encrypted. Data is stored at rest unencrypted in the configured data directory. The stored data includes ACL tokens and TLS certificates. If the built-in CA is used with Connect, root certificate private keys are also stored on disk. External CA providers do not store data in this directory. This data directory must be carefully protected to prevent an attacker from impersonating a server or specific ACL user. We plan to introduce further mitigations (including at least partial data encryption) to the data directory over time, but the data directory should always be considered secret.

For a client agent to join a cluster, it must provide a valid ACL token with node:write capabilities. The join request and all other API requests between the client and server agents communicate via TLS. Clients serve the Consul API and forward all requests to a server over a shared TLS connection. Each request contains an ACL token which is used for both authentication and authorization. Requests that do not provide an ACL token inherit the agent-configurable default ACL token.

The Connect CA provider is responsible for storing the private key of the root (or intermediate) certificate used to sign and verify connections established via Connect. Consul server agents communicate with the CA provider via an encrypted method. This method is dependent on the CA provider in use. Consul provides a built-in CA which performs all operations locally on the server agent. Consul itself does not store any private key material except for the built-in CA.

Consul API clients (the agent itself, the built-in UI, external software) must communicate to a Consul agent over TLS and must provide an ACL token per request for authentication and authorization.

## [»](https://www.consul.io/docs/internals/security.html" \l "network-ports)Network Ports

For configuring network rules to support Consul, please see [Ports Used](https://www.consul.io/docs/agent/options.html#ports) for a listing of network ports used by Consul and details about which features they are used for.

**Consul vs. Eureka**

Eureka is a service discovery tool. The architecture is primarily client/server, with a set of Eureka servers per datacenter, usually one per availability zone. Typically clients of Eureka use an embedded SDK to register and discover services. For clients that are not natively integrated, a sidecar such as Ribbon is used to transparently discover services via Eureka.

Eureka provides a weakly consistent view of services, using best effort replication. When a client registers with a server, that server will make an attempt to replicate to the other servers but provides no guarantee. Service registrations have a short Time-To-Live (TTL), requiring clients to heartbeat with the servers. Unhealthy services or nodes will stop heartbeating, causing them to timeout and be removed from the registry. Discovery requests can route to any service, which can serve stale or missing data due to the best effort replication. This simplified model allows for easy cluster administration and high scalability.

Consul provides a super set of features, including richer health checking, key/value store, and multi-datacenter awareness. Consul requires a set of servers in each datacenter, along with an agent on each client, similar to using a sidecar like Ribbon. The Consul agent allows most applications to be Consul unaware, performing the service registration via configuration files and discovery via DNS or load balancer sidecars.

Consul provides a strong consistency guarantee, since servers replicate state using the [Raft protocol](https://www.consul.io/docs/internals/consensus.html). Consul supports a rich set of health checks including TCP, HTTP, Nagios/Sensu compatible scripts, or TTL based like Eureka. Client nodes participate in a [gossip based health check](https://www.consul.io/docs/internals/gossip.html), which distributes the work of health checking, unlike centralized heartbeating which becomes a scalability challenge. Discovery requests are routed to the elected Consul leader which allows them to be strongly consistent by default. Clients that allow for stale reads enable any server to process their request allowing for linear scalability like Eureka.

The strongly consistent nature of Consul means it can be used as a locking service for leader elections and cluster coordination. Eureka does not provide similar guarantees, and typically requires running ZooKeeper for services that need to perform coordination or have stronger consistency needs.

Consul provides a toolkit of features needed to support a service oriented architecture. This includes service discovery, but also rich health checking, locking, Key/Value, multi-datacenter federation, an event system, and ACLs. Both Consul and the ecosystem of tools like consul-template and envconsul try to minimize application changes required to integration, to avoid needing native integration via SDKs. Eureka is part of a larger Netflix OSS suite, which expects applications to be relatively homogeneous and tightly integrated. As a result, Eureka only solves a limited subset of problems, expecting other tools such as ZooKeeper to be used alongside.

# Consul vs. ZooKeeper, doozerd, etcd

ZooKeeper, doozerd, and etcd are all similar in their architecture. All three have server nodes that require a quorum of nodes to operate (usually a simple majority). They are strongly-consistent and expose various primitives that can be used through client libraries within applications to build complex distributed systems.

Consul also uses server nodes within a single datacenter. In each datacenter, Consul servers require a quorum to operate and provide strong consistency. However, Consul has native support for multiple datacenters as well as a more feature-rich gossip system that links server nodes and clients.

All of these systems have roughly the same semantics when providing key/value storage: reads are strongly consistent and availability is sacrificed for consistency in the face of a network partition. However, the differences become more apparent when these systems are used for advanced cases.

The semantics provided by these systems are attractive for building service discovery systems, but it's important to stress that these features must be built. ZooKeeper et al. provide only a primitive K/V store and require that application developers build their own system to provide service discovery. Consul, by contrast, provides an opinionated framework for service discovery and eliminates the guess-work and development effort. Clients simply register services and then perform discovery using a DNS or HTTP interface. Other systems require a home-rolled solution.

A compelling service discovery framework must incorporate health checking and the possibility of failures as well. It is not useful to know that Node A provides the Foo service if that node has failed or the service crashed. Naive systems make use of heartbeating, using periodic updates and TTLs. These schemes require work linear to the number of nodes and place the demand on a fixed number of servers. Additionally, the failure detection window is at least as long as the TTL.

ZooKeeper provides ephemeral nodes which are K/V entries that are removed when a client disconnects. These are more sophisticated than a heartbeat system but still have inherent scalability issues and add client-side complexity. All clients must maintain active connections to the ZooKeeper servers and perform keep-alives. Additionally, this requires "thick clients" which are difficult to write and often result in debugging challenges.

Consul uses a very different architecture for health checking. Instead of only having server nodes, Consul clients run on every node in the cluster. These clients are part of a [gossip pool](https://www.consul.io/docs/internals/gossip.html) which serves several functions, including distributed health checking. The gossip protocol implements an efficient failure detector that can scale to clusters of any size without concentrating the work on any select group of servers. The clients also enable a much richer set of health checks to be run locally, whereas ZooKeeper ephemeral nodes are a very primitive check of liveness. With Consul, clients can check that a web server is returning 200 status codes, that memory utilization is not critical, that there is sufficient disk space, etc. The Consul clients expose a simple HTTP interface and avoid exposing the complexity of the system to clients in the same way as ZooKeeper.

Consul provides first-class support for service discovery, health checking, K/V storage, and multiple datacenters. To support anything more than simple K/V storage, all these other systems require additional tools and libraries to be built on top. By using client nodes, Consul provides a simple API that only requires thin clients. Additionally, the API can be avoided entirely by using configuration files and the DNS interface to have a complete service discovery solution with no development at all.

# Consul vs. Istio

Istio is an open platform to connect, manage, and secure microservices.

To enable the full functionality of Istio, multiple services must be deployed. For the control plane: Pilot, Mixer, and Citadel must be deployed and for the data plane an Envoy sidecar is deployed. Additionally, Istio requires a 3rd party service catalog from Kubernetes, Consul, Eureka, or others. Finally, Istio requires an external system for storing state, typically etcd. At a minimum, three Istio-dedicated services along with at least one separate distributed system (in addition to Istio) must be configured to use the full functionality of Istio.

Istio provides layer 7 features for path-based routing, traffic shaping, load balancing, and telemetry. Access control policies can be configured targeting both layer 7 and layer 4 properties to control access, routing, and more based on service identity.

Consul is a single binary providing both server and client capabilities, and includes all functionality for service catalog, configuration, TLS certificates, authorization, and more. No additional systems need to be installed to use Consul, although Consul optionally supports external systems such as Vault to augment behavior. This architecture enables Consul to be easily installed on any platform, including directly onto the machine.

Consul uses an agent-based model where each node in the cluster runs a Consul Client. This client maintains a local cache that is efficiently updated from servers. As a result, all secure service communication APIs respond in microseconds and do not require any external communication. This allows us to do connection enforcement at the edge without communicating to central servers. Istio flows requests to a central Mixer service and must push updates out via Pilot. This dramatically reduces the scalability of Istio, whereas Consul is able to efficiently distribute updates and perform all work on the edge.

Consul provides layer 7 features for path-based routing, traffic shifting, load balancing, and telemetry. Consul enforces authorization and identity to layer 4 only — either the TLS connection can be established or it can't. We believe service identity should be tied to layer 4, whereas layer 7 should be used for routing, telemetry, etc. We will be adding more layer 7 features to Consul in the future.

The data plane for Consul is pluggable. It includes a built-in proxy with a larger performance trade off for ease of use. But you may also use third party proxies such as Envoy to leverage layer 7 features. The ability to use the right proxy for the job allows flexible heterogeneous deployments where different proxies may be more correct for the applications they're proxying. We encourage users leverage the pluggable data plane layer and use a proxy which supports the layer 7 features necessary for the cluster.

In addition to third party proxy support, applications can natively integrate with the Connect protocol. As a result, the performance overhead of introducing Connect is negligible. These "Connect-native" applications can interact with any other Connect-capable services, whether they're using a proxy or are also Connect-native.

Consul implements automatic TLS certificate management complete with rotation support. Both leaf and root certificates can be rotated automatically across a large Consul cluster with zero disruption to connections. The certificate management system is pluggable through code change in Consul and will be exposed as an external plugin system shortly. This enables Consul to work with any PKI solution.

Because Consul's service connection feature "Connect" is built-in, it inherits the operational stability of Consul. Consul has been in production for large companies since 2014 and is known to be deployed on as many as 50,000 nodes in a single cluster.

This comparison is based on our own limited usage of Istio as well as talking to Istio users. If you feel there are inaccurate statements in this comparison, please click "Edit This Page" in the footer of this page and propose edits. We strive for technical accuracy and will review and update this post for inaccuracies as quickly as possible.

# Consul vs. Envoy and Other Proxies

Modern service proxies provide high-level service routing, authentication, telemetry, and more for microservice and cloud environments. Envoy is a popular and feature-rich proxy that is often used on its own. Consul [integrates with Envoy](https://www.consul.io/docs/connect/proxies/envoy.html) to simplify its configuration.

Proxies require a rich set of configuration to operate since backend addresses, frontend listeners, routes, filters, telemetry shipping, and more must all be configured. Further, a modern infrastructure contains many proxies, often one proxy per service as proxies are deployed in a "sidecar" model next to a service. Therefore, a primary challenge of proxies is the configuration sprawl and orchestration.

Proxies form what is referred to as the "data plane": the pathway which data travels for network connections. Above this is the "control plane" which provides the rules and configuration for the data plane. Proxies typically integrate with outside solutions to provide the control plane. For example, Envoy integrates with Consul to dynamically populate service backend addresses.

Consul is a control plane solution. The service catalog serves as a registry for services and their addresses and can be used to route traffic for proxies. The Connect feature of Consul provides the TLS certificates and service access graph, but still requires a proxy to exist in the data path. As a control plane, Consul integrates with many data plane solutions including Envoy, HAProxy, Nginx, and more.

The [Consul Envoy integration](https://www.consul.io/docs/connect/proxies/envoy.html) is currently the primary way to utilize advanced layer 7 features provided by Consul. In addition to Envoy, Consul enables third party proxies to integrate with Connect and provide the data plane with Consul operating as the control plane.

Proxies provide excellent solutions to layer 7 concerns such as path-based routing, tracing and telemetry, and more. By supporting a pluggable data plane model, the right proxy can be deployed as needed. For performance-critical applications or those that utilize layer 7 functionality, Envoy can be used. For non-performance critical layer 4 applications, you can use Consul's [built-in proxy](https://www.consul.io/docs/connect/proxies/built-in.html) for convenience.

For some applications that may require hardware, a hardware load balancer such an F5 appliance may be deployed. Consul encourages this use of the right proxy for the scenario and treats hardware load balancers as swappable components that can be run alongside other proxies, assuming they integrate with the [necessary APIs](https://www.consul.io/docs/connect/proxies/integrate.html) for Connect.