HashiCorp Consul

Life of a Packet Through Consul Service Mesh

HashiCorp

Contents

Preface

This white paper is intended to provide detailed information around HashiCorp Consul service mesh.

It is intended for people who are currently evaluating service mesh solutions or who are already using HashiCorp Consul as their service mesh. This white paper will help the reader obtain a deeper understanding of how Consul service mesh works and how Consul service mesh can solve challenges within a given environment. The solution will address common challenges such as setting up a common service mesh in multiple cloud and/or runtime environments, while still providing a central point of management. As a side benefit, adopters also gain the well known scalability, resiliency and ease of use of HashiCorp Consul.

A solid understanding of how HashiCorp Consul works outside the realm of service mesh is required to get the most out of this white paper.

Initially, this white paper will focus on giving some historical background around the evolution of compute and networking technologies during recent years, leading to "why" enterprises are now considering service mesh within their environments.

You will learn about the challenges new application deployment patterns like microservices are bringing to enterprises when it comes to service discovery, service-to-service connectivity, and security—especially when running on top of container orchestration platforms.

After giving a short introduction into HashiCorp Consul and its service discovery and distributed configuration features, the main focus will be how HashiCorp Consul can act as a distributed control-plane for a multi-cloud and multi-runtime service mesh, offering a state of the art service mesh feature set.

We will discuss the steps required to get started with HashiCorp Consul service mesh, and provide detailed information around the bootstrapping process and the lifecycle of sidecar proxy instances based on Envoy. We will also give details around providing authentication, authorization, and encryption for service-to-service communication, regardless of cloud platform or runtime those services reside on.

Evolution of Networking towards Cloud Native Solutions

During the last decade, compute as well as network technologies – especially within enterprise data centers – have evolved rapidly.

In nearly every evolution of computing—from mainframe, to client-server, virtualization, and modern orchestration platforms like Kubernetes or public cloud services—networking has struggled to evolve to meet the needs of compute functionalities.

Every new compute technology puts pressure on enterprise networking teams to re-architect their networking environments to meet new connectivity, reliability, and resiliency requirements while still ensuring security within the enterprise data center.

In traditional compute environments, before virtualization, networking operators could secure network segments and communication into the data center with ease using IP-based firewalls. Services typically ran on a single server which had a well known and unchanging IP address, which made it easy for operators to define policies and enforce application security.

Authentication and authorization of service communication within the network was enforced based on this clear association of IP address to a physical server. If a firewall inspected a packet, it could be assured a specific service had initiated the connection attempt, and allow or deny the connection based on the configured policy.

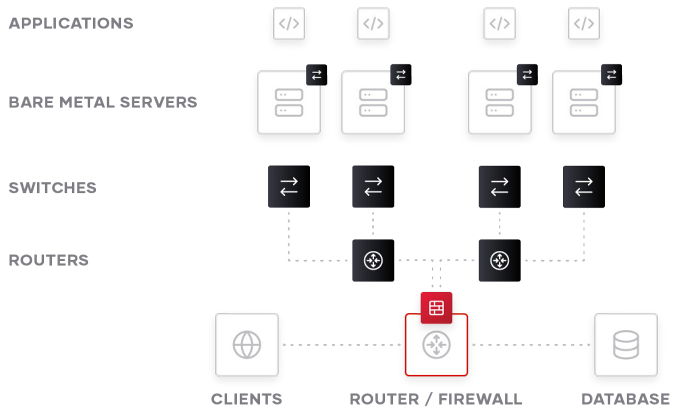


Figure 1: Traditional client-server network topology.

These network architectures were built to support monolithic applications and traffic patterns which were mostly north-south, meaning traffic coming from an external client into the network to a specific application, e.g. Internet users hitting a web frontend application.

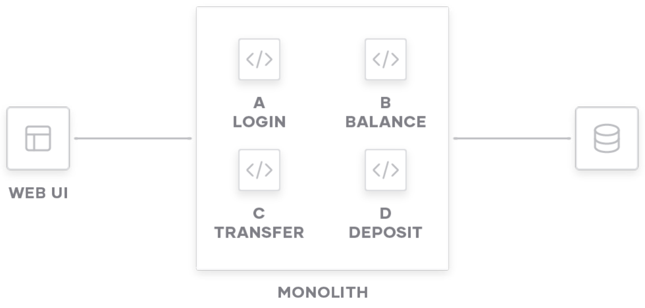


Figure 2: A monolithic banking application.

In order to achieve high availability for east-west traffic (e.g., traffic from a web frontend to a backend database), or to backend systems, enterprises often placed load balancers in front of applications. Load balancers simplified internal service discovery by providing a single IP address for frontend application connections, and implemented health checking to ensure traffic was only routed to healthy backend instances.

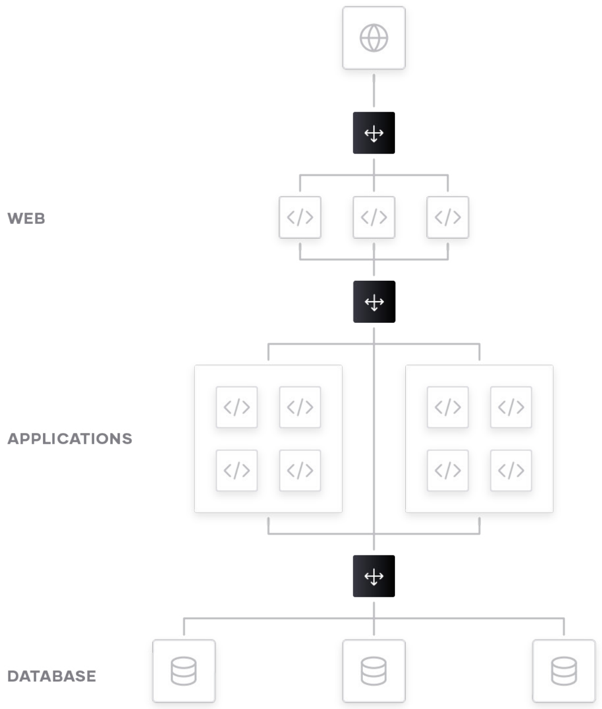


Figure 3: A static network topology with load balancers fronting the database, monolithic applications, and the web front-end.

With the rise of virtualization technologies enterprise data centers became more dynamic, as it was now much easier for administrators to spin up and down systems through virtual machine templates or automation tools.

With this newly evolved automation of virtual machine based environments and the associated features in terms of high-availability, elasticity and the possibility to automatically scale up/down or restart failed VMs on different hosts, network administrators faced new challenges; administrators were now demanding for the same IP address space/VLAN to be available across the entire data center so their VMs could be automatically restarted or moved by the VM orchestration solution seamlessly without the need to reconfigure the IP address of the virtual machine itself.

With the available technologies like Spanning-Tree Protocol, the growth of L2 broadcast domains in a data center network can cause network instability over a given size of the network. To keep up with the demand of even bigger L2 domains which needed to spread across wider parts of the infrastructure, networking vendors came up with L2 fabric technologies such as TRILL etc. which help solve the challenges associated with building scalable and robust L2 fabrics.

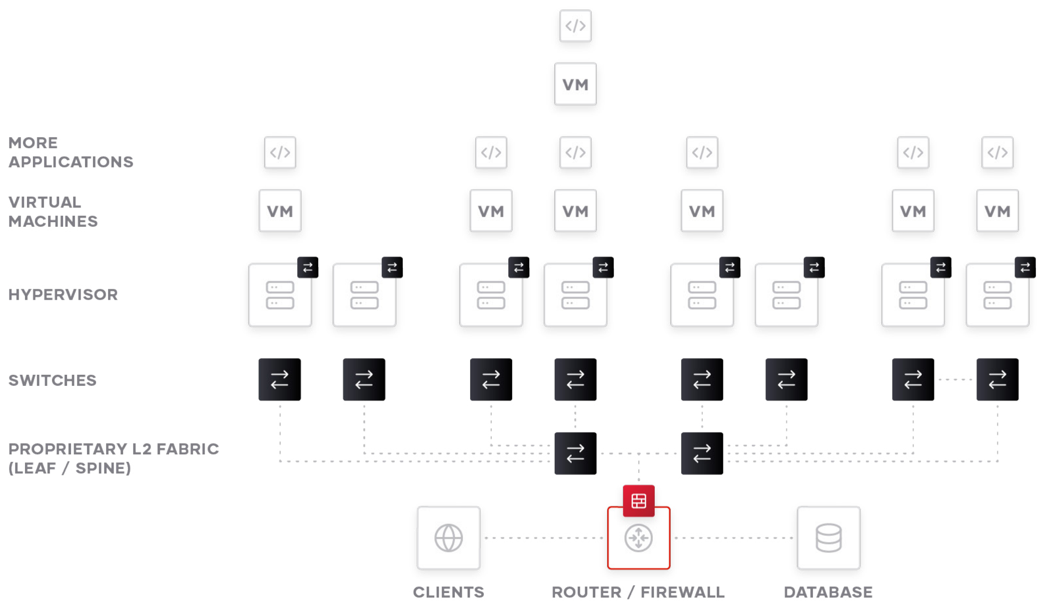


Figure 4: Layer 2 network fabric topology.

Nevertheless, other challenges which were brought to the networking teams through the rise of virtual machines and automated deployments could not be easily solved by simply scaling L2 domains: The challenges of Service discovery and authorization of service-to-service communication, still enforced by IP based firewalls, were still manual, ticket driven processes.

Although it might take minutes or hours to deploy a virtual machine to scale an application tier, it could take several days or weeks until actual production traffic could be handled by this virtual machine.

The reason behind this was mostly related to the lack of automated provisioning for network middleware components like load balancers (mostly used for Service discovery between application tiers) and firewalls.

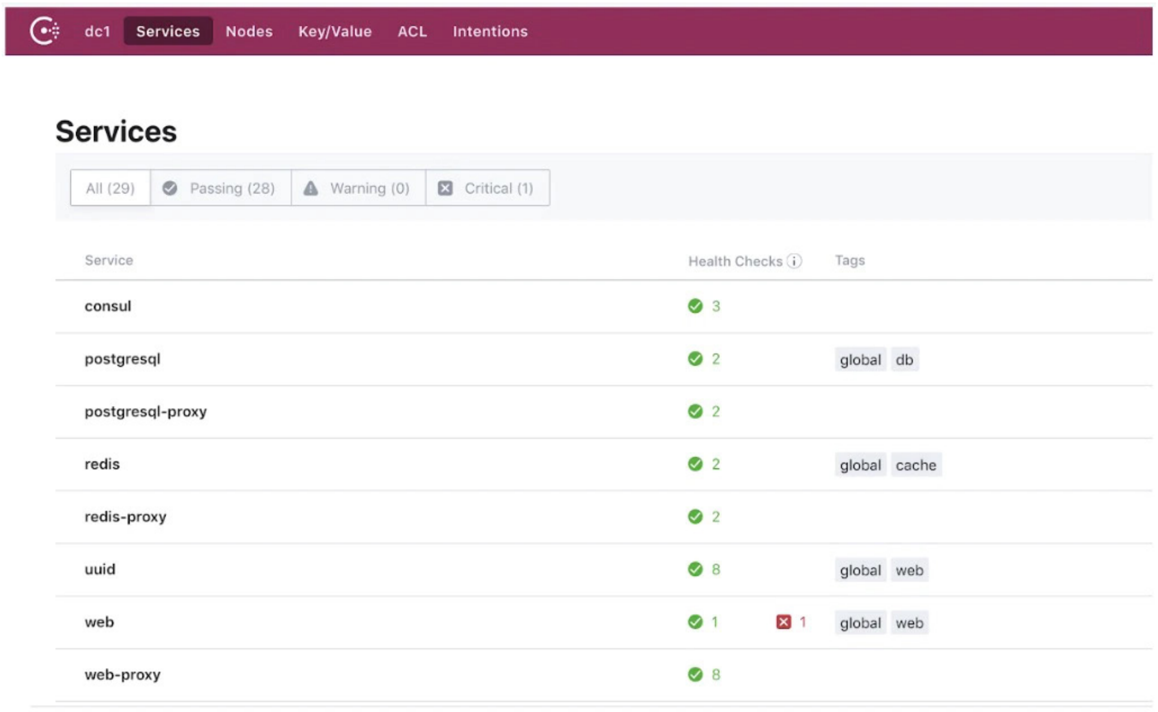
After deploying a virtual machine in an automated fashion, networking teams still needed to manually add the IP address of this newly deployed VM to load balancer pools which could take time depending on the underlying process.

Leveraging costly load balancers within a data center for east-west service discovery results in increased costs and can lead to bottlenecks for service traffic within a data center.

HashiCorp Consul offers out of the box stateless, randomized load balancing which is suitable for many east-west intra-datacenter use-cases, where no advanced load balancing features are needed.

By leveraging HashiCorp Consul as the Service discovery component, instead of hard-coding a load balancer's virtual IP address into an application, networking teams can get rid of the manual, error prone process of manually re-configuring a load balancer's backend server pool each and every time a new virtual machine gets deployed.

HashiCorp Consul's Service Registry provides a central directory of what services are running, where they are, and their current health status.



Services

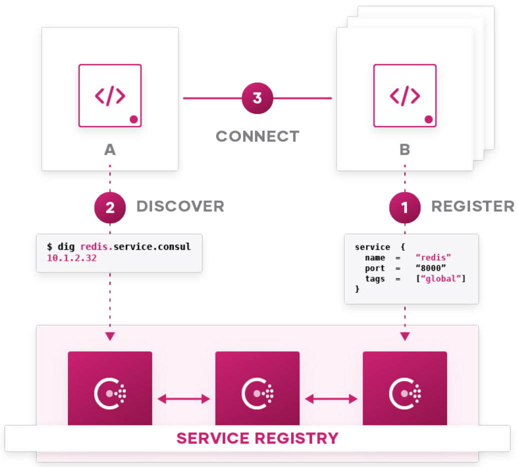
Service	Health Checks	Tags
consul	✓ 3	
postgresql	✓ 2	global db
postgresql-proxy	✓ 2	
redis	✓ 2	global cache
redis-proxy	✓ 2	
uuid	✓ 8	global web
web	✓ 1 ✗ 1	global web
web-proxy	✓ 8	

Figure 5: The service health status dashboard for HashiCorp Consul.

A lightweight Service discovery client runs alongside a service instance, allowing the service to programmatically register and deregister with the central Service Registry. The system can also monitor a service's health status so operators can triage the availability of each instance.

With a Service discovery solution in place, services can be registered as part of the central Service Registry the moment they are deployed.

If service A needs to communicate with service B, it queries the Service Registry which returns the network location for all available instances of service B. Typically, this is done without code modification using DNS. More advanced use cases are enabled by using Consul's REST API.



```
$ dig redis.service.consul
10.1.2.32
```

```
service  {
  name  =   "redis"
  port  =   "8000"
  tags  =   ["global"]
}
```

Figure 6: HashiCorp Consul Service Discovery.

Applications can perform load balancing by randomly sending traffic to different instances. In this approach, the routing and the load balancing is fully distributed which allows the system to scale and support very large environments.

Within a single data center or cloud region, if one of the service instances becomes unhealthy or dies, the registry will avoid returning its address to trigger an automatic failover to other healthy instances. Across data centers, organizations can define centralized failover policies with Service discovery to automatically route traffic to available service instances running in different geo-locations or cloud regions, meaning they no longer have to hardcode this logic into applications or manage other failover appliances.

If there is a need for advanced load balancing features within an enterprise data center, like web application firewall etc., HashiCorp Consul's Service Registry can be used as a source of truth for automating load balancer configuration.

Consul's service registry provides a "subscription" service to automate these network operations either through the use of tools like Consul Template – which can render configuration files for any config-file driven application or load balancers such as HAProxy, NGINX,etc – or through native integrations into HashiCorp Consul's service discovery features available from F5, HAProxy, NGINX Plus, etc.

Networking teams now have the ability to completely automate their load balancing backend pools depending on which applications are registered and healthy within Consul's service catalog.

Any change to a service will trigger a new configuration getting generated and reloaded to the load balancer dynamically. Traffic can be routed to new service instances instantly without manual intervention. The same approach applies to firewalls and other critical network middleware as well. This publish/subscribe model accelerates the productivity of the network and IT operation teams by removing classical ticket-driven workflows.

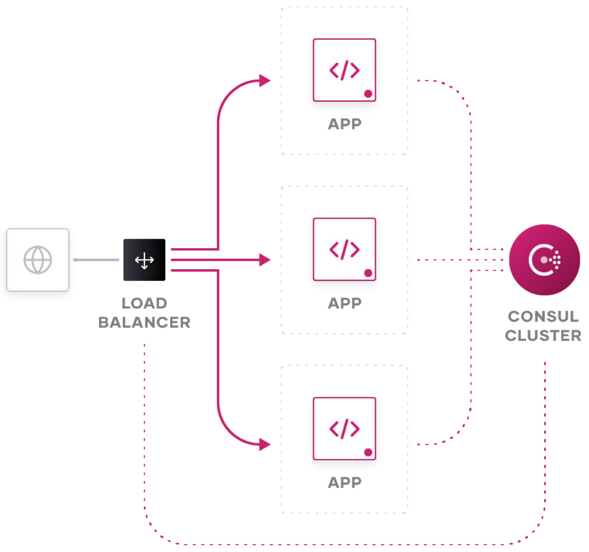


Figure 7: Load balancer integration in Consul's service catalog.

In order to keep pace with the needs of dynamic virtual machine and application deployments, many networking vendors brought Software Defined Networking (SDN) solutions to market. SDNs enable operators to automate network changes by virtualizing & orchestrating the deployment of network functions such as L2 overlays (VXLAN), load balancers, and L4 stateful firewalls.

Besides automating network deployments on-demand when a virtual machine was deployed, SDN technologies also helped to bring more robust network topologies into enterprises data centers as most of the SDN technologies leveraged VXLAN and virtualized L2 broadcast domains on top of a robust L3 leaf and spine networking fabric.

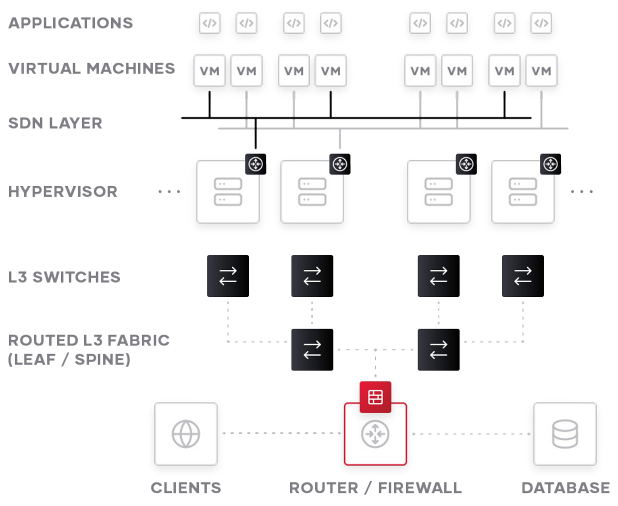


Figure 8: SDN layer on top of L3 network fabric topology.

The combination of automated virtual machines deployments in conjunction with an SDN technology satisfied enterprise networking needs for a time, until compute-based technologies evolved to the next level: Containerized applications and the related rise of container orchestration solutions like Kubernetes.

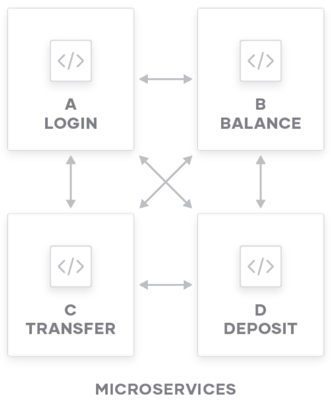


Figure 9: A microservices banking application.

Microservices offered a number of benefits over monolithic software architectures to developers, and the associated rise of container orchestration solutions brought many benefits to compute administrators such as better hardware utilization, faster deployment cycles, failure isolation, automated scheduling of applications, out of the box service discovery, etc. However, these changes brought new challenges for network administrators, especially pertaining to providing security for network connections originating from the container platform to applications running outside of the given platform.

Take Kubernetes as an example, depending on the used CNI plugin, it is possible that many different services residing on the same node leverage the node's physical IP address as their source IP address when connecting to services outside of the Kubernetes cluster.

Hence, it is very difficult (if not impossible) to identify which service is currently using this given IP address in a classical IP based network firewall.

This circumstance can lead to network teams modifying firewalls rules to permit all traffic sourced from Kubernetes nodes in the cluster.

This allows legitimate services to connect to the services residing on the "other side" of the IP based firewall, but also gives the exact same privileges to all other services on the same Kubernetes cluster, even though they may not be necessarily needed, hence imposing a potential security risk.

Also, this "over provisioning" of access rights within IP-based firewalls is an anti-pattern to a pure whitelisted, zero-trust networking security posture that is being adopted by many enterprises.

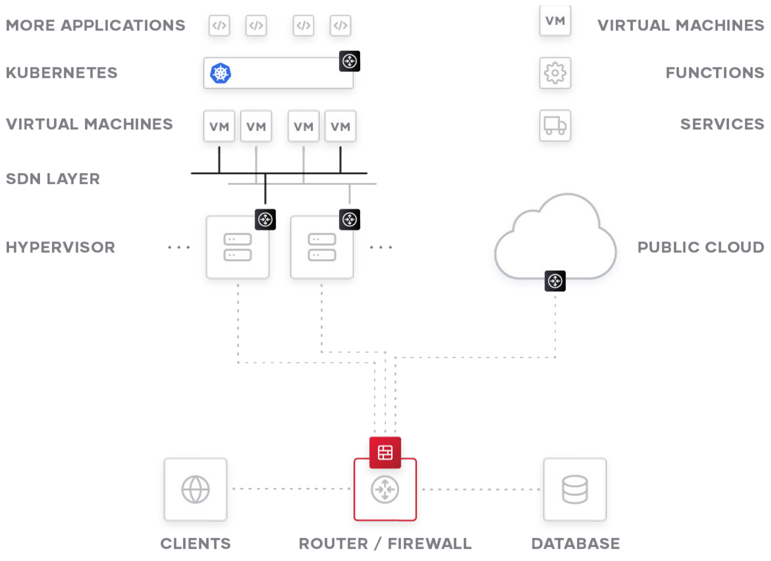


Figure 10: Kubernetes workloads connecting to outside workloads (here in a Public Cloud environment).

Another example could be public cloud based services: cloud services normally receive dynamically assigned IP addresses. Once a service instance stops, the IP address can be recycled and used for a different application. As a result, tracking cloud services and resources as well as keeping them up to date in a multi-cloud environment becomes much more difficult and complex.

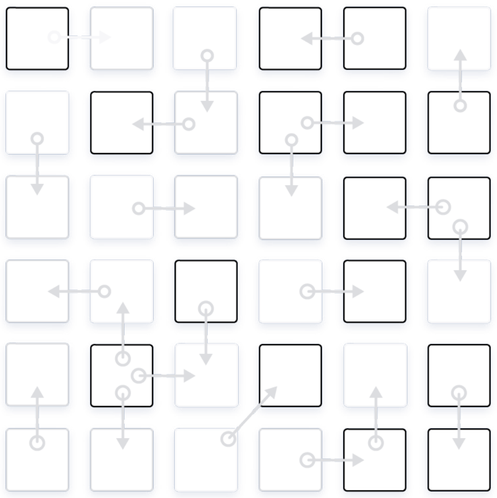


Figure 11: Communication patterns in a simple microservices application.

Service meshes do not identify and authorize a given service based on its source IP address within a network. Service meshes like Consul leverage a higher layer methodology to identify, differentiate and authorize services within the network even if they may use the exact same source IP address to initiate their connections.

In Consul service mesh the authentication and authorization of service-to-service communications is done by means of SPIFFE compliant X.509 certificates, which are issued to each service participating in the service mesh.

This approach provides a more fine-grained notion of identity compared to an IP-based identity model, giving network and firewall administrators greater control over network traffic.

Microservice architectures and cloud infrastructures give development teams the freedom to choose between different runtime platforms and cloud services that are best suited to their applications. However, this can lead to multiple islands of resources where the dynamic service networking is constrained within a single platform or a particular cloud. For example, a service deployed in a Kubernetes cluster cannot easily discover a database running on virtual machines outside of the Kubernetes environment using Kuberentes' built-in service discovery.

Consul service mesh is both runtime and cloud agnostic. This makes it possible to extend Consul to multiple Kubernetes clusters, and legacy workloads running on bare metal servers or virtual machines – regardless of whether the workloads reside on-premises or in different cloud environments.

In addition to authentication and authorization of services, Consul service mesh also offers encryption for service-to-service communication, load balancing, advanced layer 7 routing, telemetry, and more.

These features will be explained in detail in the following chapters.

Consul Introduction

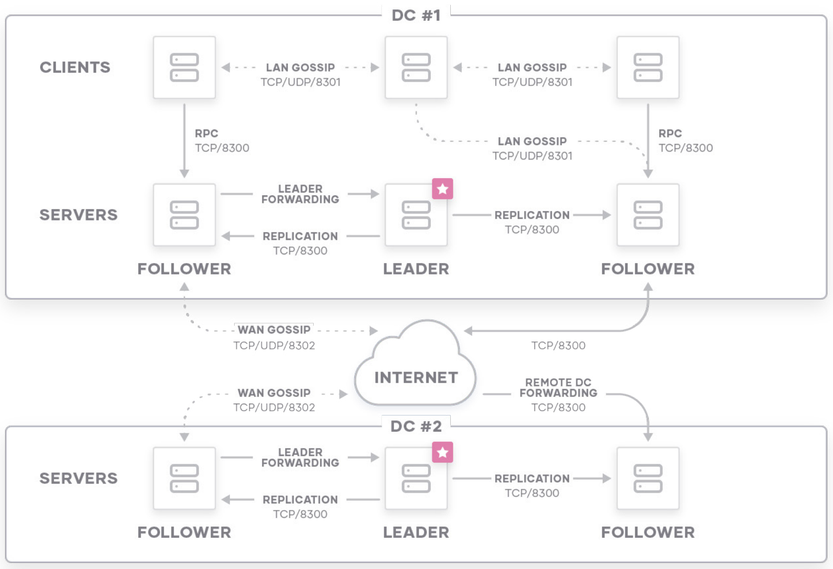
Consul is a distributed system that has many different moving parts. To help users and developers of Consul form a mental model of how it works, this section documents the system architecture.

Before describing the architecture, we recommend reading the glossary of terms on the consul.io website to help clarify what is being discussed.

The architecture concepts in this document can be used with the Reference Architecture guide when deploying Consul in production.

10,000 foot view

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



Let's break down this image and describe each piece. First, we can see that there are two data centers, labeled "one" and "two." Consul has first class support for multiple data centers and expects this to be a common deployment pattern (though, is not required).

Within each data center, we have a mixture of clients and servers. It is expected that there are 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 servers 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 agents in a data center participate in a gossip protocol, and are members of a common gossip pool. 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 take place such as leader election.

The servers in each data center 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. 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 data centers to discover each other in a low-touch manner. Bringing a new data center 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 resource in a different data center, it forwards the request to a random server in the target data center. That server may then forward the request to the local leader.

This results in a very low coupling between data centers. Failure detection, connection caching and multiplexing all allow cross-datacenter requests to be relatively fast and reliable.

In general, data is not replicated between different Consul Datacenters. When a request is made for a resource in another data center, the local Consul servers forward an RPC request to the remote Consul servers for that resource and return the results. If the remote data center is not available, then those resources will also not be available, but otherwise will not affect the local data center. There are some special situations where a limited subset of data can be replicated, such as with Consul's built-in ACL replication capability, or using external tools like consul-replicate.

Clients may cache certain 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 Service Mesh

This white paper focuses on the service mesh functionality of Consul. It will not provide any detailed content on the following topics:

- Consul basics and underlying protocols
- Setting up a production grade Consul Cluster

These topics are covered elsewhere in white papers and blog posts. It is assumed the reader already has a reasonable understanding of these topics.

The following link contains the source code for the two-tier application which will be referenced throughout this document.

https://github.com/hashicorp/demo-consul-101/tree/master/services

The sample application exposes a "Dashboard" service which communicates with and displays results from the backend "Counting" service.

The application will be discussed in a single Consul Datacenter environment, as well as spread across a Consul multi-datacenter federation.

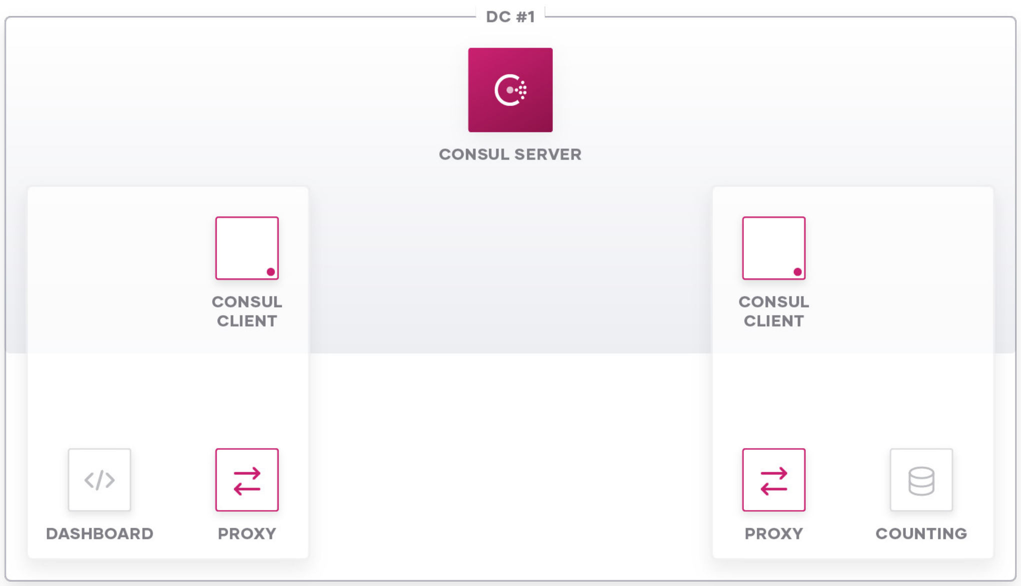


Figure 12: Dashboard—Counting Application setup in single Consul Datacenter.

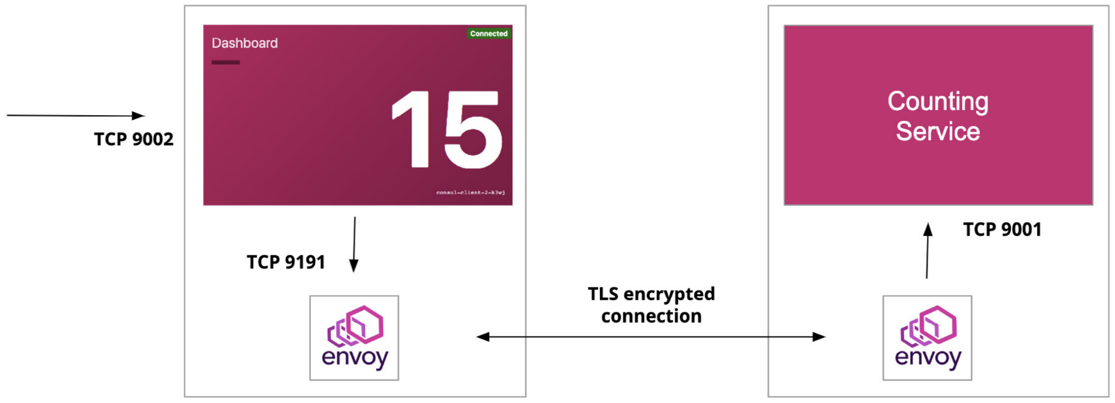
The service definition for the Dashboard service looks as follows:

```
{
 “service”: {
 “name”: “dashboard”,
 “port”: 9002,
 “connect”: {
  “sidecar_service”: {
  “proxy”: {
   “upstreams”: [
   {
    “destination_name”: “counting”,
    “local_bind_port”: 9191
   }
   ]
  }
  }
 }
 }
}
```

The service definitions for the Counting service looks as follows:

```
{
 “service”: {
 “name”: “counting”,
 “port”: 9001,
 “connect”: {
  “sidecar_service”: {}
 }
 }
}
```

The resulting application topology including the various TCP ports configured will look like this:



Preparing a Consul DC for Consul Service Mesh

To enable Consul's Service Mesh feature “Connect” (“Consul service mesh” and “Consul Connect” will be used interchangeably throughout this paper) in an existing Consul Datacenter the “connect” stanza needs to be added to the configuration of the Consul Servers:

```
“connect”: {
 “enabled”: true
 }
```

This stanza will automatically enable the Consul integrated CA (certificate authority) functionality which is used to sign certificates, which serve as the identities for the services registered in Consul.

Consul may be configured to utilize an external CA, such as HashiCorp Vault, to sign service certificates. This may be desirable to provide a separation of duties between Consul operators and security teams, or to integrate with existing CA infrastructure.

Consul can be used to implement a common service mesh across disparate cloud or runtime environments spanning multiple data centers. However, it is necessary to have a common root of trust between all services within a federated, multi-datacenter environment.

For this to work, the involved Consul Servers must agree on a primary Consul Datacenter, which acts as this root of trust from a CA perspective.

This needs to be configured in the Consul Server configuration of all federated Consul Datacenters:

```
primary_data center": "dc-gcp"
```

The Consul Clients should have the

```
enable_central_service_config
```

Flag set to true in their configuration file to allow for centralized configuration of service or proxy specific settings, which should apply to all instances.

Consul control-plane vs. data-plane

Consul service mesh provides a clear separation between the control-plane which determines configuration policies, and the data-plane which handles the actual packet forwarding.

This is a common model used by most SDN and service mesh solutions as it allows for central configuration of distributed components when the system configuration changes.

Consul's control-plane is distributed, and consists of both Consul server and client agents.

This distributed control-plane architecture allows for large scale and avoids single points of failure or choke points in the system overall.

The Consul Servers take care of collecting information about service health, service location, service availability and perform calculations like service resolution, certificate signing etc. They are also the central point of configuration for things like the Service Access Graph (Intentions) or Layer 7 traffic management features, which are then provided to the respective Consul clients.

Consul clients take care of tasks which are distributed across the control-plane such as the generation of key material for service certificates, health checking of services, and configuring sidecar proxy instances.

Consul's data-plane is pluggable, allowing users to choose the supported proxy which best meets their needs. Consul supports operating a mix of sidecar proxies in the data-plane. This may be desirable when operating across different runtimes, or to satisfy varying application requirements.

The following data-plane components are supported within Consul service mesh:

- **Consul Agent integrated proxy:** The Consul agent provides an integrated layer 4 proxy which can be used when first getting started with Consul, or for testing and development purposes.
- **Native application integration:** Consul service mesh offers the ability to integrate applications directly into the service mesh without the need for a sidecar proxy solution. This may be desirable in latency sensitive environments where the additional latency introduced by routing through the sidecar proxy is not acceptable. Native integration requires changes to application code, and may not be suitable for all applications.
- **HAProxy:** HAProxy can be leveraged as a Layer 4 sidecar proxy solution and uses a "Connector" application between the Consul Client Agent and the HAProxy Dataplane API to bootstrap and lifecycle the HAProxy sidecar.
- **Envoy:** Envoy can be leveraged for proxying both layer 4 and layer 7 traffic. Envoy is currently the only data-plane proxy which supports the layer 7 traffic management features of Consul service mesh.

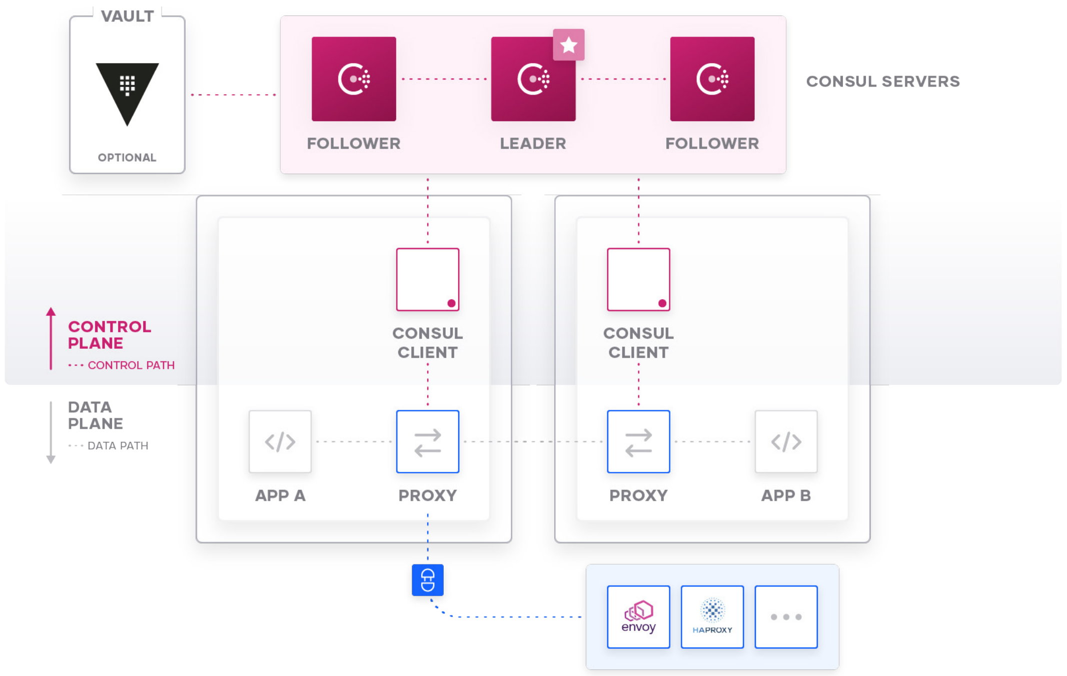


Figure 13: Control- and data-plane separation in Consul service mesh.

The remainder of this paper will focus on Envoy as the data-plane proxy as it fully supports Consul's layer 7 traffic management and Mesh Gateway functionality.

Service Registration

In order to allow a given service to participate in the service mesh, and to instruct Consul to prepare the associated configuration, at a minimum the following configuration must be added to the service definition.

```
“connect”: {
 “sidecar_service”: {}
}
```

This minimum configuration will allow a service to be targeted by other services within the service mesh, but it will not allow the service to talk to other upstream services as that has not yet been defined.

Sidecar Proxy Listeners

A sidecar proxy supports two types of listeners which will be discussed in this chapter.

At a high-level, there are two different traffic flows to consider that a sidecar proxy needs to manage; inbound traffic from a downstream application to the local application instance and outbound traffic from the local application instance to an upstream service.

For inbound traffic the high-level flow would be as follows:

- Sidecar proxy receives traffic from a remote proxy instance on its public listener
- Sidecar proxy forwards traffic to local application instance listener

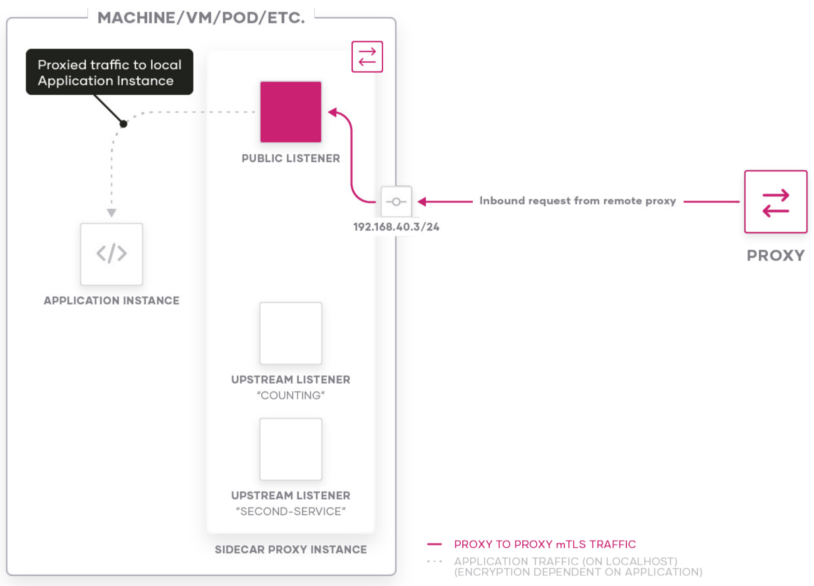
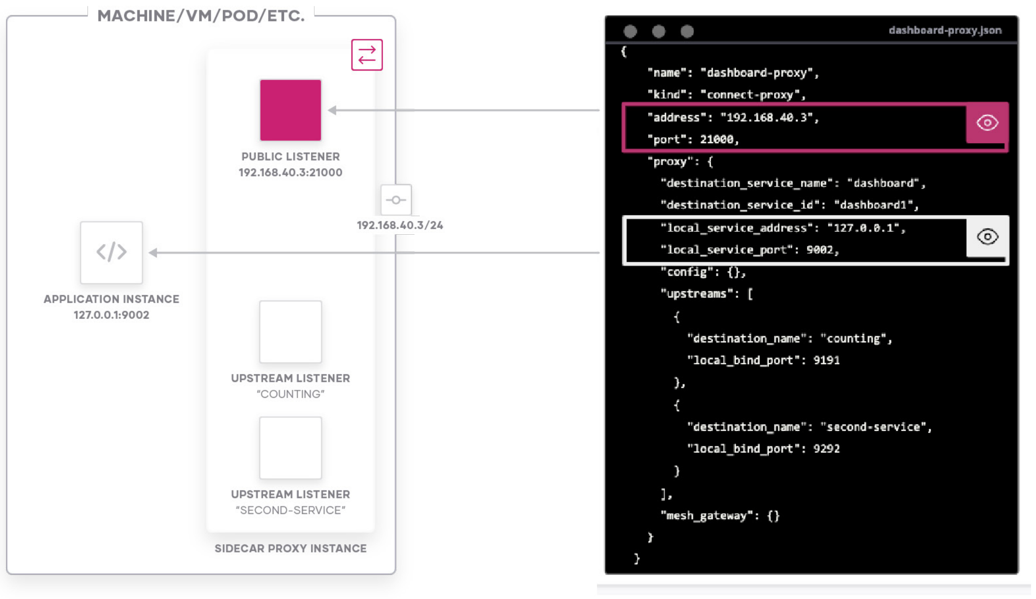

MACHINE/VM/POD/ETC.
Proxied traffic to local Application Instance
PUBLIC LISTENER
Inbound request from remote proxy
192.168.40.3/24
PROXY
APPLICATION INSTANCE
UPSTREAM LISTENER "COUNTING"
UPSTREAM LISTENER "SECOND-SERVICE"
SIDECAR PROXY INSTANCE
PROXY TO PROXY mTLS TRAFFIC
APPLICATION TRAFFIC (ON LOCALHOST) (ENCRYPTION DEPENDENT ON APPLICATION)

Figure 14: Service-to-service inbound traffic flow.

By default the sidecar proxy's public listener will use the host's/pod's IP address, but this can be customized.

Similarly, application instances are expected to listen on the 'localhost' address by default, but this too can be customized.



```
{
    "name": "dashboard-proxy",
    "kind": "connect-proxy",
    "address": "192.168.40.3",
    "port": 21000,
    "proxy": {
      "destination_service_name": "dashboard",
      "destination_service_id": "dashboard1",
      "local_service_address": "127.0.0.1",
      "local_service_port": 9002,
      "config": {},
      "upstreams": [
        {
          "destination_name": "counting",
          "local_bind_port": 9191
        },
        {
          "destination_name": "second-service",
          "local_bind_port": 9292
        }
      ],
      "mesh_gateway": {}
    }
  }
```

Figure 15: Mapping of "inbound" configuration.

If a local application instance needs to reach an upstream service, this must be explicitly configured within the respective service definition or the sidecar-proxy definition to allow the sidecar proxy to perform a mapping between an incoming TCP session and the upstream service.

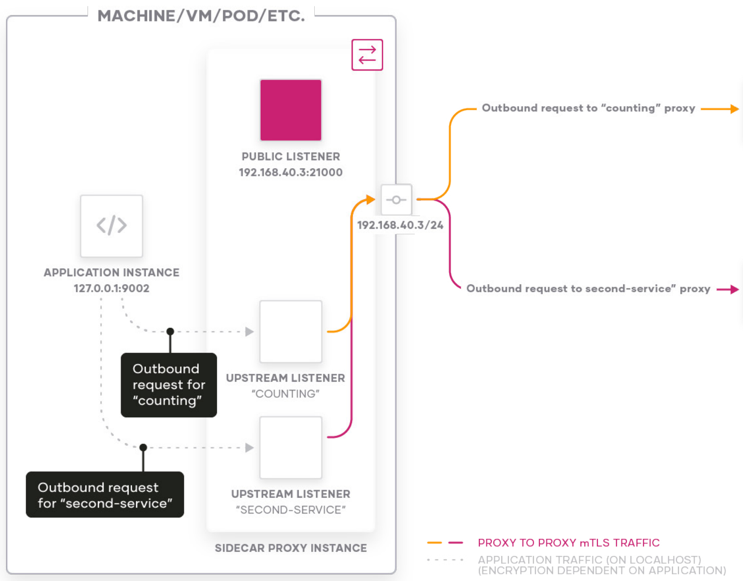
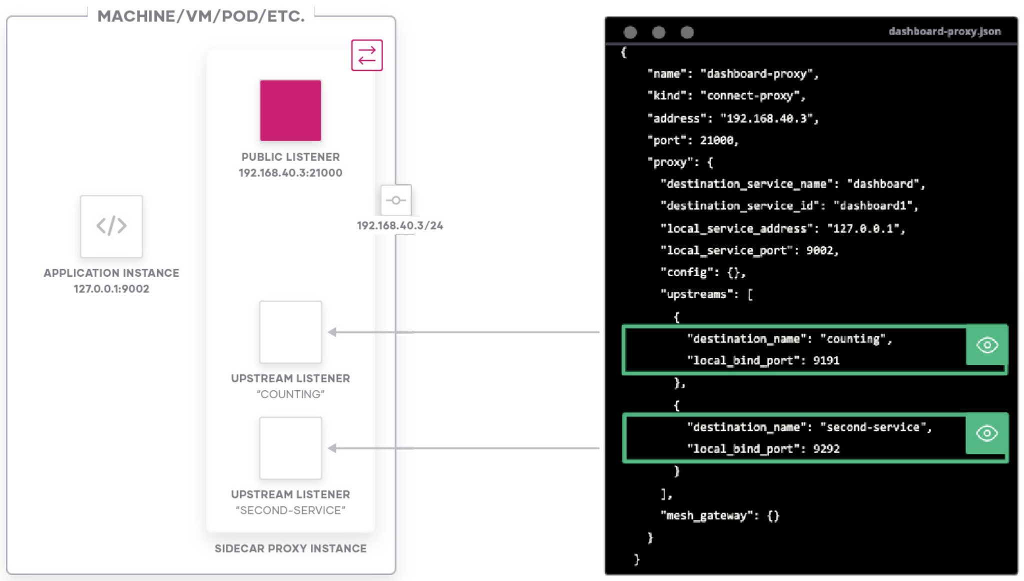

MACHINE/VM/POD/ETC.
PUBLIC LISTENER
192.168.40.3:21000
192.168.40.3/24
APPLICATION INSTANCE
127.0.0.1:9002
Outbound request for "counting"
Outbound request for "second-service"
UPSTREAM LISTENER
"COUNTING"
UPSTREAM LISTENER
"SECOND-SERVICE"
SIDECAR PROXY INSTANCE
Outbound request to "counting" proxy
PROXY
Outbound request to second-service" proxy
PROXY
PROXY TO PROXY mTLS TRAFFIC
APPLICATION TRAFFIC (ON LOCALHOST)
(ENCRYPTION DEPENDENT ON APPLICATION)

Figure 16: Service-to-service outbound traffic flow.

A sidecar proxy can have one or more listeners representing different upstream services. Depending on which listener the local application instance connects to, the sidecar proxy will forward the traffic to the appropriate upstream service.

If an application needs to connect to an upstream service, the destination must be declared in the service's upstream definition in order for the sidecar proxy to create the respective listeners.



```
{
    "name": "dashboard-proxy",
    "kind": "connect-proxy",
    "address": "192.168.40.3",
    "port": 21000,
    "proxy": {
      "destination_service_name": "dashboard",
      "destination_service_id": "dashboard1",
      "local_service_address": "127.0.0.1",
      "local_service_port": 9002,
      "config": {},
      "upstreams": [
        {
          "destination_name": "counting",
          "local_bind_port": 9191
        },
        {
          "destination_name": "second-service",
          "local_bind_port": 9292
        }
      ],
      "mesh_gateway": {}
    }
  }
```

Figure 17: Mapping of "outbound" configuration.

The example configuration in this chapter will create upstream listeners which can be used to transport connections for any TCP-based application within the service mesh such as an HTTP or database server.

For HTTP/gRPC type of applications, Consul service mesh offers a more comprehensive feature set, which allows headers or HTTP verbs to be used when deciding where to route traffic. These Layer 7 traffic management options will be covered in an upcoming chapter.

Envoy Sidecar Bootstrapping

In Consul service mesh, sidecar proxies are co-located with the services they represent, and proxy all inbound and outbound traffic for destinations within the service mesh.

The Consul Client Agent instantiates the sidecar proxy and registers it as a service.

The agent configures the sidecar proxy's local listeners and upstreams according to the registered service definition.

The high-level workflow of bootstrapping Envoy as a sidecar proxy for a service is as follows.

The operator initiates the bootstrap process by:

- Registering a service within Consul with a proxy definition
- Creating initial Envoy config with **`consul connect envoy`** command
- Start and supervise Envoy process

Tasks handled by the local Consul Client Agent:

- Request certificate for the service proxy
- Fetch relevant intentions from Consul Server and cache locally
- Configure Envoy through gRPC

Further maintenance tasks of Consul Client Agent during the lifetime of the sidecar proxy instance:

- Request a new certificate if required and update Envoy
- Keep track of new/updated intentions and cache locally
- Update Envoy cluster config if service endpoint status changes

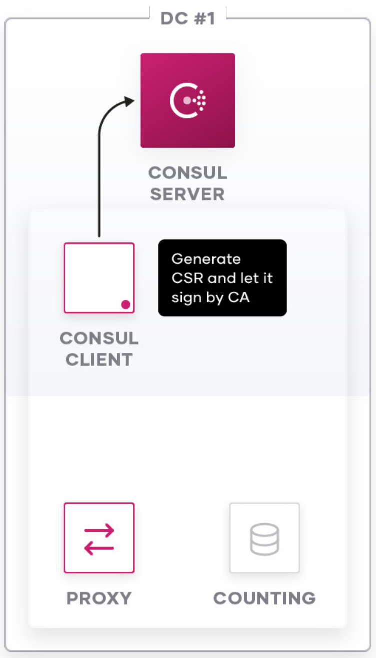
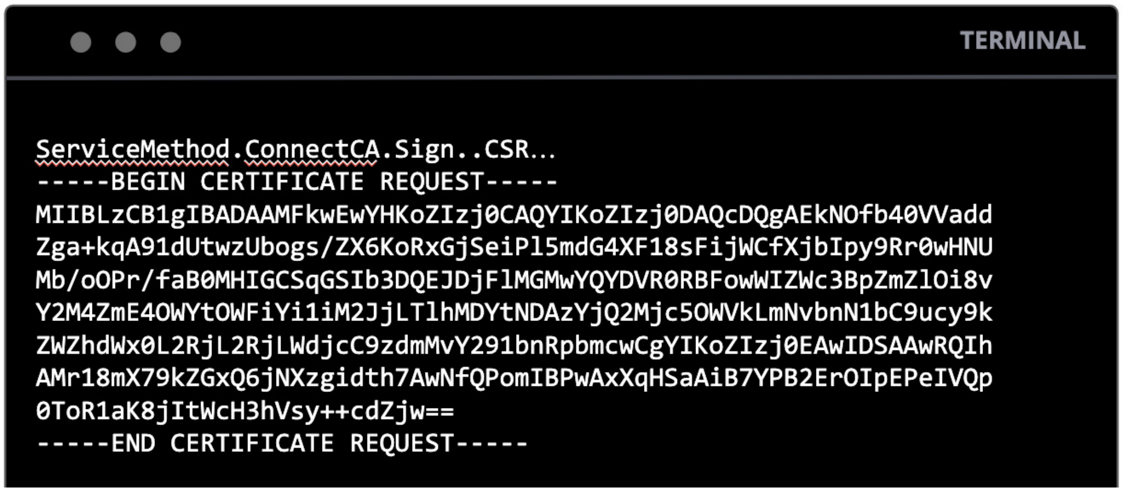


Figure 18: Certificate signing request in Consul service mesh

The following is an example service definition which registers a sidecar proxy instance for the "dashboard" service and also defines the "counting" service as an upstream service to be exposed locally on TCP port 9191:

```
“service”: {
 “name”: “dashboard”,
 “port”: 9002,
 “connect”: {
  “sidecar_service”: {
  “proxy”: {
   “upstreams”: [
   { “destination_name”: “counting”,
    “local_bind_port”: 9191 }
   ]
  }}
 }}
```

When the service definition, including its respective proxy definition, is registered in Consul, the local Consul Client Agent generates a CSR for the service identity and sends the CSR to the Consul Servers to be signed. The Consul Servers can either sign this CSR directly using the built-in CA or defer to a third party CA, like HashiCorp Vault, depending on the Consul Server configuration. The Consul Client Agent will receive a SPIFFE compatible X.509 certificate for this service from the Consul Servers as a result of this process:



```
ServiceMethod.ConnectCA.Sign..CSR...
-----BEGIN CERTIFICATE REQUEST-----
MIIBLzCB1gIBADAAMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEkNOfb40VVadd
Zga+kqA91dUtwzUbogs/ZX6KoRxGjSeiPl5mdG4XF18sFijWCfXjbIpy9Rr0wHNU
Mb/oOPr/faB0MHIGCSqGSIb3DQEJDjFlMGMwYQYDVR0RBFowWIZWc3BpZmZlOi8v
Y2M4ZmE4OWYtOWFiYi1iM2JjLTlhMDYtNDAzYjQ2Mjc5OWVkLmNvbnN1bC9ucy9k
ZWZhdWx0L2RjL2RjLWdjcC9zdmMvY291bnRpbmcwCgYIKoZIzj0EAwIDSAAwRQIh
AMr18mX79kZGxQ6jNXzgidth7AwNfQPomIBPwAxXqHSaAiB7YPB2ErOIpEPeIVQp
0ToR1aK8jItWcH3hVsy++cdZjw==
-----END CERTIFICATE REQUEST-----
```

Figure 19: Raw packet capture of certificate signing request (CSR) from Consul Client to Consul Server.

The Consul Server will then send the corresponding service certificate back to the Consul Client over the existing RPC channel:

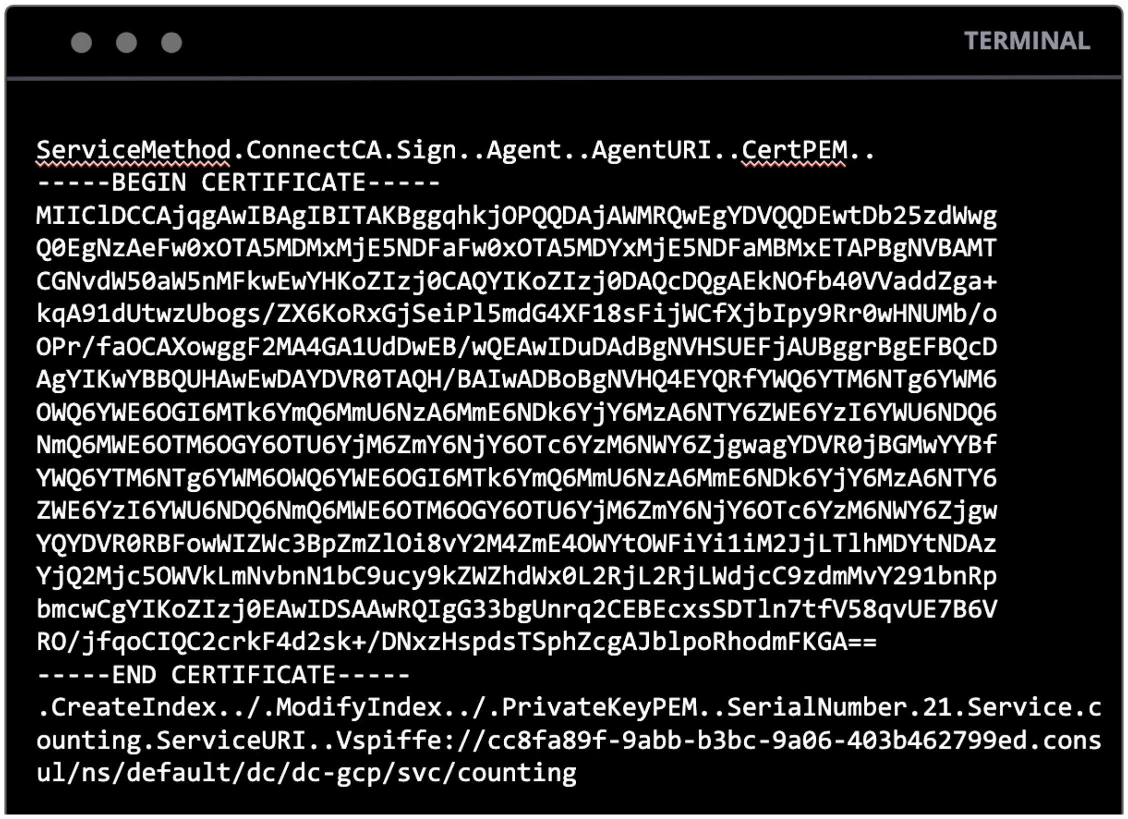
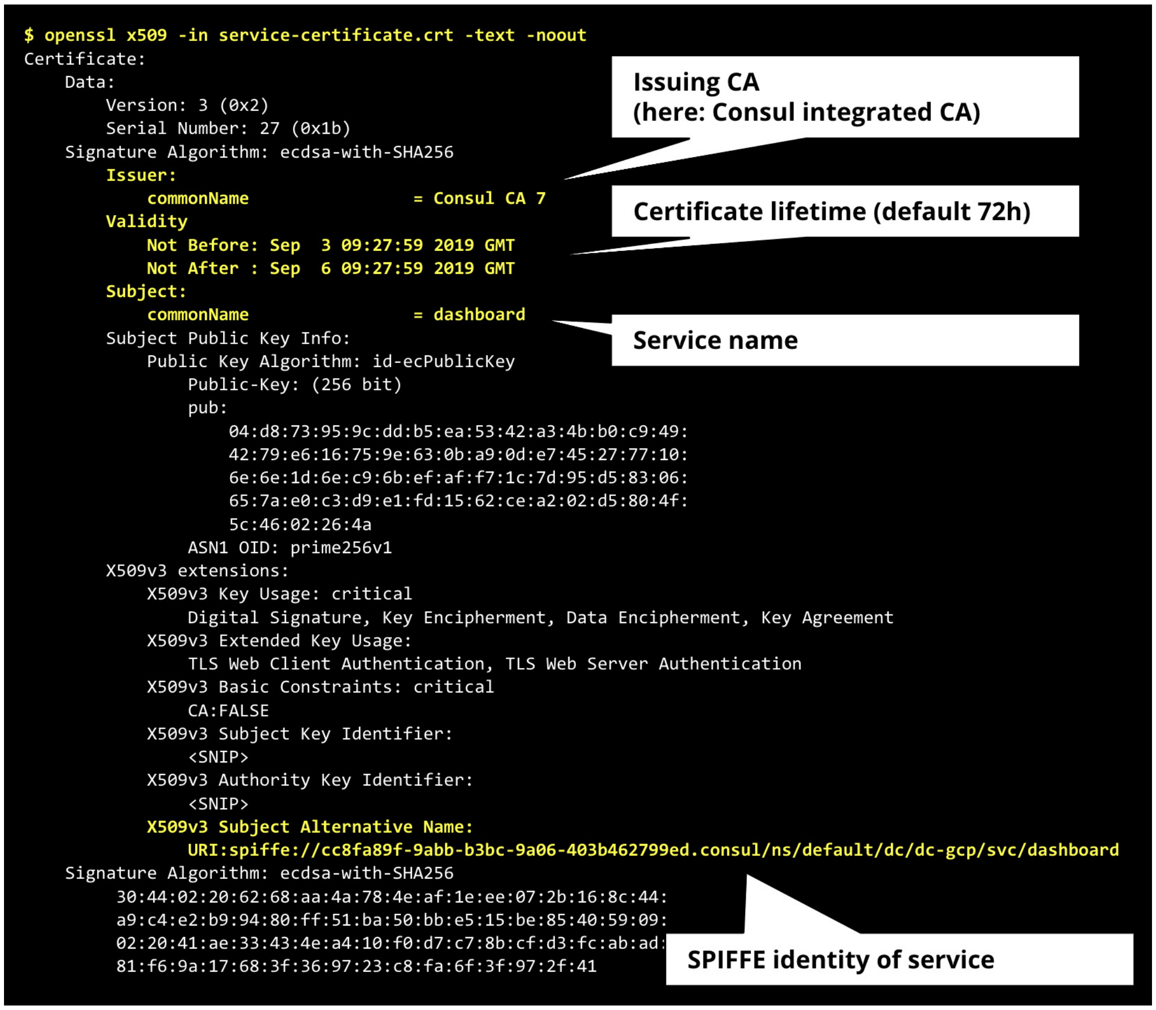

```
ServiceMethod.ConnectCA.Sign..Agent..AgentURI..CertPEM..
-----BEGIN CERTIFICATE-----
MIIClDCCAjqgAwIBAgIBITAKBggqhkjOPQQDAjAWMRQwEgYDVQQDEwtDb25zdWwg
Q0EgNzAeFw0xOTA5MDMxMjE5NDFaFw0xOTA5MDYxMjE5NDFaMBMxETAPBgNVBAMT
CGNvdW50aW5nMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEkNOfb40VVaddZga+
kqA91dUtwzUbogs/ZX6KoRxGjSeiPl5mdG4XF18sFijWCfXjbIpy9Rr0wHNUMb/o
OPr/faOCAXowggF2MA4GA1UdDwEB/wQEAwIDuDAdBgNVHSUEFjAUBggrBgEFBQcD
AgYIKwYBBQUHAwEwDAYDVR0TAQH/BAIwADBoBgNVHQ4EYQRfYWQ6YTM6NTg6YWM6
OWQ6YWE6OGI6MTk6YmQ6MmU6NzA6MmE6NDk6YjY6MzA6NTY6ZWE6YzI6YWU6NDQ6
NmQ6MWE6OTM6OGY6OTU6YjM6ZmY6NjY6OTc6YzM6NWY6ZjgwagYDVR0jBGMwYYBf
YWQ6YTM6NTg6YWM6OWQ6YWE6OGI6MTk6YmQ6MmU6NzA6MmE6NDk6YjY6MzA6NTY6
ZWE6YzI6YWU6NDQ6NmQ6MWE6OTM6OGY6OTU6YjM6ZmY6NjY6OTc6YzM6NWY6Zjgw
YQYDVR0RBFowWIZWc3BpZmZlOi8vY2M4ZmE4OWYtOWFiYi1iM2JjLTlhMDYtNDAz
YjQ2Mjc5OWVkLmNvbnN1bC9ucy9kZWZhdWx0L2RjL2RjLWdjcC9zdmMvY291bnRp
bmcwCgYIKoZIzj0EAwIDSAAwRQIgG33bgUnrq2CEBEcxsSDTln7tfV58qvUE7B6V
RO/jfqoCIQC2crkF4d2sk+/DNxzHspdsTSphZcgAJblpoRhodmFKGA==
-----END CERTIFICATE-----
.CreateIndex../.ModifyIndex../.PrivateKeyPEM..SerialNumber.21.Service.c
ounting.ServiceURI..Vspiffe://cc8fa89f-9abb-b3bc-9a06-403b462799ed.cons
ul/ns/default/dc/dc-gcp/svc/counting
```

Figure 20: Raw packet capture of signed certificate sent from Consul Server back to Consul Client.

As a result of this step, the Consul Client Agent has a SPIFFE compatible X.509 certificate in its cache, which will be installed directly into the Envoy data-plane later on.

The service certificate generated by the Consul Client Agent looks as follows and has the service's name as the "commonName."

A SPIFFE-compliant identity of the service is encoded as a "SAN" (Subject Alternative Name) which additionally contains the Consul Namespace in which the service resides (default namespace in this example):



```
$ openssl x509 -in service-certificate.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 27 (0x1b)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer:
            commonName                = Consul CA 7
        Validity
            Not Before: Sep  3 09:27:59 2019 GMT
            Not After : Sep  6 09:27:59 2019 GMT
        Subject:
            commonName                = dashboard
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:d8:73:95:9c:dd:b5:ea:53:42:a3:4b:b0:c9:49:
                    42:79:e6:16:75:9e:63:0b:a9:0d:e7:45:27:77:10:
                    6e:6e:1d:6e:c9:6b:ef:af:f7:1c:7d:95:d5:83:06:
                    65:7a:e0:c3:d9:e1:fd:15:62:ce:a2:02:d5:80:4f:
                    5c:46:02:26:4a
                ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment, Data Encipherment, Key Agreement
            X509v3 Extended Key Usage:
                TLS Web Client Authentication, TLS Web Server Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                <SNIP>
            X509v3 Authority Key Identifier:
                <SNIP>
            X509v3 Subject Alternative Name:
                URI:spiffe://cc8fa89f-9abb-b3bc-9a06-403b462799ed.consul/ns/default/dc/dc-gcp/svc/dashboard
    Signature Algorithm: ecdsa-with-SHA256
          30:44:02:20:62:68:aa:4a:78:4e:af:1e:ee:07:2b:16:8c:44:
          a9:c4:e2:b9:94:80:ff:51:ba:50:bb:e5:15:be:85:40:59:09:
          02:20:41:ae:33:43:4e:a4:10:f0:d7:c7:8b:cf:d3:fc:ab:ad:
          81:f6:9a:17:68:3f:36:97:23:c8:fa:6f:3f:97:2f:41
```

Issuing CA
(here: Consul integrated CA)
Certificate lifetime (default 72h)
Service name
SPIFFE identity of service

Figure 21: Decoded X.509 certificate for "dashboard" service.

The Consul agent will handle renewing and rotating the certificates during the lifetime of the Envoy sidecar proxy.

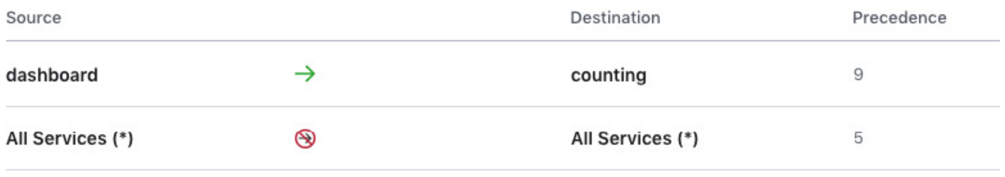
The Consul Client Agent will generate a new CSR for the service identity and will request a new certificate from Consul's configured CA after 75% of the lifetime of the currently installed certificate has elapsed.

This new service certificate is automatically updated by the Consul Client Agent through the existing gRPC connection into Envoy's running configuration.

The Consul Client Agent will also request a copy of all relevant intentions for this service and cache them locally.

These intentions will be used when authorizing service-to-service communication. The local cache avoids the need to callback to central servers for AuthZ requests, which may be a bottleneck in large scale service mesh deployments.

A cached intention may be an intention that references the service explicitly, or an intention referencing a service implicitly such as a wildcard source or destination:



Source		Destination	Precedence
dashboard	→	counting	9
All Services (*)	⊘	All Services (*)	5

Figure 22: View of configured intentions in Consul UI.

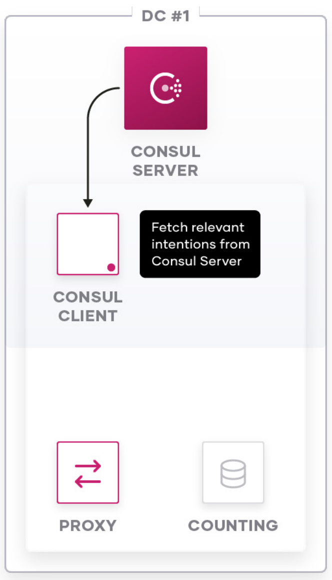
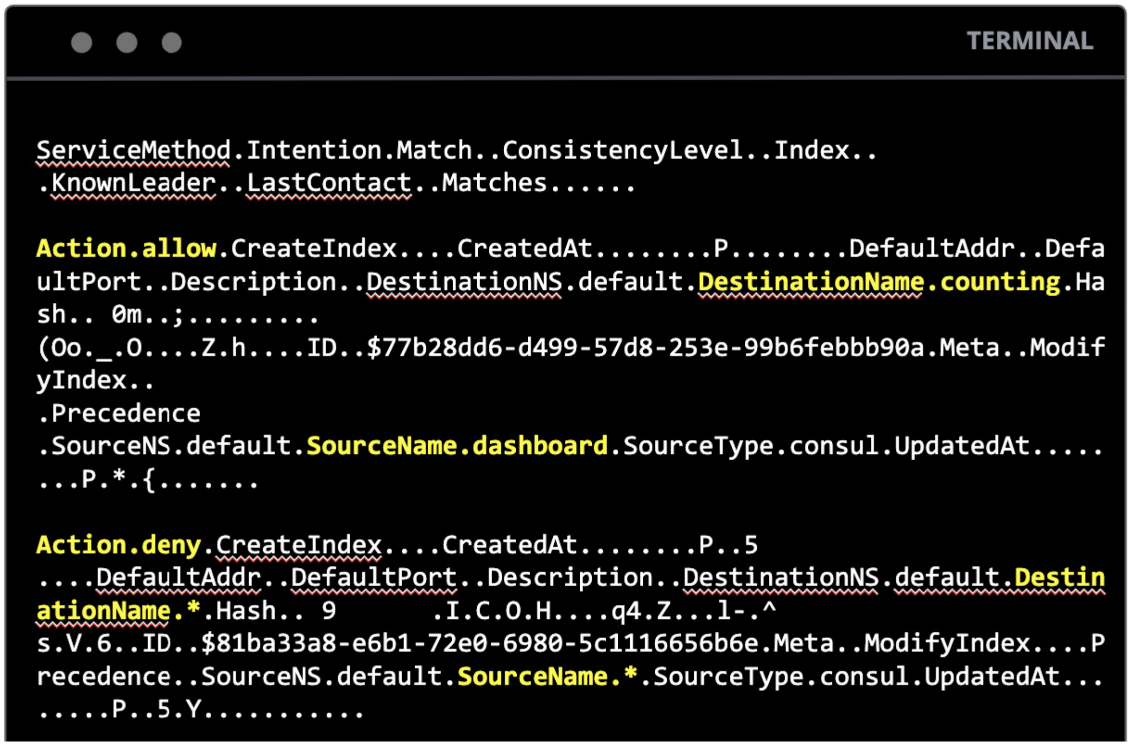


Figure 23: Consul Client receives relevant intentions from Consul Server.



```
ServiceMethod.Intention.Match..ConsistencyLevel..Index..
.KnownLeader..LastContact..Matches......

Action.allow.CreateIndex....CreatedAt........P........DefaultAddr..Defa
ultPort..Description..DestinationNS.default.DestinationName.counting.Ha
sh.. 0m..;.........
(Oo._.O....Z.h....ID..$77b28dd6-d499-57d8-253e-99b6febbb90a.Meta..Modif
yIndex..
.Precedence
.SourceNS.default.SourceName.dashboard.SourceType.consul.UpdatedAt.....
...P.*.{.......

Action.deny.CreateIndex....CreatedAt........P..5
....DefaultAddr..DefaultPort..Description..DestinationNS.default.Destin
ationName.*.Hash.. 9      .I.C.O.H....q4.Z...l-.^
s.V.6..ID..$81ba33a8-e6b1-72e0-6980-5c1116656b6e.Meta..ModifyIndex....P
recedence..SourceNS.default.SourceName.*.SourceType.consul.UpdatedAt...
.....P..5.Y..........
```

Figure 24: Raw packet capture of relevant intentions received by Consul Client.

The Consul agent will also fetch the relevant endpoints for the upstream services configured in the service definition.

In summary, the Consul Client Agent performs the following tasks when a service is registered:

- CSR generation and certificate signing for the service
- Loading intentions relevant for the service
- Loading upstream Service discovery results relevant for the service

The process of fetching all this relevant data for a sidecar proxy and spinning up a sidecar proxy instance is asynchronous.

To start Envoy as a sidecar proxy instance, it is first necessary to generate a bootstrap configuration for Envoy.

The **`consul connect envoy`** command can be used to generate this Envoy bootstrap configuration. The command can either directly start Envoy with the configuration, or it can write the bootstrap configuration to disk so that Envoy can be started separately with that configuration, for example using Docker.

The bootstrap config is populated with connection details for the local agent, such as the agent's gRPC port.

After starting Envoy with this bootstrap configuration, Envoy will initiate a connection to the local Consul Client agent's gRPC port (TCP 8502, by default), to receive the remainder of its configuration.

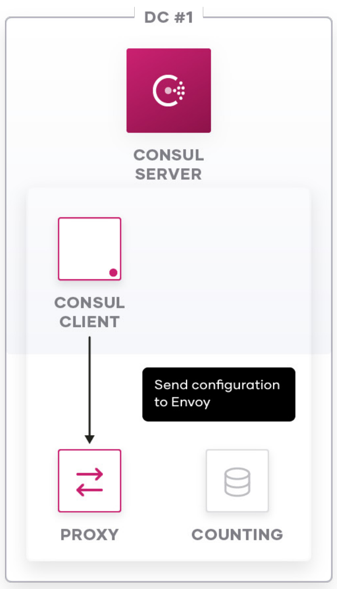
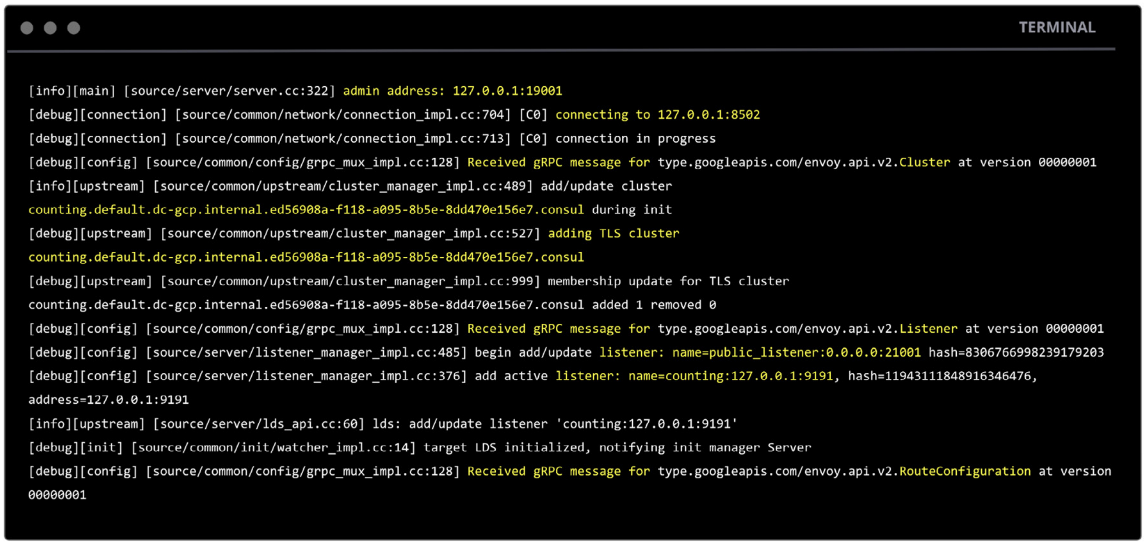


Figure 25: Consul Client configuring Envoy sidecar through local gRPC connection.

This Envoy config bundle sent from the Consul Client Agent to the Envoy sidecar proxy includes:

- Signed X.509 certificate for the service identity including private key
- Consul CA root certificate
- Listener configuration for
  - "public_listener" for inbound communication
  - "upstream" listeners on localhost for outbound communication (if one or more upstreams are configured in service definition)
- Cluster configuration for upstream services including currently available upstream service endpoints (if upstream is configured in service definition)
- L7 Route configurations

The following is a log output from an Envoy sidecar proxy during bootstrapping and highlights the most important configuration pieces received through the gRPC connection from the Consul Client Agent discussed during this chapter:



```
[info][main] [source/server/server.cc:322] admin address: 127.0.0.1:19001
[debug][connection] [source/common/network/connection_impl.cc:704] [C0] connecting to 127.0.0.1:8502
[debug][connection] [source/common/network/connection_impl.cc:713] [C0] connection in progress
[debug][config] [source/common/config/grpc_mux_impl.cc:128] Received gRPC message for type.googleapis.com/envoy.api.v2.Cluster at version 00000001
[info][upstream] [source/common/upstream/cluster_manager_impl.cc:489] add/update cluster
counting.default.dc-gcp.internal.ed56908a-f118-a095-8b5e-8dd470e156e7.consul during init
[debug][upstream] [source/common/upstream/cluster_manager_impl.cc:527] adding TLS cluster
counting.default.dc-gcp.internal.ed56908a-f118-a095-8b5e-8dd470e156e7.consul
[debug][upstream] [source/common/upstream/cluster_manager_impl.cc:999] membership update for TLS cluster
counting.default.dc-gcp.internal.ed56908a-f118-a095-8b5e-8dd470e156e7.consul added 1 removed 0
[debug][config] [source/common/config/grpc_mux_impl.cc:128] Received gRPC message for type.googleapis.com/envoy.api.v2.Listener at version 00000001
[debug][config] [source/server/listener_manager_impl.cc:485] begin add/update listener: name=public_listener:0.0.0.0:21001 hash=8306766998239179203
[debug][config] [source/server/listener_manager_impl.cc:376] add active listener: name=counting:127.0.0.1:9191, hash=11943111848916346476,
address=127.0.0.1:9191
[info][upstream] [source/server/lds_api.cc:60] lds: add/update listener 'counting:127.0.0.1:9191'
[debug][init] [source/common/init/watcher_impl.cc:14] target LDS initialized, notifying init manager Server
[debug][config] [source/common/config/grpc_mux_impl.cc:128] Received gRPC message for type.googleapis.com/envoy.api.v2.RouteConfiguration at version
00000001
```

Figure 26: Envoy log output during bootstrapping.

The gRPC connection between the running Envoy sidecar instance and the Consul Client Agent will be kept open during the lifetime of the Envoy sidecar proxy instance to allow for near real-time updates of Envoy's data-plane configuration.

Those changes could be additions or removals of service endpoints to a given cluster, new Discovery Chain results due to new L7 configurations (discussed in the next chapter), rotation of a service certificate, etc.

Consul L7 Traffic Management

The configurations introduced in the previous sections enable Consul service mesh to securely route, authenticate, authorize, and encrypt traffic between healthy service endpoints within the service mesh.

If an application calls an upstream listener on the sidecar proxy to connect to "Service X", the service mesh will take care of forwarding this request to an healthy endpoint of "Service X."

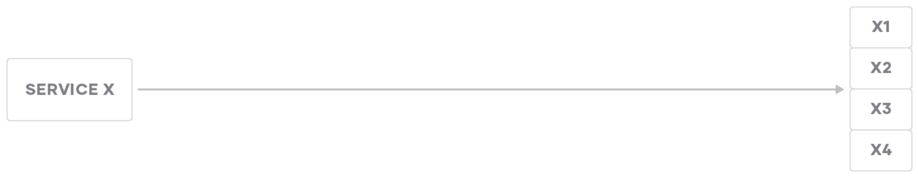


Figure 27: Service endpoint resolution without Layer 7 Traffic Management.

In addition to this, Consul service mesh introduces the concept of a "Discovery Chain" which provides multiple stages for controlling L7 traffic routing and endpoint resolution for HTTP or gRPC services.

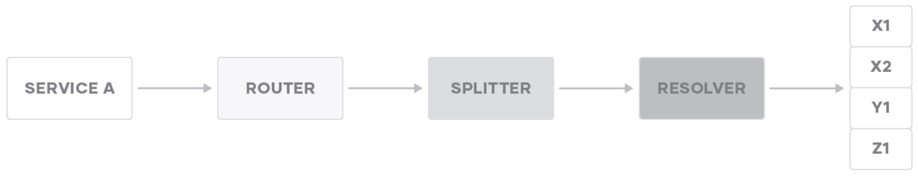


Figure 28: Service endpoint resolution with Layer 7 Traffic Management.

NOTE: These advanced Layer 7 Traffic Management options are currently available for Envoy sidecar proxies only.

The introduced "Discovery Chain" consists of three steps:

- service-router
- service-splitter
- service-resolver

Consul's Discovery Chain is a pure control-plane construct, meaning that configuration will be pre-computed within Consul and the results will be pushed down to Envoy's data-plane in the appropriate configuration sections.

One does not need to explicitly configure every Discovery Chain component. Every phase in the discovery chain has sane defaults implemented in Consul.

A given service within Consul can have any combination of these Discovery Chain components configured depending on the desired routing behavior.

Below is a subset of possible configuration options for a given service.

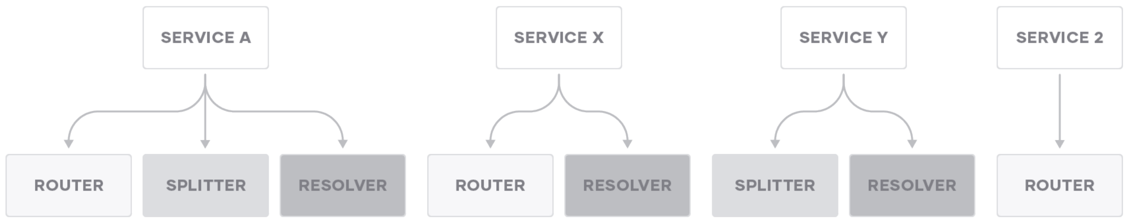


Figure 29: Possible configuration options for the service discovery chain.

If a configuration entry is not explicitly configured for a service, the Discovery Chain stages have the following default behavior:

- Default service-router passes all routes to the same service
- Default service-splitter passes 100% of the traffic to the same service
- Default service-resolver selects all healthy instances from the local Consul Datacenter

The discovery chain allows referencing other services, for example when using a service-router to reroute traffic to an alternate service.

In this case, there is a clear rule within Consul's Discovery Chain: Endpoint discovery can only move forward or laterally in the Discovery Chain, not backwards.

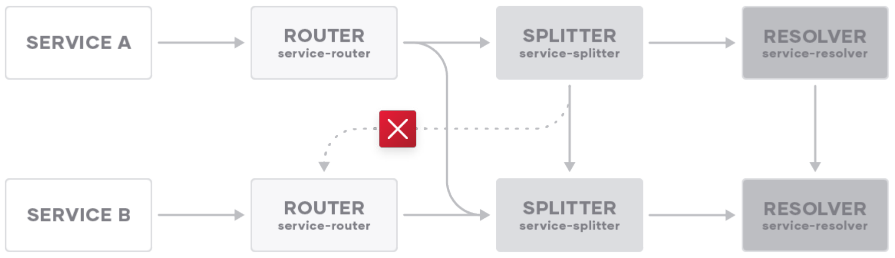


Figure 30: Discovery Chain flow.

For example, when re-routing traffic to a different service at a service-router, this means that request handling resumes at the service-splitter of the next service or—if there is no service-splitter config—all requests pass through to the service-resolver for that service.

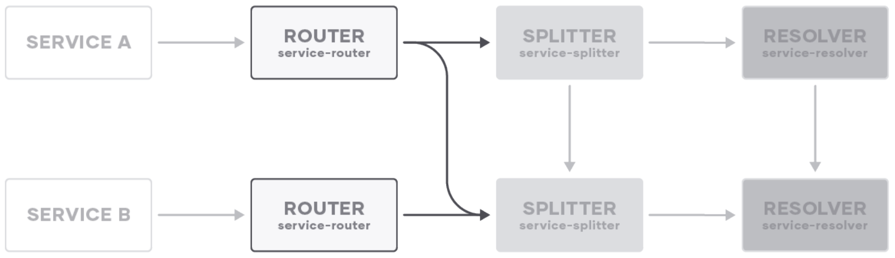


Figure 31: Possible Discovery Chain references of a service-router.

If a service-splitter targets another service for one or more segments of traffic, request processing continues from that service's splitter.

Splitters are not allowed to form cyclic redirects. Consul will not allow such configurations and will return an error message if the operator tries to configure a cyclic redirect.

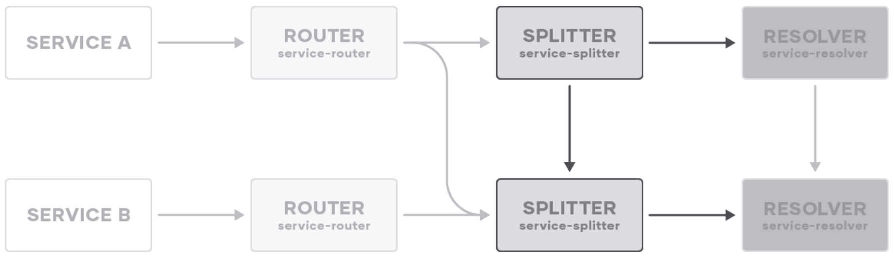


Figure 32: Possible Discovery Chain references of a service-splitter.

As the last step in the Discovery Chain, service-resolvers may redirect or failover to another service.

Request handling resumes at the target service's resolver and respects its failover configuration. Service-resolvers are also not allowed to form cyclic redirects.

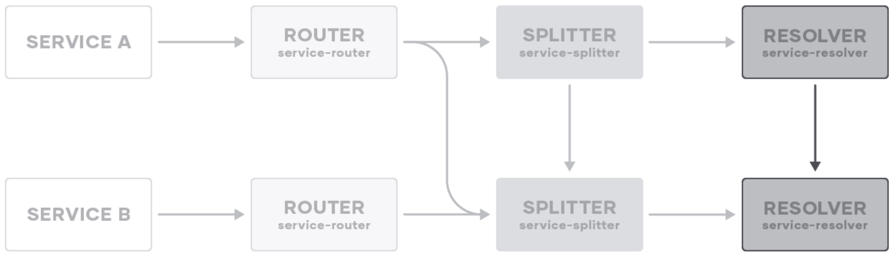


Figure 33: Possible Discovery Chain references of a service-resolver.

The results of this Discovery Chain will be pre-calculated in Consul and pushed into Envoy's data-plane configuration on proxy startup, so as to avoid the need for additional lookups once traffic starts flowing through the sidecar proxy.

service-defaults

With the exception of the service-resolver, the remaining Discovery Chain phases are only supported for services registered in Consul as using the "http", "http2", or "grpc" protocols.

By default services are identified as utilizing the "tcp" protocol. The default protocol for a given service can be set using the "service-defaults" configuration type:

```
# Setting "service-defaults" for "counting" service

$ consul config write -<<EOF
kind = "service-defaults"
name = "counting"
protocol = "http"
EOF
```

It is possible to retrieve the current "service-defaults" config entries through Consul CLI (or API):

```
# Read "service-defaults" for "counting" service

$ consul config read -kind=service-defaults -name=counting
{
 "Kind": "service-defaults",
 "Name": "counting",
 "Protocol": "http",
...
}
```

When a "service-defaults" entry is not explicitly configured for a service, the service's protocol defaults to TCP and an error will occur when attempting to read the "service-defaults" configuration:

```
# Response if "service-defaults" are not explicitly configured for a service

$ consul config read -kind=service-defaults -name=service-name
Error reading config entry "service-defaults" / "service-name": Unexpected
response code: 404 (Config entry not found for "service-defaults" /
"service-name")
```

service-router

A service-router enables you to re-route traffic to a different service than requested by the downstream service based on HTTP path, headers, verbs, etc.

It also offers the ability to manipulate HTTP paths and configure request retry logic.

A service-router can be helpful if an enterprise is transitioning towards a microservice architecture.

Think of a simple two-tier application where a frontend application needs to talk to a monolithic backend application which offers various different APIs. If this backend application is disassembled into various microservices, each serving a different API path for the frontend application, this would normally require a reconfiguration of the frontend application to instruct the frontend which path is now being served by which microservice.

A service-router helps enable this transition while removing the need to reconfigure the frontend application, as Consul service mesh will take care of routing API requests to the correct microservice.

A service-router can also be helpful for blue-green deployment scenarios, where traffic is routed differently through the service mesh depending on which HTTP headers are present in the request, allowing for different traffic treatment based on the source e.g. for special "test groups."

A service-router can have multiple different L7 routes configured, each matching on different criteria and sending traffic to different services.

Every service-router within Consul service mesh has an implicit "catch-all" route which sends unmatched traffic to the originally requested service.

The following example service-router matches HTTP requests containing a specific path prefix, and redirects the traffic to a different service while also re-writing the HTTP path.

```
$ consul config write -<<EOF
kind = “service-router”
name = “counting”
routes = [
 {
 match {
  http {
  path_prefix = “/admin”
  }
 }

 destination {
  service = “counting-admin”,
  prefix_rewrite = “/”
 }
 }
]
EOF
```

If a downstream service calls the “counting” upstream service, and requests the /admin HTTP path, this service-router will re-route those requests to a different microservice called “counting- admin.”

As there are no other explicit routes configured within this service-router, all remaining unmatched traffic will be sent to the “counting” service by the implicit “catch-all” route.

Other examples for service-routers in Consul would be:

Match all traffic that uses the HTTP PUT method:

```
...
match {
  http {
  methods = “PUT”
  }
 }
…
```

Match all traffic that uses HTTP PUT and matches HTTP path_prefix /admin:

```
...
match {
  http {
  path_prefix = “/admin”
  methods = “PUT”
  }
 }
...
```

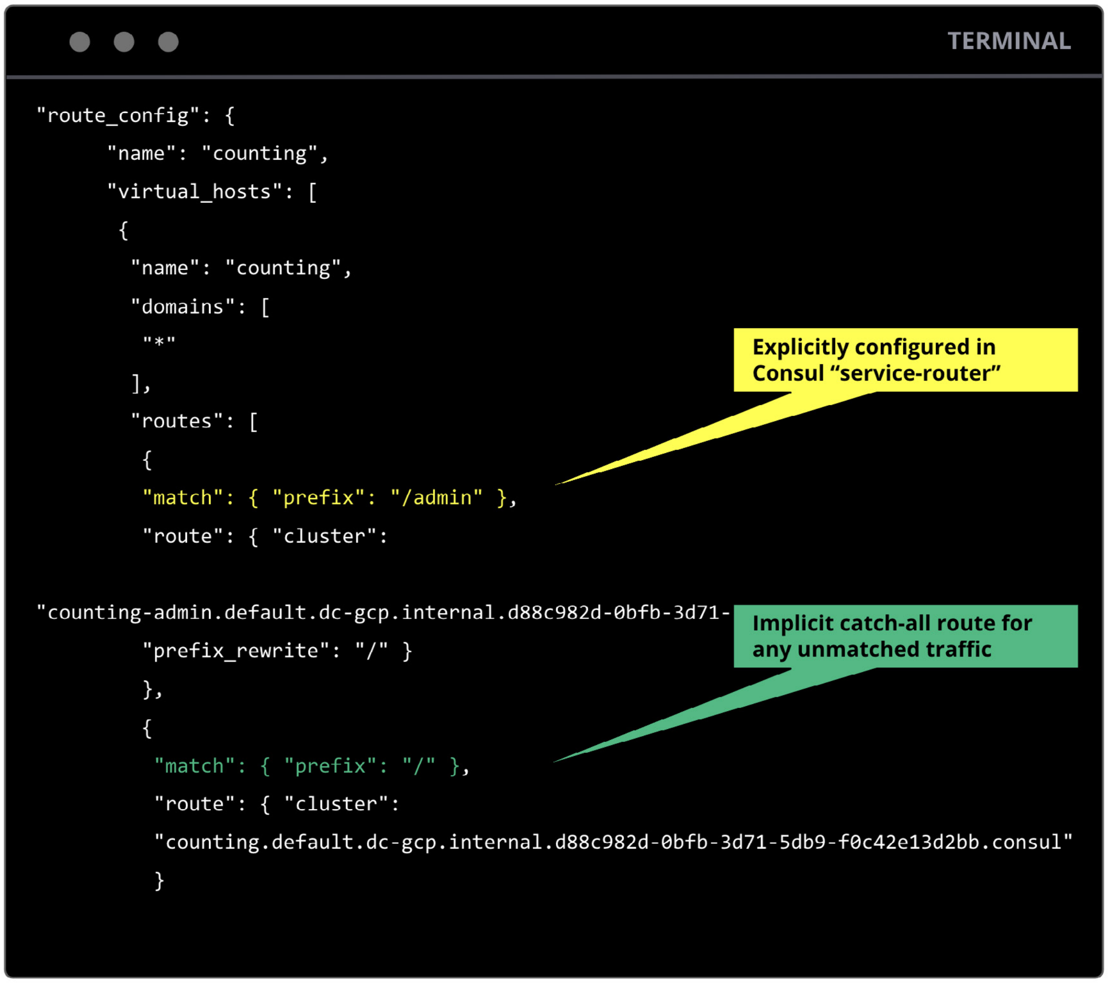
The “match” operation within a service-router is a logical “AND” operation.

The resulting data-plane constructs configured in a service-router will be pre-calculated in Consul and pushed down into the Envoy data-plane through the gRPC channel between the Consul Client Agent and the Envoy sidecar proxy instance.

Each configured L7 route will be an Envoy route within the data-plane with a separate Envoy cluster attached.

The service endpoint resolution will be performed within Consul and pushed down to the Envoy data-plane.

The example service-router configuration which routes traffic matching the HTTP path_prefix /admin to a service named "counting-admin", while sending the remainder of the traffic to the "counting" service, will look like this within the resulting Envoy data-plane configuration:

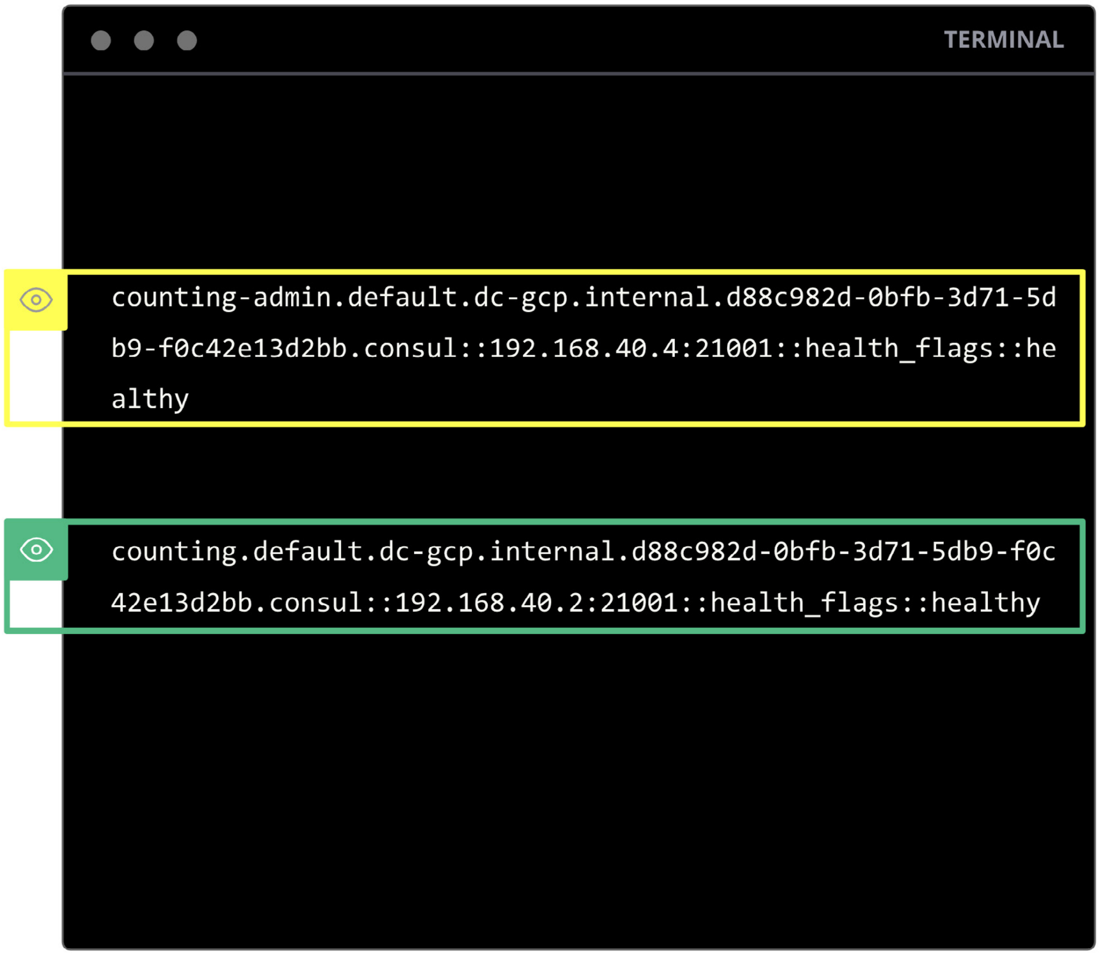

TERMINAL
Explicitly configured in Consul "service-router"
Implicit catch-all route for any unmatched traffic

```
"route_config": {
      "name": "counting",
      "virtual_hosts": [
       {
        "name": "counting",
        "domains": [
         "*"
        ],
        "routes": [
         {
         "match": { "prefix": "/admin" },
         "route": { "cluster":

"counting-admin.default.dc-gcp.internal.d88c982d-0bfb-3d71-
         "prefix_rewrite": "/" }
         },
         {
          "match": { "prefix": "/" },
          "route": { "cluster":
          "counting.default.dc-gcp.internal.d88c982d-0bfb-3d71-5db9-f0c42e13d2bb.consul"
          }
```

Figure 34: Resulting Envoy data-plane dynamic route configuration.

Each route points to a corresponding cluster which contains the currently available and healthy endpoints for a given service:

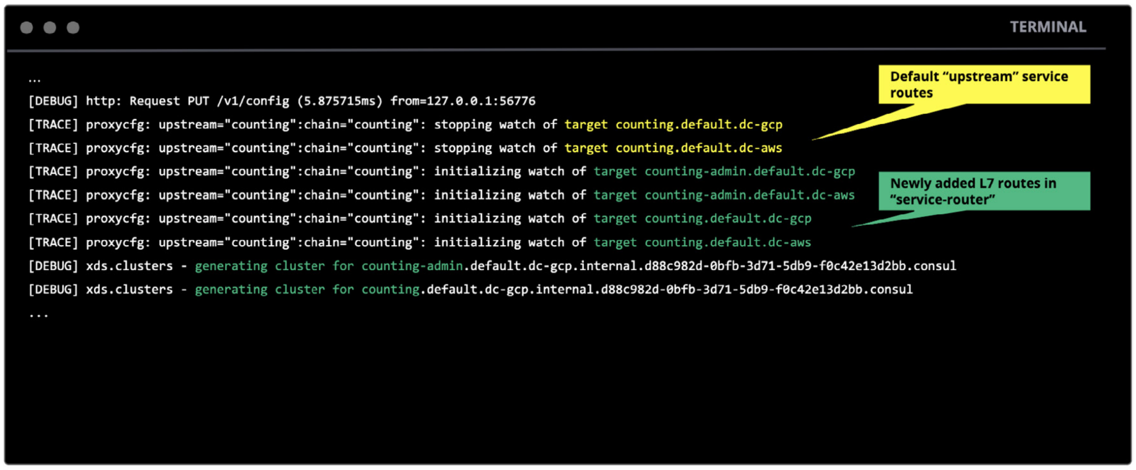


```
counting-admin.default.dc-gcp.internal.d88c982d-0bfb-3d71-5d
b9-f0c42e13d2bb.consul::192.168.40.4:21001::health_flags::he
althy

counting.default.dc-gcp.internal.d88c982d-0bfb-3d71-5db9-f0c
42e13d2bb.consul::192.168.40.2:21001::health_flags::healthy
```

Figure 35: Associated Envoy EDS cluster endpoints.

The following Envoy log outputs show the new data-plane constructs pushed down to Envoy after configuring the above service-router:


TERMINAL
Default "upstream" service routes
Newly added L7 routes in "service-router"

```
...
[DEBUG] http: Request PUT /v1/config (5.875715ms) from=127.0.0.1:56776
[TRACE] proxycfg: upstream="counting":chain="counting": stopping watch of target counting.default.dc-gcp
[TRACE] proxycfg: upstream="counting":chain="counting": stopping watch of target counting.default.dc-aws
[TRACE] proxycfg: upstream="counting":chain="counting": initializing watch of target counting-admin.default.dc-gcp
[TRACE] proxycfg: upstream="counting":chain="counting": initializing watch of target counting-admin.default.dc-aws
[TRACE] proxycfg: upstream="counting":chain="counting": initializing watch of target counting.default.dc-gcp
[TRACE] proxycfg: upstream="counting":chain="counting": initializing watch of target counting.default.dc-aws
[DEBUG] xds.clusters - generating cluster for counting-admin.default.dc-gcp.internal.d88c982d-0bfb-3d71-5db9-f0c42e13d2bb.consul
[DEBUG] xds.clusters - generating cluster for counting.default.dc-gcp.internal.d88c982d-0bfb-3d71-5db9-f0c42e13d2bb.consul
...
```

Figure 36: Envoy log output during service-router configuration.

If a "service-router" is not explicitly configured or is configured with no routes then the system behaves as if a "service-router" were configured to send all traffic to a service of the same name.

service-splitter

A service-splitter allows splitting traffic between subsets of a given service or between completely different services.

This can be useful for Canary deployments where a percentage of traffic should be sent to a new version of a given service for testing purposes, while the remainder of the traffic should continue to be sent to the previous known-good version.

The weighting configuration within a service-splitter is expressed as a percentage and can be configured in increments of 0.01, giving a 1/10000 requests granularity.

The sum of all defined weights needs to equal 100, otherwise Consul will not accept the configuration and will reply with an error message.

```
# Write “service-splitter” for “counting-admin” service

$ consul config write -<<EOF
kind = “service-splitter”
name = “counting-admin”
splits = [
 {
 weight = 80
 service_subset = “v1”
 },
 {
 weight = 20
 service_subset = “v2”
 },
]
EOF
```

The above service-splitter configuration will split traffic for the “counting-admin” service between two subsets named “v1” and “v2.”. The service_subsets will be defined in a service-resolver, which will be explained in the next chapter.

It is also possible to split traffic to a completely different service, by referencing it within the splits configuration:

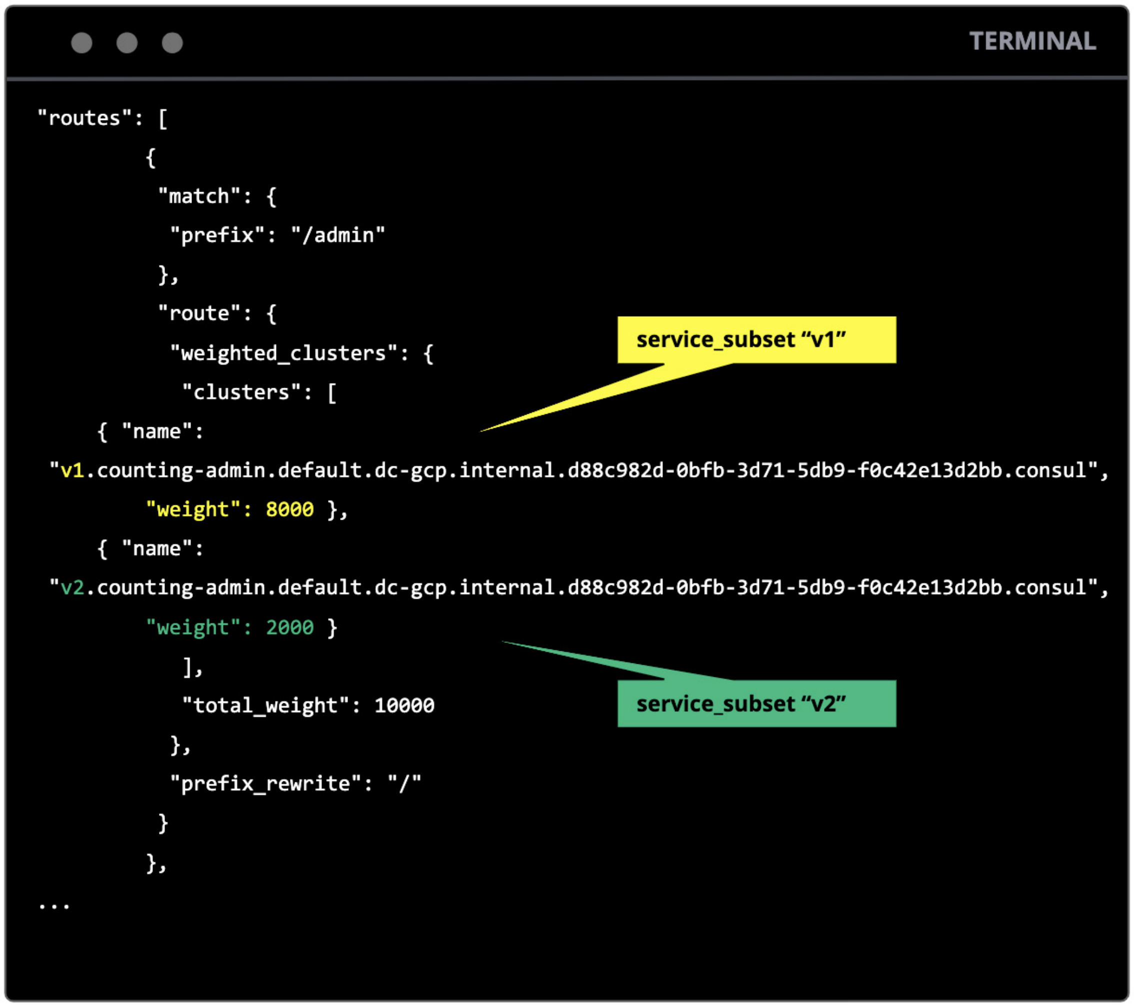
```
service = “some-other-service”
```

The resulting data-plane constructs configured within a service-splitter will be pre-calculated in Consul and pushed down into the Envoy data-plane through the gRPC channel between the Consul Client Agent and the Envoy sidecar proxy instance.

Each configured split will be an Envoy "weighted cluster" within the data-plane with a separate Envoy cluster attached.

The service's endpoints will be resolved within Consul and pushed down to the Envoy data-plane.

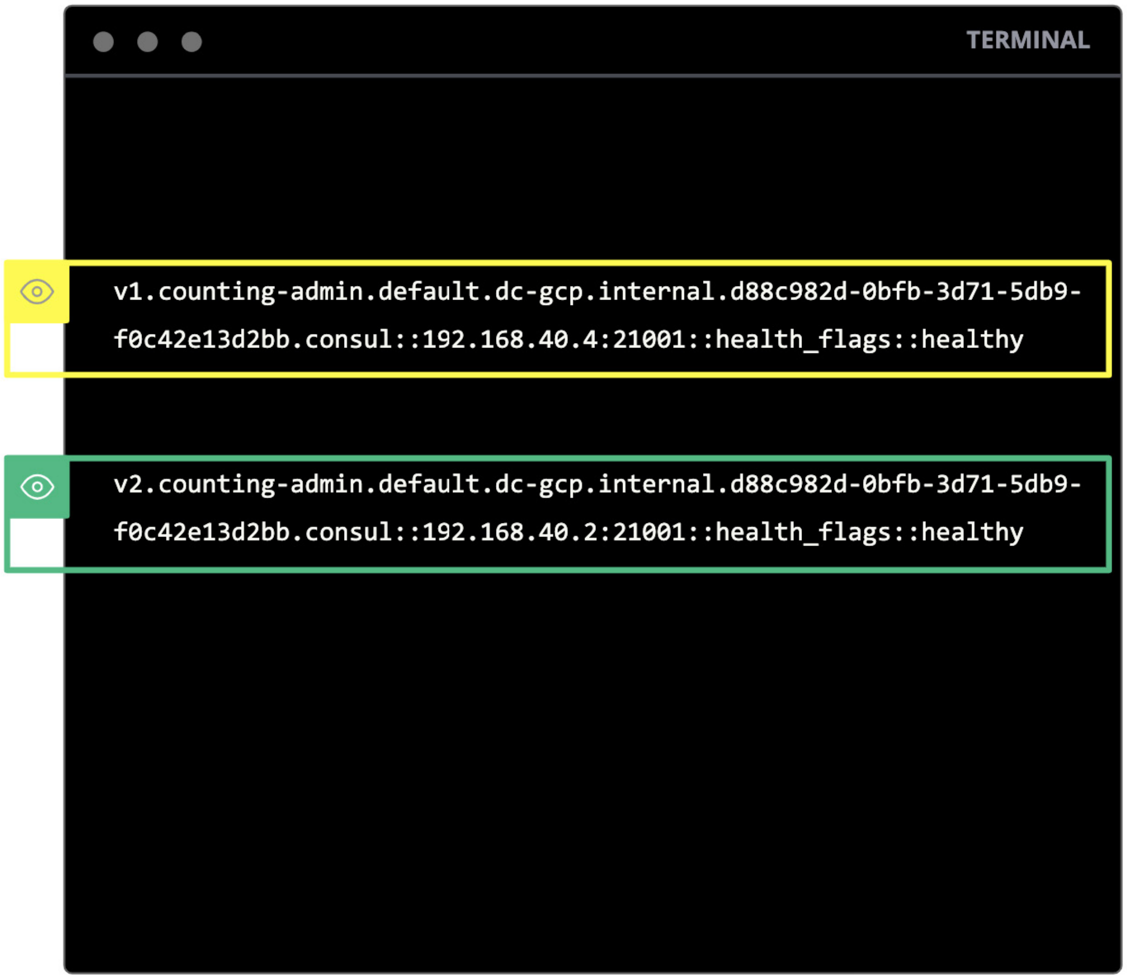
The example service-splitter configuration ends up in two Envoy weighted clusters for the "/admin" route (configured in the previous service-router example) and will look like this within the corresponding Envoy data-plane configuration:



```
"routes": [
         {
          "match": {
           "prefix": "/admin"
          },
          "route": {
           "weighted_clusters": {
            "clusters": [
    { "name":
 "v1.counting-admin.default.dc-gcp.internal.d88c982d-0bfb-3d71-5db9-f0c42e13d2bb.consul",
         "weight": 8000 },
    { "name":
 "v2.counting-admin.default.dc-gcp.internal.d88c982d-0bfb-3d71-5db9-f0c42e13d2bb.consul",
         "weight": 2000 }
            ],
            "total_weight": 10000
           },
           "prefix_rewrite": "/"
          }
         },
...
```

Figure 37: Resulting Envoy data-plane dynamic route configuration with weighted clusters.

The corresponding cluster for each service_subset contains the currently available and healthy endpoints for that subset:

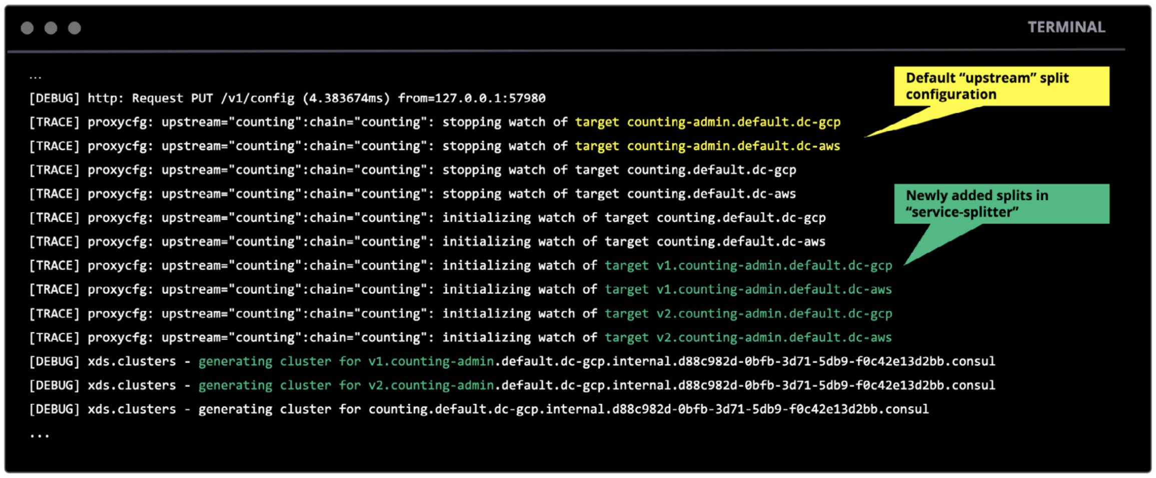


```
v1.counting-admin.default.dc-gcp.internal.d88c982d-0bfb-3d71-5db9-
f0c42e13d2bb.consul::192.168.40.4:21001::health_flags::healthy

v2.counting-admin.default.dc-gcp.internal.d88c982d-0bfb-3d71-5db9-
f0c42e13d2bb.consul::192.168.40.2:21001::health_flags::healthy
```

Figure 38: Associated Envoy EDS cluster endpoints for configured service_subsets.

The following Envoy log outputs show the new data-plane constructs pushed down to Envoy after configuring the above service-splitter:


TERMINAL
Default "upstream" split configuration
Newly added splits in "service-splitter"

```
...
[DEBUG] http: Request PUT /v1/config (4.383674ms) from=127.0.0.1:57980
[TRACE] proxycfg: upstream="counting":chain="counting": stopping watch of target counting-admin.default.dc-gcp
[TRACE] proxycfg: upstream="counting":chain="counting": stopping watch of target counting-admin.default.dc-aws
[TRACE] proxycfg: upstream="counting":chain="counting": stopping watch of target counting.default.dc-gcp
[TRACE] proxycfg: upstream="counting":chain="counting": stopping watch of target counting.default.dc-aws
[TRACE] proxycfg: upstream="counting":chain="counting": initializing watch of target counting.default.dc-gcp
[TRACE] proxycfg: upstream="counting":chain="counting": initializing watch of target counting.default.dc-aws
[TRACE] proxycfg: upstream="counting":chain="counting": initializing watch of target v1.counting-admin.default.dc-gcp
[TRACE] proxycfg: upstream="counting":chain="counting": initializing watch of target v1.counting-admin.default.dc-aws
[TRACE] proxycfg: upstream="counting":chain="counting": initializing watch of target v2.counting-admin.default.dc-gcp
[TRACE] proxycfg: upstream="counting":chain="counting": initializing watch of target v2.counting-admin.default.dc-aws
[DEBUG] xds.clusters - generating cluster for v1.counting-admin.default.dc-gcp.internal.d88c982d-0bfb-3d71-5db9-f0c42e13d2bb.consul
[DEBUG] xds.clusters - generating cluster for v2.counting-admin.default.dc-gcp.internal.d88c982d-0bfb-3d71-5db9-f0c42e13d2bb.consul
[DEBUG] xds.clusters - generating cluster for counting.default.dc-gcp.internal.d88c982d-0bfb-3d71-5db9-f0c42e13d2bb.consul
...
```

Figure 39: Envoy log output during service-splitter configuration.

If no "service-splitter" config is defined for a service, it is assumed 100% of traffic flows to a service with the same name and discovery continues on to the resolution stage.

service-resolver

A service-resolver provides the ability to define rules for steering traffic to a different service, service subset, or data center.

A service-resolver can also define rules for failover, redirecting traffic to a healthy instance in a different data center only when local instances are unhealthy.

Finally, a service-resolver can define service subsets—named filters that match a subset of instances—that can be used as route, split, redirect and failover targets (as used in the service-splitter configuration in the previous section).

A service-resolver configuration cannot be mapped directly to a data-plane construct in Envoy as was possible for service-router and service-splitter.

Service-resolver results will be used to populate Envoy clusters according to service-router and service-splitter configuration.

The following service-resolver example defines subsets for the "counting-admin" service:

```
# Write "service-resolver" for "counting-admin" service

$ consul config write -<<EOF
kind = "service-resolver"
name = "counting-admin"
subsets = {
 "v1" = {
 filter = "Service.Meta.version == v1"
 }
 "v2" = {
 filter = "Service.Meta.version == v2"
 }
}
failover = {
 "*" = {
 data centers = ["dc-aws", "dc-gcp"]
 }
}
EOF
```

This example defines the subsets for "v1" and "v2" based on filtering for metadata attached to the service endpoints which specify the current version of the service.

The filtering mechanism in a service-resolver allows for a variety of filtering options, not only on meta-data, but tags attached to a service endpoint, or the node on which a service is running, and many more.

Additionally, the failover behavior for all defined service subsets (by using * as a wildcard for all subsets) is defined in this service-resolver configuration which specifies the order of DCs to failover to if the local instances are unhealthy. The failover behavior can also be configured explicitly for each service subset.

Another useful example of leveraging a service-resolver configuration would be to expose services in the local Consul Datacenter, but redirect them to a given service in a remote federated Consul Datacenter:

```
kind = “service-resolver”
name = “web-dc2”
redirect {
 service = “web”
 data center = “dc2”
}
```

This example will redirect all traffic destined for service “web-dc2” in the local Consul Datacenter to the service “web” in the remote Consul Datacenter “dc2.”

If no “service-resolver” config is defined, the Discovery Chain assumes 100% of traffic goes to the healthy instances of the default service in the current data center+namespace and discovery terminates.

Consul Discovery Chain example

In this example, we will use the previous configuration examples for service-router, service-splitter and service-resolver to visualize how Discovery Chain results in Consul are computed.

Two important reminders on the following diagrams are:

- Discovery Chain results will be pre-computed and results will be installed directly in the Envoy data-plane
- Discovery Chain resolution does not happen on a per-request or per-session basis

Example 1: Dashboard—Counting /anypath

If the Dashboard service calls its upstream service Counting on any path (except /admin), traffic will be routed to healthy instances of the Counting service because:

- service-router will match the “catch-all” rule and forward traffic within Counting Discovery Chain
- service-splitter not explicitly configured for Counting
- service-resolver not explicitly for Counting

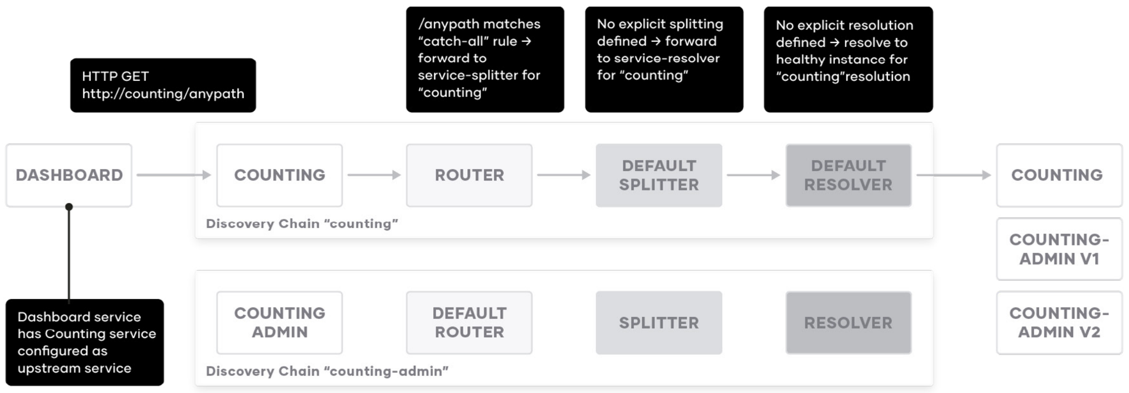

HTTP GET
http://counting/anypath
/anypath matches "catch-all" rule → forward to service-splitter for "counting"
No explicit splitting defined → forward to service-resolver for "counting"
No explicit resolution defined → resolve to healthy instance for "counting"resolution
DASHBOARD
COUNTING
ROUTER
DEFAULT SPLITTER
DEFAULT RESOLVER
COUNTING
Discovery Chain "counting"
COUNTING-ADMIN V1
Dashboard service has Counting service configured as upstream service
COUNTING ADMIN
DEFAULT ROUTER
SPLITTER
RESOLVER
COUNTING-ADMIN V2
Discovery Chain "counting-admin"

Figure 40: Discovery Chain visualization for Dashboard—Counting /anypath

Example 2: Dashboard—Counting /admin

If the Dashboard service calls its upstream service Counting on path /admin, traffic will be re-routed and split in an 80/20 fashion between healthy instances of the Counting-Admin service "v1" and "v2" service subsets because:

- service-router will match "/admin" rule, hence endpoint resolution continues in Counting-Admin Discovery Chain
- service-splitter for Counting-Admin splits traffic between "v1" and "v2" subset
- service-resolver for Counting-Admin defines subsets for "v1" and "v2" based on service meta-data

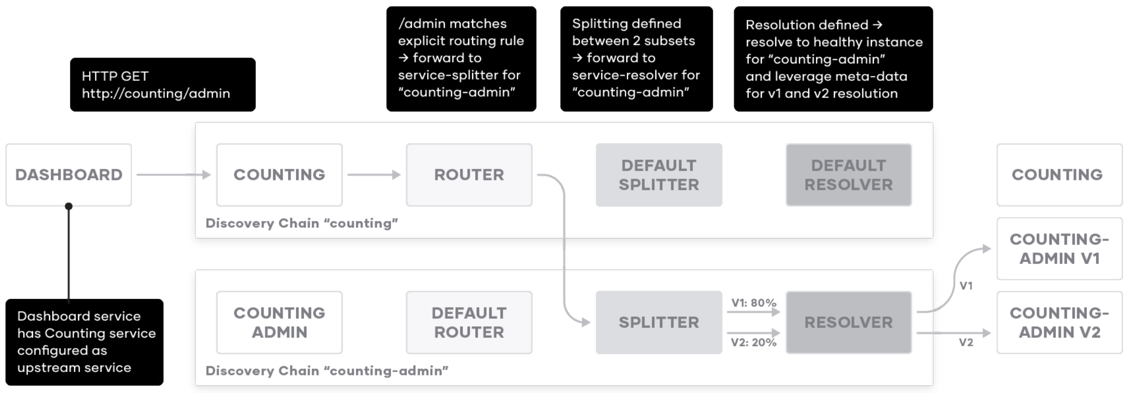

HTTP GET
http://counting/admin
/admin matches explicit routing rule → forward to service-splitter for "counting-admin"
Splitting defined between 2 subsets → forward to service-resolver for "counting-admin"
Resolution defined → resolve to healthy instance for "counting-admin" and leverage meta-data for v1 and v2 resolution
DASHBOARD
COUNTING
ROUTER
DEFAULT SPLITTER
DEFAULT RESOLVER
Discovery Chain "counting"
COUNTING
COUNTING-ADMIN V1
COUNTING-ADMIN V2
COUNTING ADMIN
DEFAULT ROUTER
SPLITTER
V1: 80%
V2: 20%
RESOLVER
V1
V2
Discovery Chain "counting-admin"
Dashboard service has Counting service configured as upstream service

Figure 41: Discovery Chain visualization for Dashboard—Counting /admin

Example 3: Service X—Counting-Admin

If any other service calls the Counting-Admin service as its upstream service directly on any path, traffic will be routed and split in an 80/20 fashion between healthy instances of the Counting-Admin service "v1" and "v2" service subsets because:

- service-router not explicitly configured for Counting-Admin
- service-splitter for Counting-Admin splits traffic between "v1" and "v2" subset
- service-resolver for Counting-Admin defines subsets for "v1" and "v2" based on service meta-data

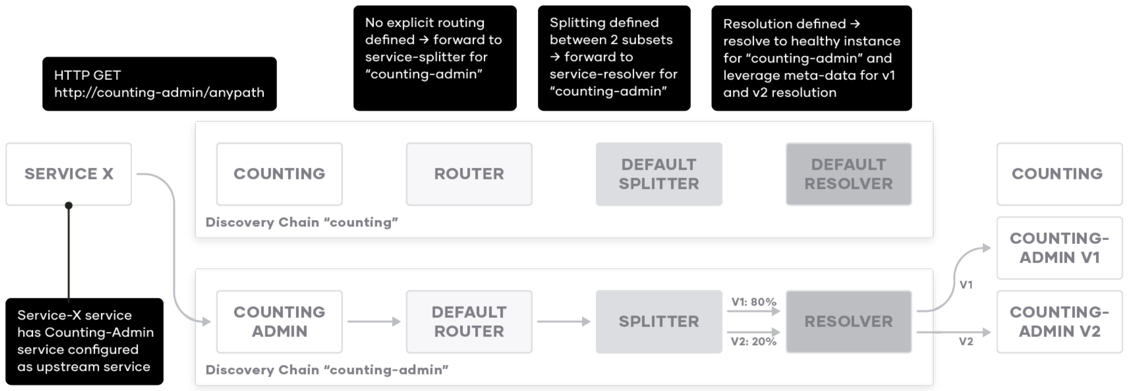

HTTP GET
http://counting-admin/anypath
No explicit routing defined → forward to service-splitter for "counting-admin"
Splitting defined between 2 subsets → forward to service-resolver for "counting-admin"
Resolution defined → resolve to healthy instance for "counting-admin" and leverage meta-data for v1 and v2 resolution
SERVICE X
COUNTING
ROUTER
DEFAULT SPLITTER
DEFAULT RESOLVER
Discovery Chain "counting"
COUNTING
COUNTING-ADMIN V1
COUNTING-ADMIN V2
COUNTING ADMIN
DEFAULT ROUTER
SPLITTER
V1: 80%
V2: 20%
RESOLVER
V1
V2
Discovery Chain "counting-admin"
Service-X service has Counting-Admin service configured as upstream service

Figure 42: Discovery Chain visualization for Service-X—Counting-Admin.

Resolving “virtual services” with Consul’s Discovery Chain

Consul’s service discovery chain allows Consul to resolve services which have no instances actually registered in Consul’s service catalog.

Service-routers, service-splitters and service-resolvers can all be defined against any service name. We call this pattern of creating routing rules for a service name with no backing instances a “virtual service.”

This “virtual service” can then be referenced as an upstream service directly or by another stage of the Discovery Chain.

A “virtual service” is only useful if it resolves to a real service at some point in the Discovery Chain.

The following configuration examples show all stages of the Consul Discovery Chain referencing a “virtual service”, which ultimately resolve real service endpoints in the last stage of the Discovery Chain at the service-resolver.

The following service-router configuration exposes a virtual service called “virtual-admin” which does not exist in Consul’s service catalog.

The service-router routes traffic for path /login to a service named “login” and everything else to another “virtual service” called “global-admin”:

```
$ consul config write -<<EOF
kind = “service-router”
name = “virtual-admin”
routes = [
 {
 match {
  http {
  path_prefix = “/login”
  }
 }
 destination {
  service = “login”,
  prefix_rewrite = “/”
 }
 },
 {
 destination {
  service = “global-admin”
 }
 }
]
EOF
```

The next resolution stage for the virtual “global-admin” service would be the service-splitter stage which defines splits across two other virtual services called “admin-dc1” and “admin-dc2”:

```
$ consul config write -<<EOF
kind = “service-splitter”
name = “global-admin”
splits = [
 {
 weight = 50
 service = “admin-dc1”
 },
 {
 weight = 50
 service = “admin-dc2”
 },
]
EOF
```

The ultimate resolution of the virtual services “admin-dc1” and “admin-dc2” will then happen at the last stage of Consul’s Discovery Chain in the service-resolver stage, which will resolve the “admin” service in the respective Consul Datacenter:

```
$ consul config write -<<EOF
kind = “service-resolver”
name = “admin-dc1”
redirect {
 service = “admin”
 data center = “dc1”
}
EOF
```

```
$ consul config write -<<EOF
kind = “service-resolver”
name = “admin-dc2”
redirect {
 service = “admin”
 data center = “dc2”
}
EOF
```

This example shows how the resolution to real service endpoints in Consul’s Discovery Chain works, even though an upstream service was defined within a service definition referencing a service which has no real endpoint registered in Consul’s service catalog:

In this example “virtual-admin” → “global-admin” → “admin-dc1” / “admin-dc” → “admin” in data center dc1 / dc2.

Packet walk Consul Service Mesh Single-DC

We've discussed the various traffic management configuration options and how the resulting configurations are loaded into the Envoy data-plane constructs through the Consul Client Agent. This chapter will discuss the connection setup between sidecar proxy instances within the service mesh and detail how packets travel through the system from source to destination service.

To get all the benefits out of Consul service mesh like service-to-service authentication, authorization and automatic encryption of service-to-service traffic, without the need to implement all of those features directly into the application itself, it is necessary that applications connect to the target upstream service through the sidecar proxy.

The first step in the life of a packet through Consul service mesh is the session initiation of the downstream service, the Dashboard application in our example, towards the respective sidecar proxy instance.

Remember that the service definition of the Dashboard service looks as follows. As discussed earlier, this configuration instructs Consul service mesh to implement a local TCP listener at port 9191 on the Dashboard service's local sidecar proxy:

```
“service”: {
 “name”: “dashboard”,
 “port”: 9002,
 “connect”: {
  “sidecar_service”: {
  “proxy”: {
   “upstreams”: [
   { “destination_name”: “counting”,
    “local_bind_port”: 9191 }
   ]
  }}
 }}
```

When the Dashboard now wants to connect to the Counting service it will not reach out to the Counting service directly. Instead it will initiate a connection to:

127.0.0.1:9191

which is the respective TCP port on the sidecar proxy. From there, the sidecar proxy will provide services such as Layer 7 routing and end-to-end encryption.

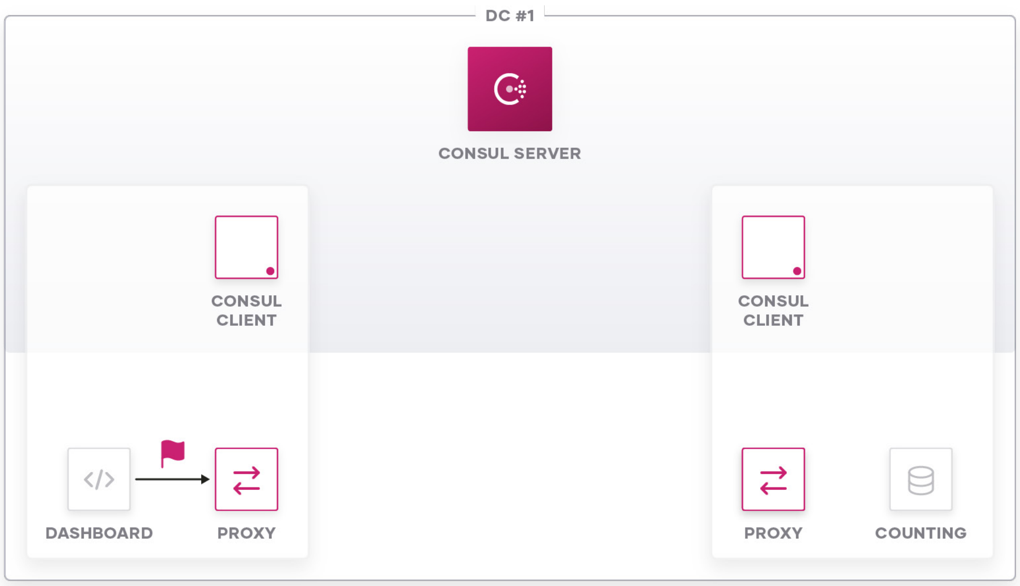
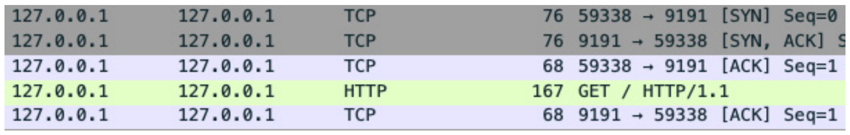


Figure 43: Session initialization of Dashboard service to its local sidecar proxy.



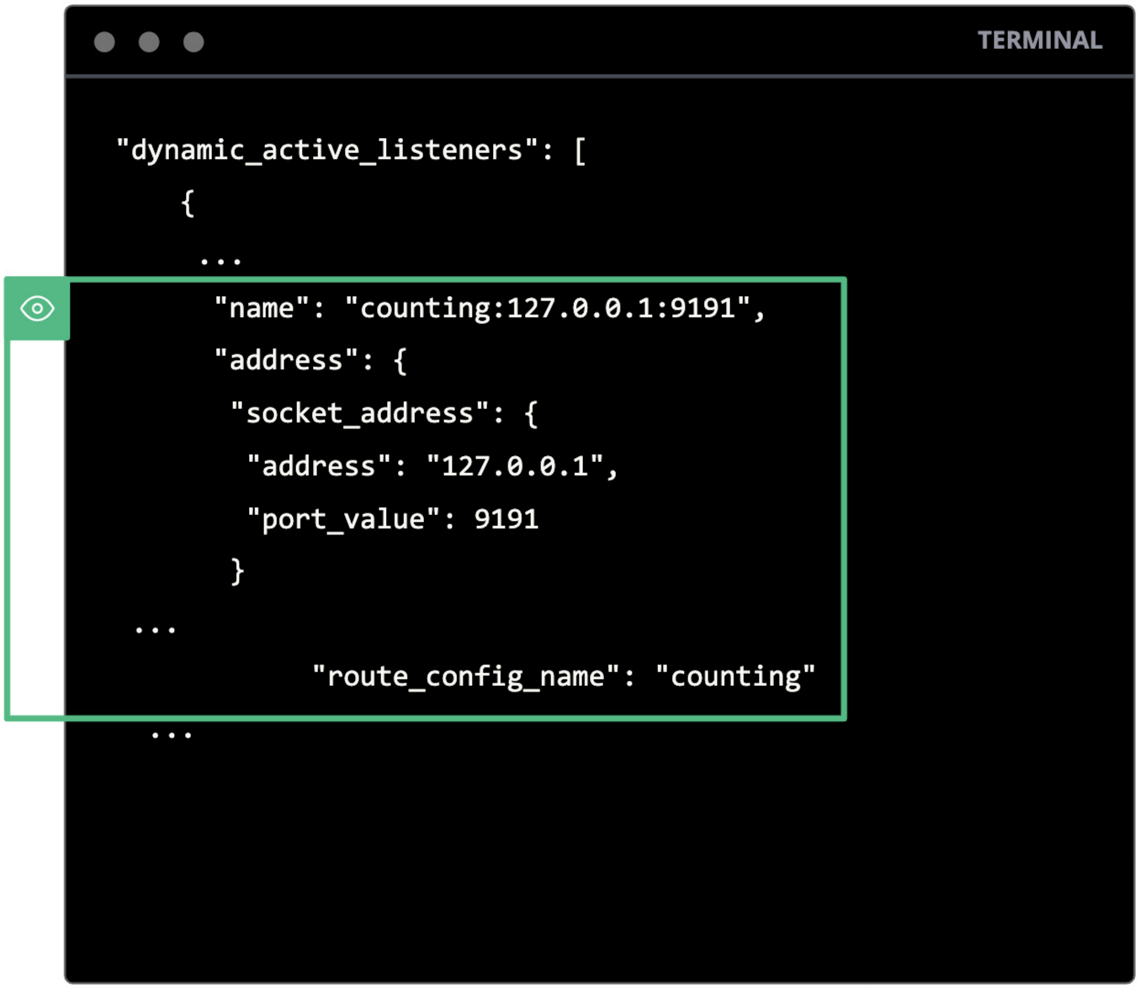
```
127.0.0.1        127.0.0.1        TCP              76  59338 → 9191 [SYN] Seq=0
127.0.0.1        127.0.0.1        TCP              76  9191 → 59338 [SYN, ACK] S
127.0.0.1        127.0.0.1        TCP              68  59338 → 9191 [ACK] Seq=1
127.0.0.1        127.0.0.1        HTTP            167  GET / HTTP/1.1
127.0.0.1        127.0.0.1        TCP              68  9191 → 59338 [ACK] Seq=1
```

Figure 44: Packet capture of TCP handshake of Dashboard service to its local sidecar proxy.

The local sidecar proxy will accept this TCP connection and based on the listener port (9191 in this example) the Dashboard app initiates the connection to, the sidecar proxy will forward the traffic to the appropriate upstream destination.

The Envoy data-plane is already pre-configured and the resolution of available endpoints of the upstream service looks as follows:

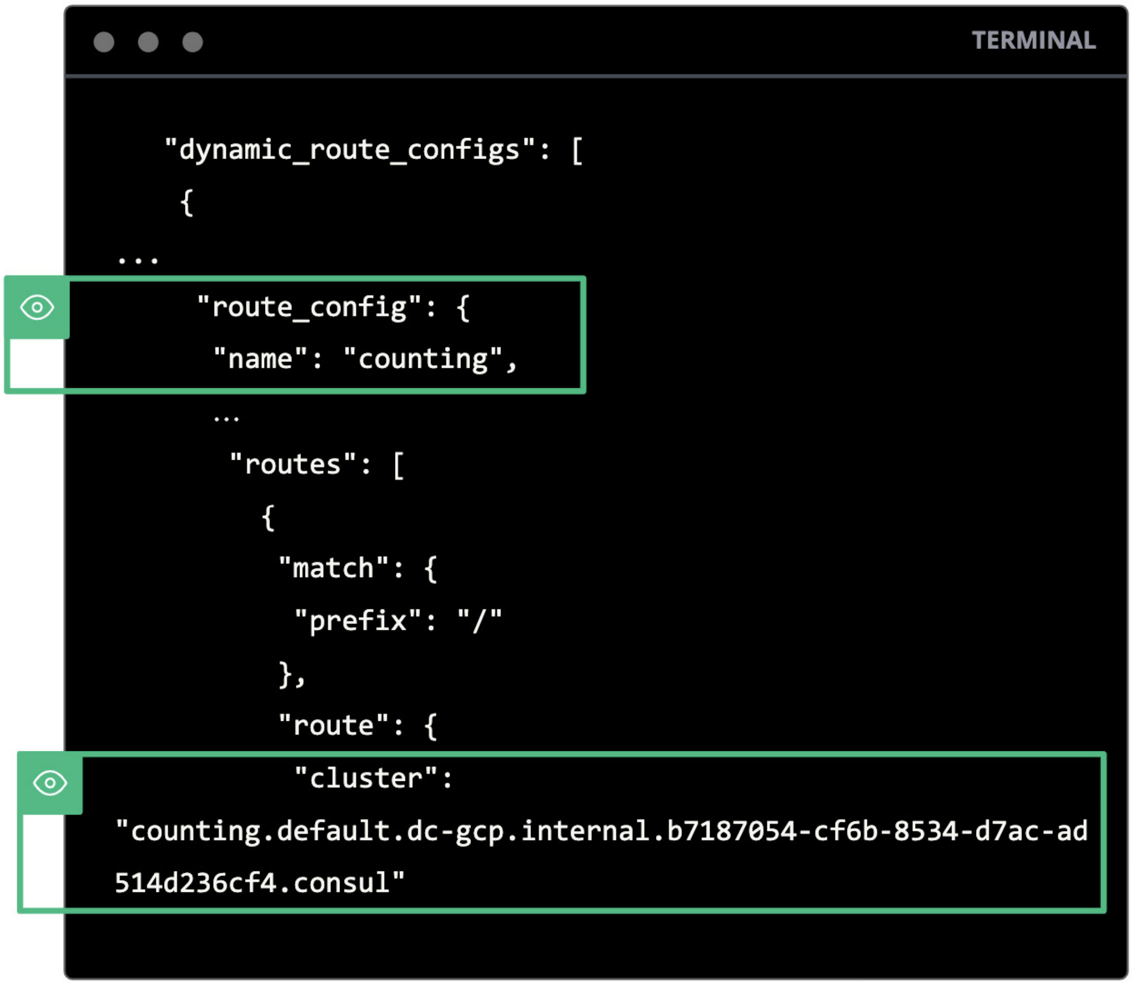
The Envoy sidecar proxy received its dynamic listener configuration from the Consul Client Agent for TCP port 9191 and also has a dynamic route configuration for the Counting service, which is referenced in the dynamic listener configuration:



```
"dynamic_active_listeners": [
    {
     ...
      "name": "counting:127.0.0.1:9191",
      "address": {
       "socket_address": {
        "address": "127.0.0.1",
        "port_value": 9191
       }
 ...
            "route_config_name": "counting"
  ...
```

Figure 45: Extract of Envoy data-plane listener configuration.

The dynamic route for the Counting service in this example does not contain any specific L7 routes, hence it will forward all the traffic to a healthy available endpoint in the cluster for the Counting service attached to the route:

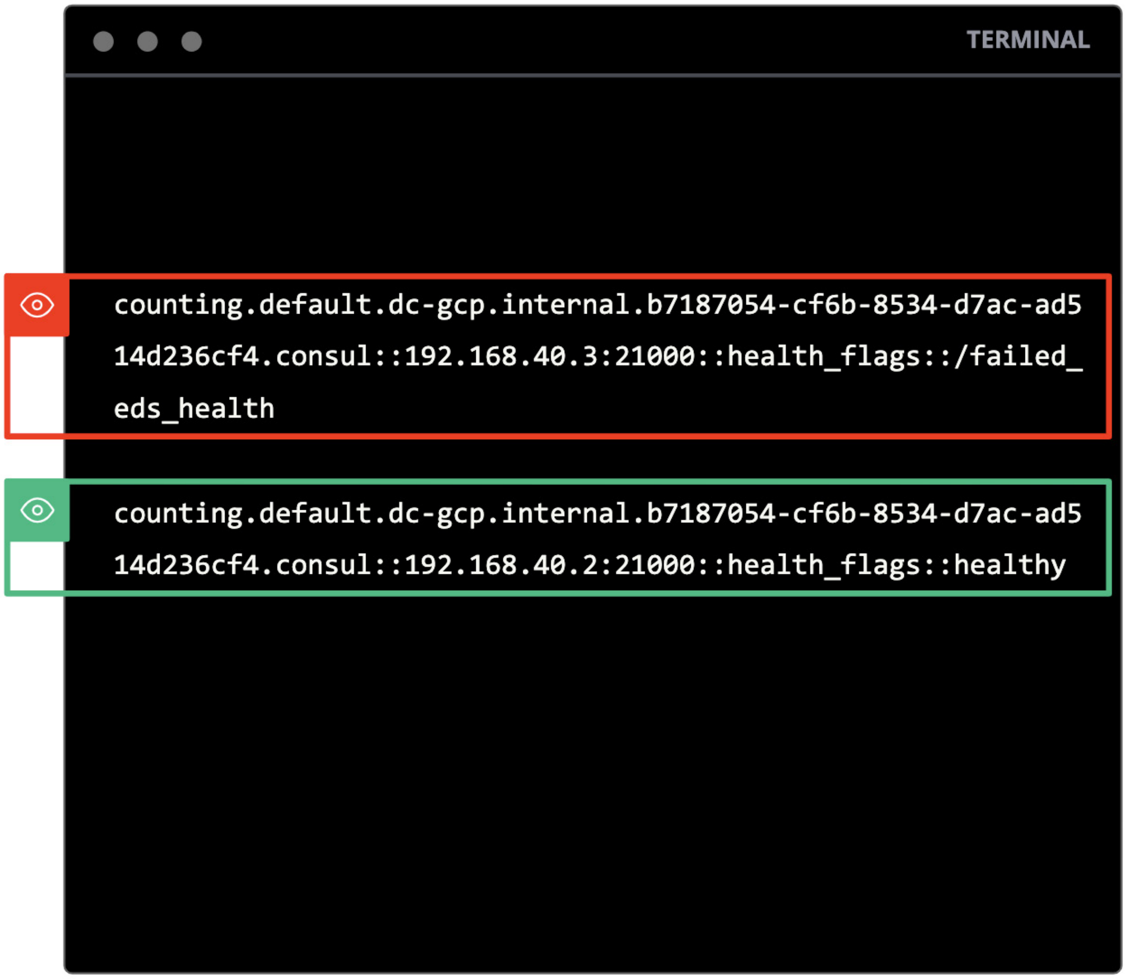


```
    "dynamic_route_configs": [
     {
...
       "route_config": {
        "name": "counting",
        ...
         "routes": [
          {
           "match": {
            "prefix": "/"
           },
           "route": {
            "cluster":
"counting.default.dc-gcp.internal.b7187054-cf6b-8534-d7ac-ad
514d236cf4.consul"
```

Figure 46: Extract of Envoy data-plane route configuration.

The available service endpoints for the upstream Counting service are pre-provisioned within the Envoy cluster, so that Envoy can pick a healthy instance according to the configured load balancing algorithm (round-robin per default) and initiate a session with the upstream service's sidecar proxy.

Service endpoints are also calculated and pre-provisioned into the Envoy data-plane, avoiding the need for additional lookups at runtime.



```
counting.default.dc-gcp.internal.b7187054-cf6b-8534-d7ac-ad5
14d236cf4.consul::192.168.40.3:21000::health_flags::/failed_
eds_health

counting.default.dc-gcp.internal.b7187054-cf6b-8534-d7ac-ad5
14d236cf4.consul::192.168.40.2:21000::health_flags::healthy
```

Figure 47: Associated Envoy EDS cluster endpoints for Counting service.

Generally speaking, you can think of the session establishment for service-to-service traffic within Consul service mesh as a two step approach:

1) Mutual TLS session for service-to-service authentication and traffic encryption
2) Service-to-service traffic authorization at the destination service

After the local Envoy sidecar proxy chooses an upstream service endpoint to connect to, it will initiate a mutual TLS session with the public listener of the upstream Counting service sidecar proxy (TCP 21000 in this example).

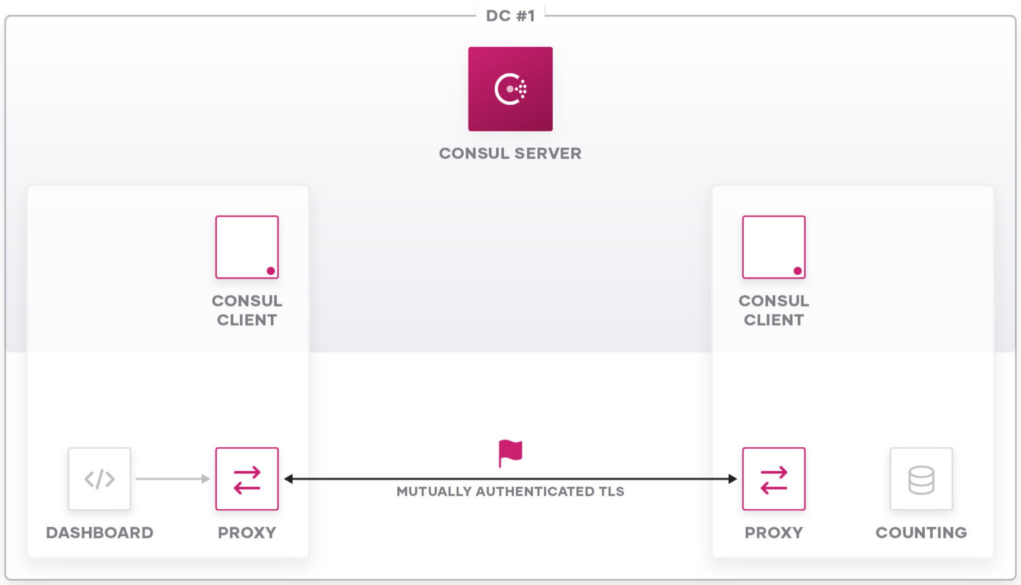
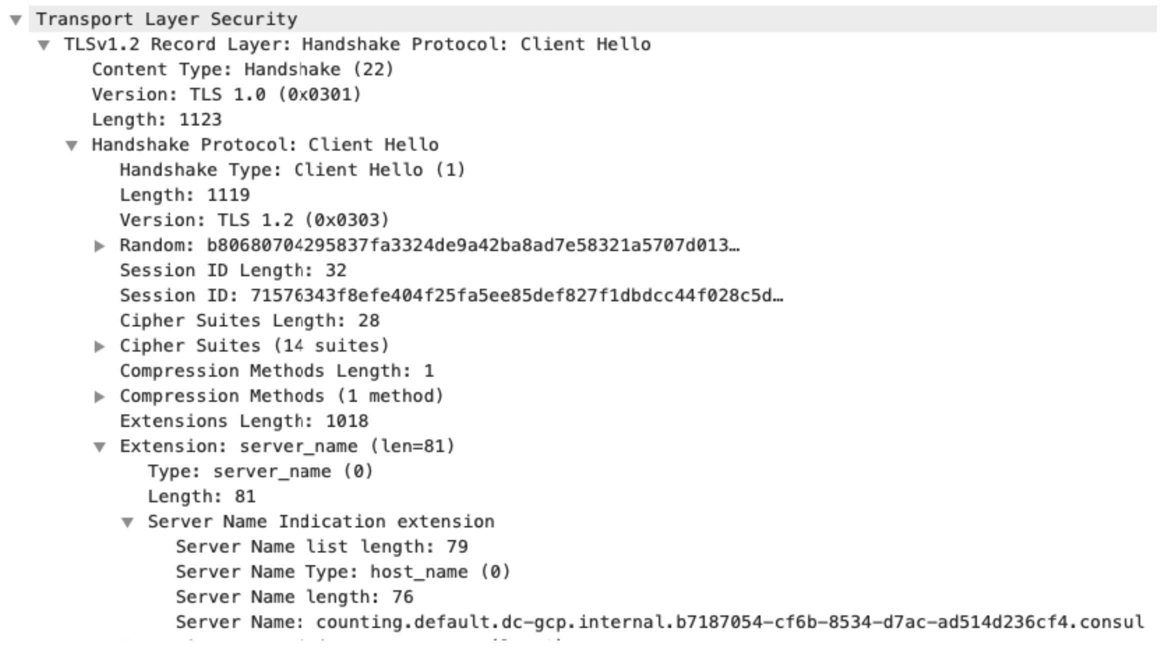


Figure 48: Sidecar proxy to sidecar proxy mTLS session initialization.

```
TLSv1.2         1196 Client Hello
TLSv1.2         1006 Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done
TLSv1.2          926 Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message
TLSv1.2         1014 New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
```

Figure 49: Packet capture of TLS handshake between sidecar proxies.

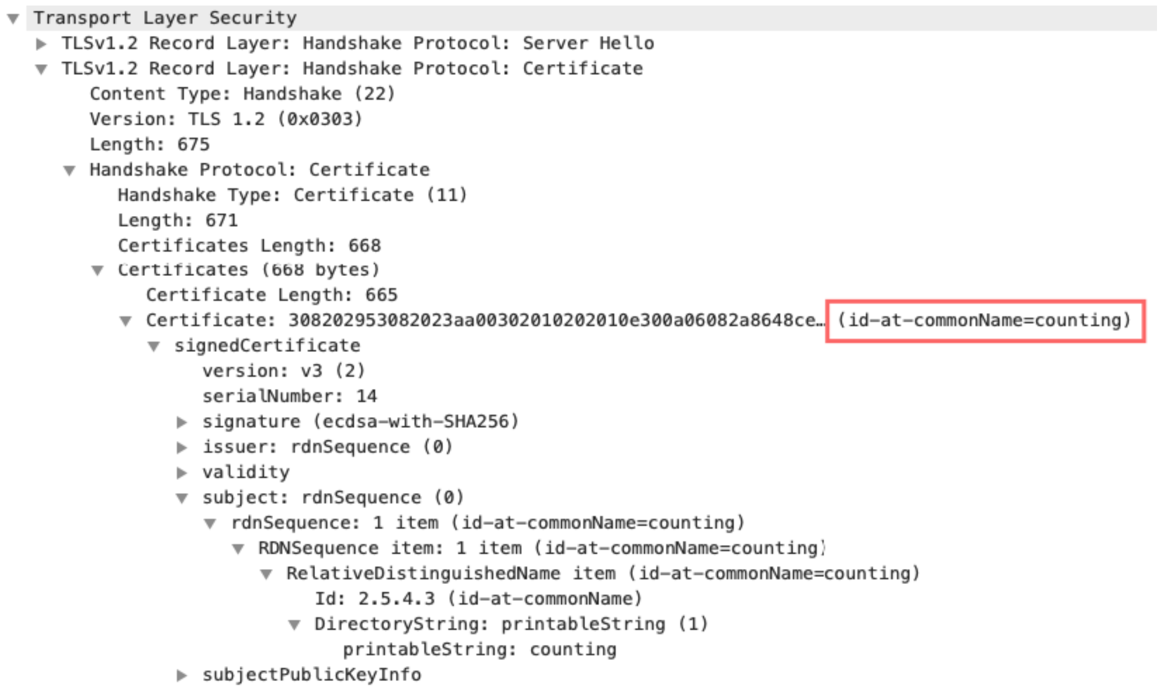
The first step of the mutual TLS connection setup is the "Client Hello" from the source proxy to the destination proxy which contains an SNI header identifying the target service for whom this packet is intended:



```
▼ Transport Layer Security
  ▼ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 1123
    ▼ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 1119
        Version: TLS 1.2 (0x0303)
      ▶ Random: b80680704295837fa3324de9a42ba8ad7e58321a5707d013…
        Session ID Length: 32
        Session ID: 71576343f8efe404f25fa5ee85def827f1dbdcc44f028c5d…
        Cipher Suites Length: 28
      ▶ Cipher Suites (14 suites)
        Compression Methods Length: 1
      ▶ Compression Methods (1 method)
        Extensions Length: 1018
      ▼ Extension: server_name (len=81)
          Type: server_name (0)
          Length: 81
        ▼ Server Name Indication extension
            Server Name list length: 79
            Server Name Type: host_name (0)
            Server Name length: 76
            Server Name: counting.default.dc-gcp.internal.b7187054-cf6b-8534-d7ac-ad514d236cf4.consul
```

Figure 50: TLS handshake Client Hello.

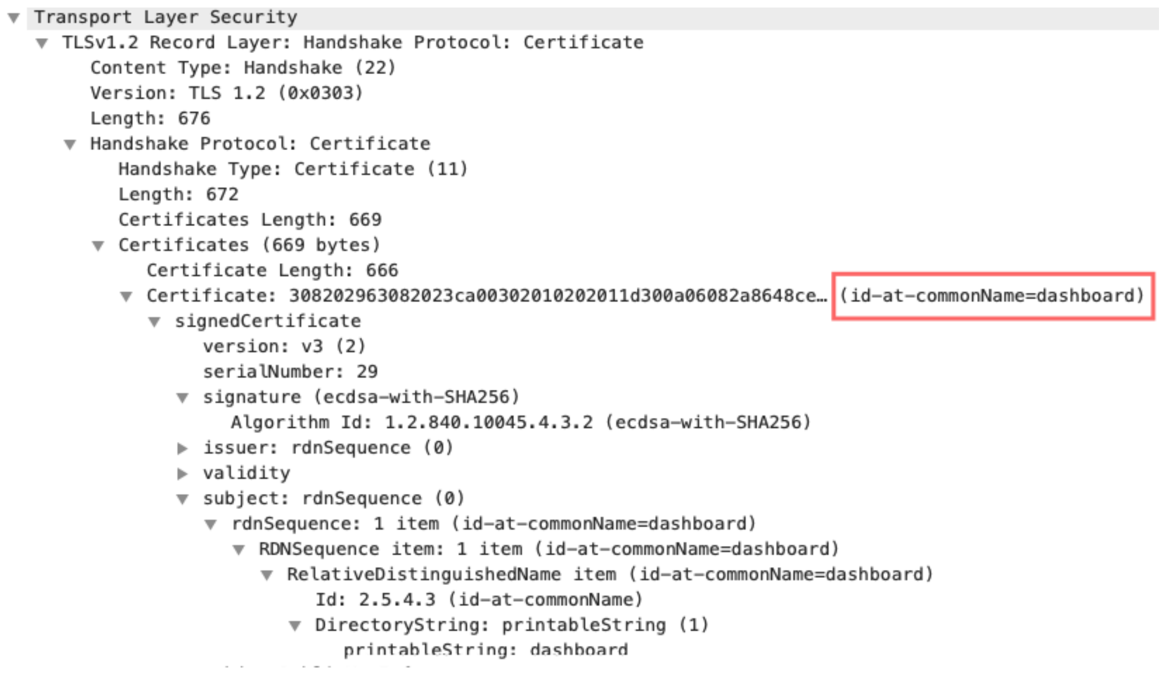
During the “Server Hello” the destination sidecar proxy offers its identity to the source sidecar proxy so that the source sidecar proxy can verify it connected to the intended service:



```
▼ Transport Layer Security
  ▶ TLSv1.2 Record Layer: Handshake Protocol: Server Hello
  ▼ TLSv1.2 Record Layer: Handshake Protocol: Certificate
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 675
    ▼ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 671
        Certificates Length: 668
      ▼ Certificates (668 bytes)
          Certificate Length: 665
        ▼ Certificate: 308202953082023aa00302010202010e300a06082a8648ce… (id-at-commonName=counting)
          ▼ signedCertificate
              version: v3 (2)
              serialNumber: 14
            ▶ signature (ecdsa-with-SHA256)
            ▶ issuer: rdnSequence (0)
            ▶ validity
            ▼ subject: rdnSequence (0)
              ▼ rdnSequence: 1 item (id-at-commonName=counting)
                ▼ RDNSequence item: 1 item (id-at-commonName=counting)
                  ▼ RelativeDistinguishedName item (id-at-commonName=counting)
                      Id: 2.5.4.3 (id-at-commonName)
                    ▼ DirectoryString: printableString (1)
                        printableString: counting
            ▶ subjectPublicKeyInfo
```

Figure 51: TLS handshake Server Hello and Certificate.

As a last step of the mutual TLS session setup, the destination sidecar proxy asks for the identity of the source sidecar proxy which is provided by the service certificate the Consul Client Agent installed into the Envoy data-plane during the sidecar proxy bootstrapping process:



```
▼ Transport Layer Security
  ▼ TLSv1.2 Record Layer: Handshake Protocol: Certificate
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 676
    ▼ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 672
        Certificates Length: 669
      ▼ Certificates (669 bytes)
          Certificate Length: 666
        ▼ Certificate: 308202963082023ca00302010202011d300a06082a8648ce… (id-at-commonName=dashboard)
          ▼ signedCertificate
              version: v3 (2)
              serialNumber: 29
            ▼ signature (ecdsa-with-SHA256)
                Algorithm Id: 1.2.840.10045.4.3.2 (ecdsa-with-SHA256)
            ▶ issuer: rdnSequence (0)
            ▶ validity
            ▼ subject: rdnSequence (0)
              ▼ rdnSequence: 1 item (id-at-commonName=dashboard)
                ▼ RDNSequence item: 1 item (id-at-commonName=dashboard)
                  ▼ RelativeDistinguishedName item (id-at-commonName=dashboard)
                      Id: 2.5.4.3 (id-at-commonName)
                    ▼ DirectoryString: printableString (1)
                        printableString: dashboard
```

Figure 52: TLS handshake Client Certificate.

At this point the TLS session has been established and both proxy instances have verified each other's identity.

They have also established an encrypted channel which secures data in transit for their respective service.

After the proxies are mutually authenticated, the next step is the connection Authorization, which takes place at the destination sidecar proxy.

The reason the authorization happens at the destination is for security purposes and to avoid privilege escalation scenarios which might be possible if an attacker compromised the source service, and authorization was enforced at the source of the connection.

To authorize the connection attempt from the Dashboard sidecar proxy, the Counting sidecar proxy (i.e., the destination) will extract the SPIFFE identities of the source and destination service out of the X.509 service certificates and will perform an AuthZ callback to the local Consul Client Agent through the existing gRPC channel between the Envoy sidecar proxy and the Consul Client Agent.

Remember that the local Consul Client Agent holds a cache of relevant intentions for services registered with this Consul Client.

Hence the AuthZ request can be handled locally and no callback to the central servers is required.

By caching the relevant intentions in a distributed fashion, Consul service mesh allows for faster intention verification and for a more robust and scalable system, enabling AuthZ checks to be answered even when there might be an outage in between the local Consul Client Agent and the central Consul Server backend.

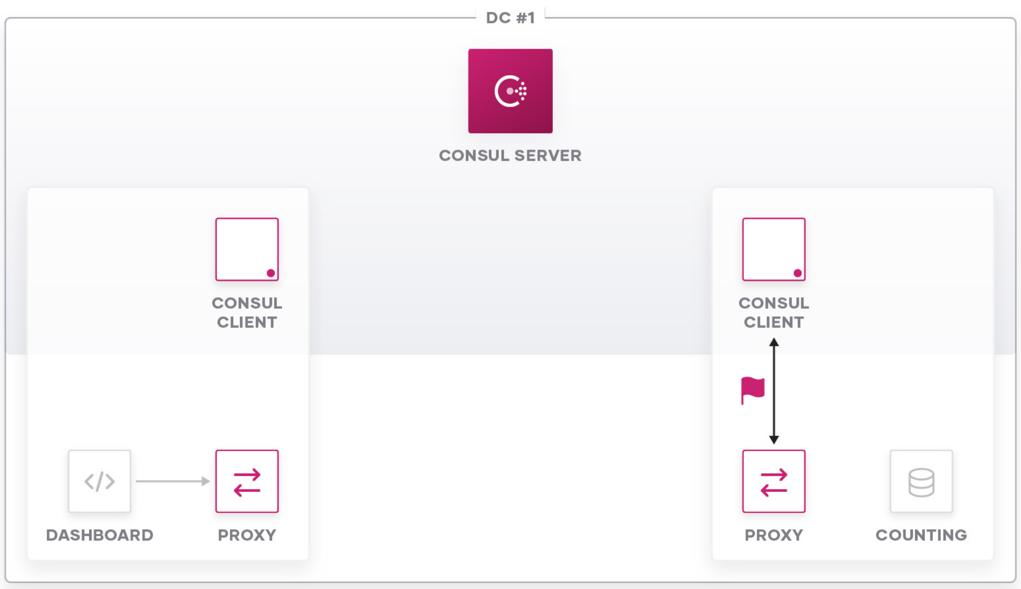
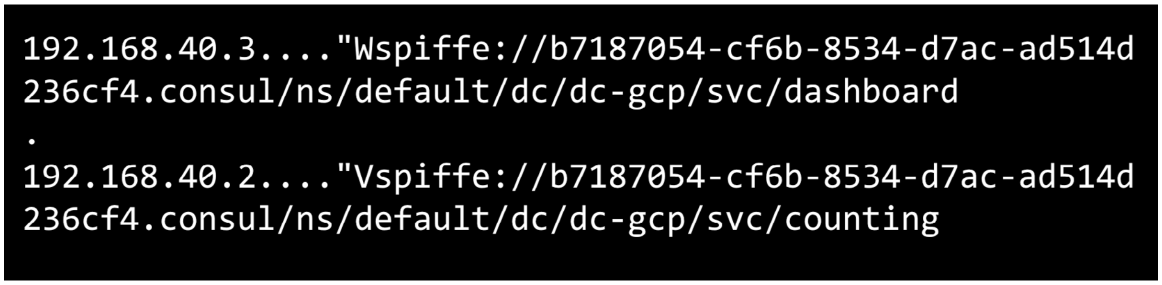


Figure 53: AuthZ callback from Envoy to Consul Client Agent.



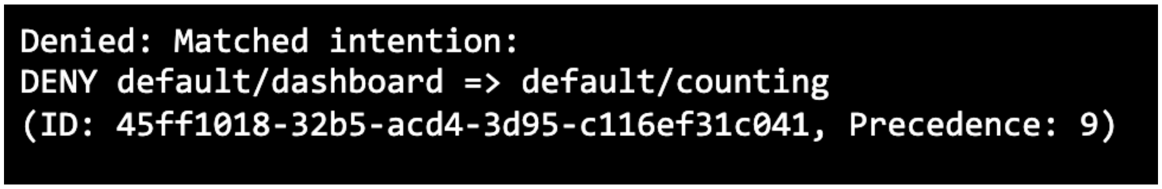
```
192.168.40.3...."Wspiffe://b7187054-cf6b-8534-d7ac-ad514d
236cf4.consul/ns/default/dc/dc-gcp/svc/dashboard
.
192.168.40.2...."Vspiffe://b7187054-cf6b-8534-d7ac-ad514d
236cf4.consul/ns/default/dc/dc-gcp/svc/counting
```

Figure 54: Raw SPIFFE IDs presented from Envoy to Consul Client Agent.

There are two possible responses that a Consul client can return for an AuthZ request from the local Envoy proxy.

Case 1: Intention denies Communication between Services

If the configured intentions do not allow communication between the two service endpoints, the local Consul Client Agent will reply back to the AuthZ request of Envoy that the communication is "Denied" by an intention. It will also reference the matching intention and report the intention ID:



```
Denied: Matched intention:
DENY default/dashboard => default/counting
(ID: 45ff1018-32b5-acd4-3d95-c116ef31c041, Precedence: 9)
```

Figure 55: Consul Client Agent reply denying connection attempt.

The destination sidecar proxy will immediately send a TCP Reset to the source sidecar proxy which will cause the session to be terminated.

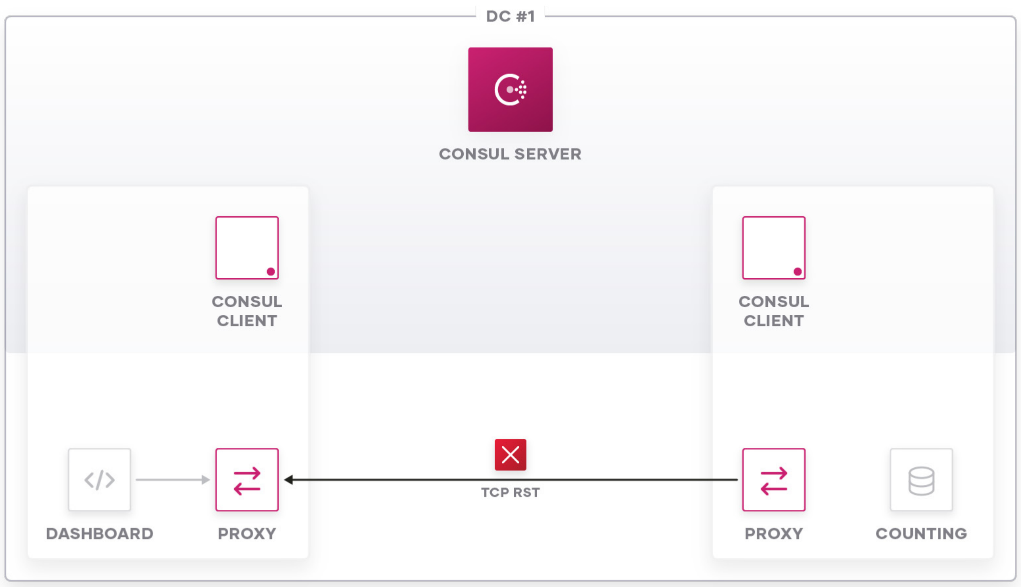
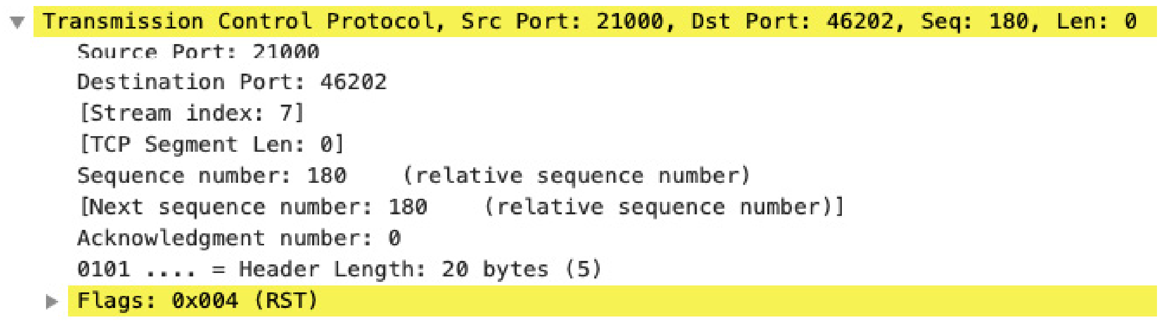


Figure 56: TCP session reset by destination sidecar proxy.

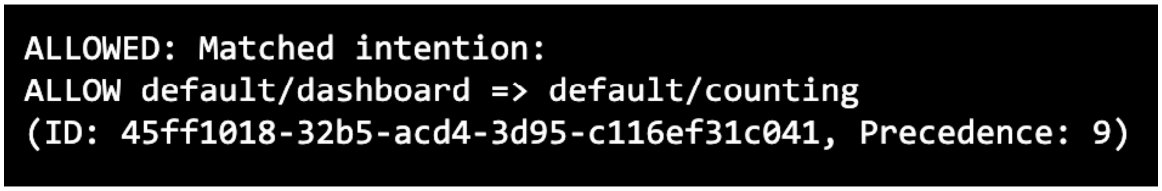


```
▼ Transmission Control Protocol, Src Port: 21000, Dst Port: 46202, Seq: 180, Len: 0
     Source Port: 21000
     Destination Port: 46202
     [Stream index: 7]
     [TCP Segment Len: 0]
     Sequence number: 180    (relative sequence number)
     [Next sequence number: 180    (relative sequence number)]
     Acknowledgment number: 0
     0101 .... = Header Length: 20 bytes (5)
  ▶ Flags: 0x004 (RST)
```

Figure 57: Packet capture of TCP session reset by destination sidecar proxy.

Case 2: Intention allows Communication between Services

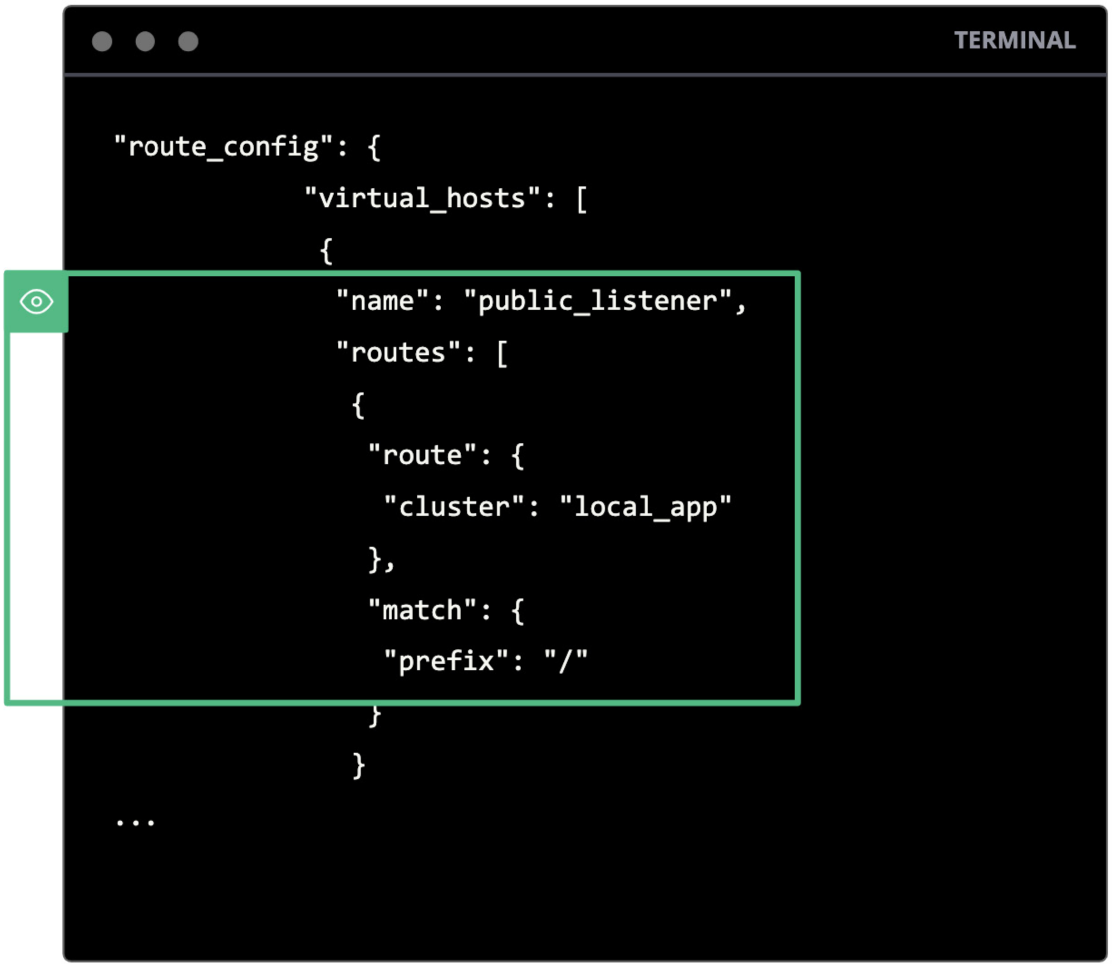
If the configured intentions allow communication between the two service endpoints, the local Consul Client Agent will reply back to the AuthZ request that the communication is "Allowed" by an intention. It will also reference the matching intention and report the intention ID:



```
ALLOWED: Matched intention:
ALLOW default/dashboard => default/counting
(ID: 45ff1018-32b5-acd4-3d95-c116ef31c041, Precedence: 9)
```

Figure 58: Consul Client Agent reply allowing connection attempt.

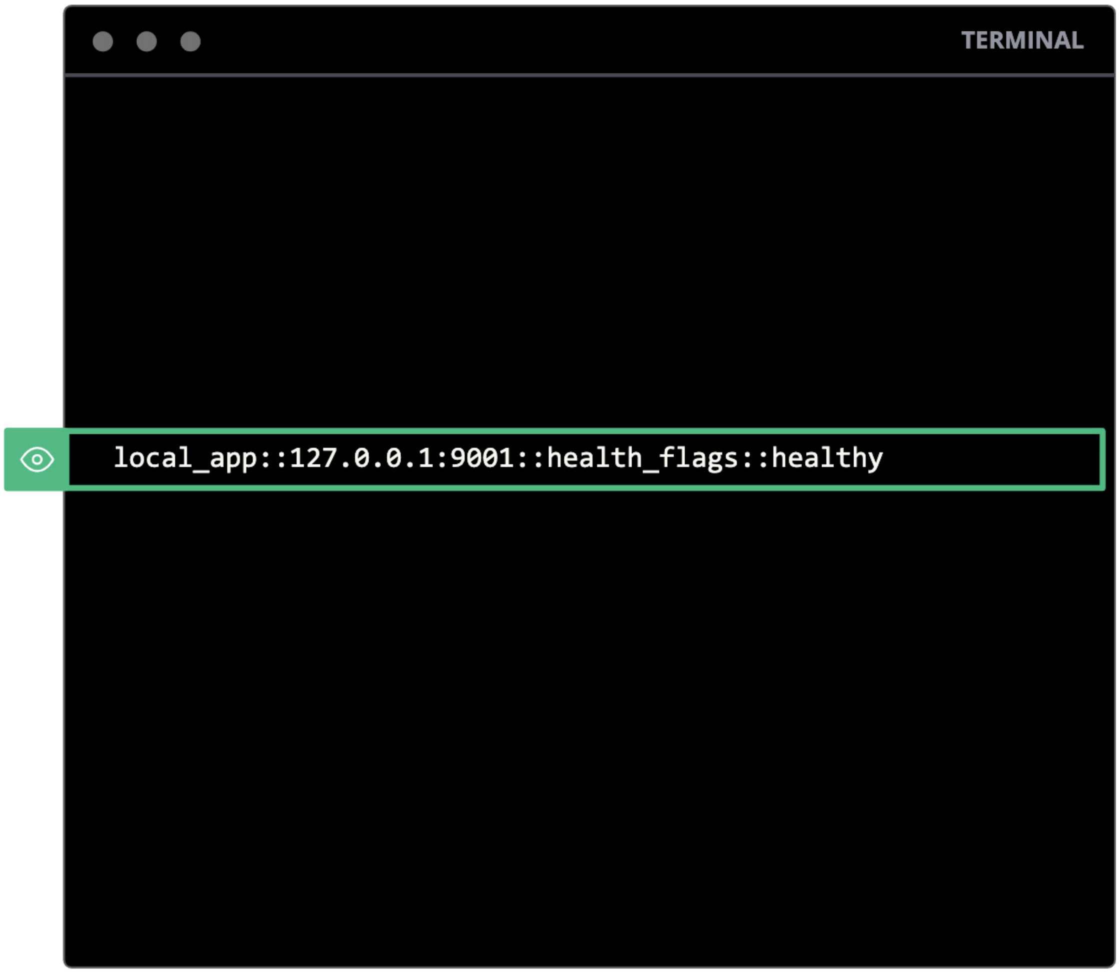
In this case, the destination sidecar proxy's public_listener virtual host will route traffic to the 'local_app' cluster.



```
"route_config": {
            "virtual_hosts": [
             {
              "name": "public_listener",
              "routes": [
               {
                "route": {
                 "cluster": "local_app"
                },
                "match": {
                 "prefix": "/"
                }
               }
...
```

Figure 59: Extract of Envoy data-plane route configuration at destination.

The "local_app" cluster contains only one endpoint which is the Counting application running on "localhost" next to the sidecar proxy and listening to TCP port 9001 as was defined earlier in the Counting service definition."



```
local_app::127.0.0.1:9001::health_flags::healthy
```

Figure 60: Associated Envoy static cluster for local_app.

As the final step in the session setup, the destination sidecar proxy will initiate a TCP connection to the local application listener (Counting application TCP 9001 in this example) and forward the application data received from the source application to the destination application:

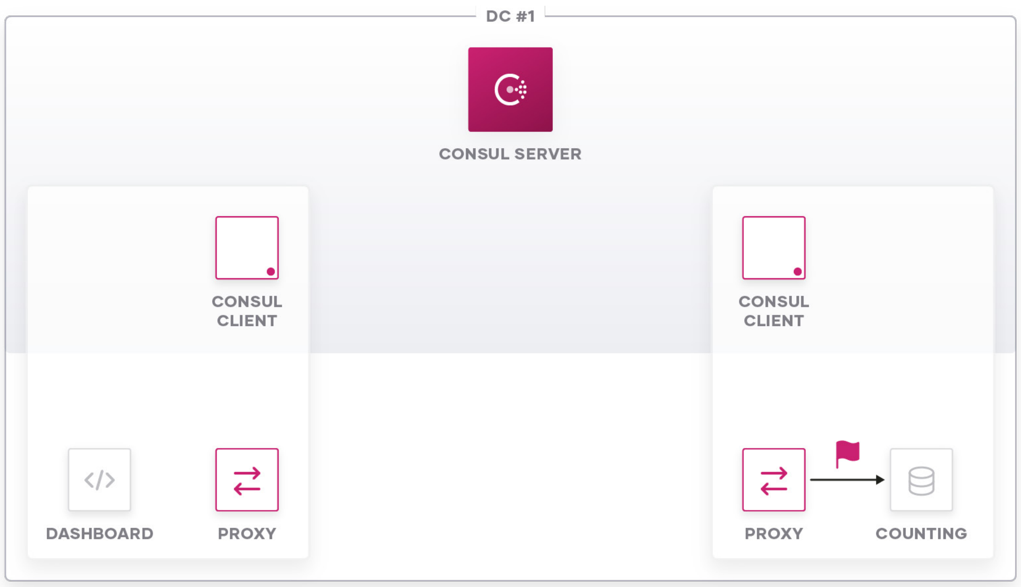
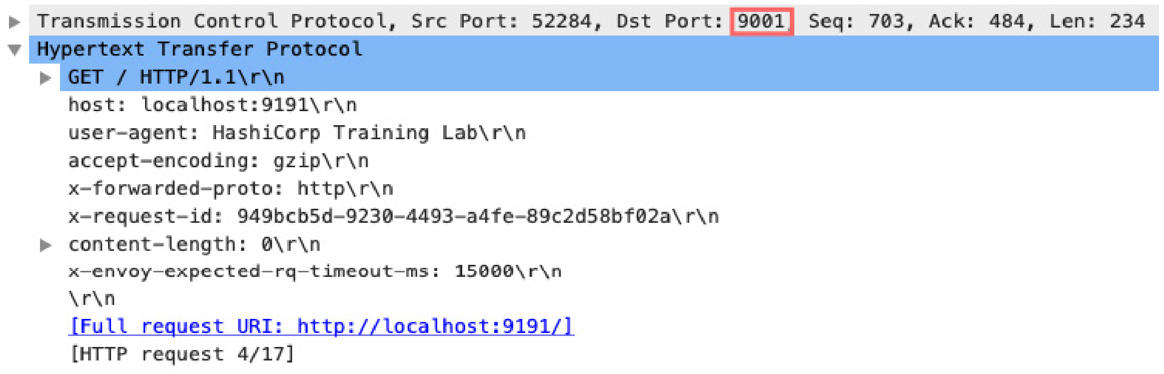


Figure 61: Session initialization of sidecar proxy to local Counting service.



```
▶ Transmission Control Protocol, Src Port: 52284, Dst Port: 9001  Seq: 703, Ack: 484, Len: 234
▼ Hypertext Transfer Protocol
  ▶ GET / HTTP/1.1\r\n
    host: localhost:9191\r\n
    user-agent: HashiCorp Training Lab\r\n
    accept-encoding: gzip\r\n
    x-forwarded-proto: http\r\n
    x-request-id: 949bcb5d-9230-4493-a4fe-89c2d58bf02a\r\n
  ▶ content-length: 0\r\n
    x-envoy-expected-rq-timeout-ms: 15000\r\n
    \r\n
    [Full request URI: http://localhost:9191/]
    [HTTP request 4/17]
```

Figure 62: Packet capture of session to local Counting service.

After the initial session is established, the TCP session between the Envoy sidecar proxy instances is kept open and subsequent requests from the Dashboard application to the Counting application will reuse this connection.

Application data flows end-to-end between the service instances and data is unchanged for L4 services. For L7 services payload data is unchanged (dependent on L7 Traffic Management configurations, e.g., HTTP path rewrite).

Application data is always TLS encrypted between the sidecar proxy instances, regardless of whether the applications are already encrypting their own connections.

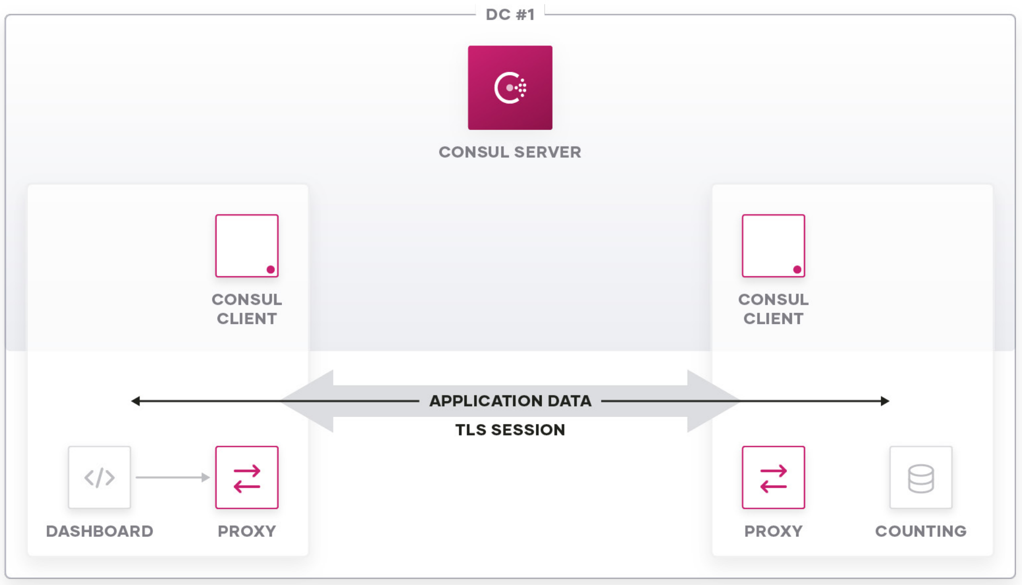


Figure 63: Application data flow vs. mTLS session between sidecar proxy instances.

Consul Multi-DC Service Mesh

Consul offers the ability to federate multiple Consul Datacenters and run a common service mesh across data centers, regardless of whether the instances are running on different public/private clouds, or different runtime platforms.

This provides the ability to run a service mesh across containerized environments, like Kubernetes, interconnecting those containerized applications back to services running on bare metal machines or within a VM based environment. Consul service mesh also supports operating exclusively within a bare metal or VM environment.

Normally service proxies require direct network connectivity to other endpoints participating in the service mesh.

This requirement presents challenges, for example, as pod IPs within a Kubernetes cluster may not be routable from the underlying network (depending on the CNI plugin used within the Kubernetes cluster) where bare metal or VM-based workloads are present. This makes it nearly impossible to interconnect the different runtime environments without changes to the networking layer, such as using NAT appliances, etc.

Another common challenge is interconnecting different networks which utilize overlapping IP addresses.

One could argue that circumstances like this might be a result of improper network address planning, but enterprises run into those kinds of problems not only due to planning errors, but also through mergers and acquisitions, where address conflicts arise when attempting to merge the different network environments.

Even in scenarios where there are non-overlapping IP addresses, security teams within an enterprise may not be willing to open network firewalls to permit direct connectivity between service endpoints as this might be a security concern – especially when the service mesh spans multiple cloud environments.

Consul 1.6 introduced a feature called “Mesh Gateways” which solves these challenges: it allows interconnecting various cloud and runtime environments, even though there might be IP address overlap between the participating endpoints. These gateways allow interconnectivity between multiple Consul Datacenters.

Mesh Gateways are a special type of proxy instance which handles routing traffic between different Consul Datacenters.

The Consul Mesh Gateway is based on Envoy and performs traffic routing between the different federated Consul Datacenters based on Server Name Indication (SNI) headers, while still allowing for end-to-end traffic encryption.

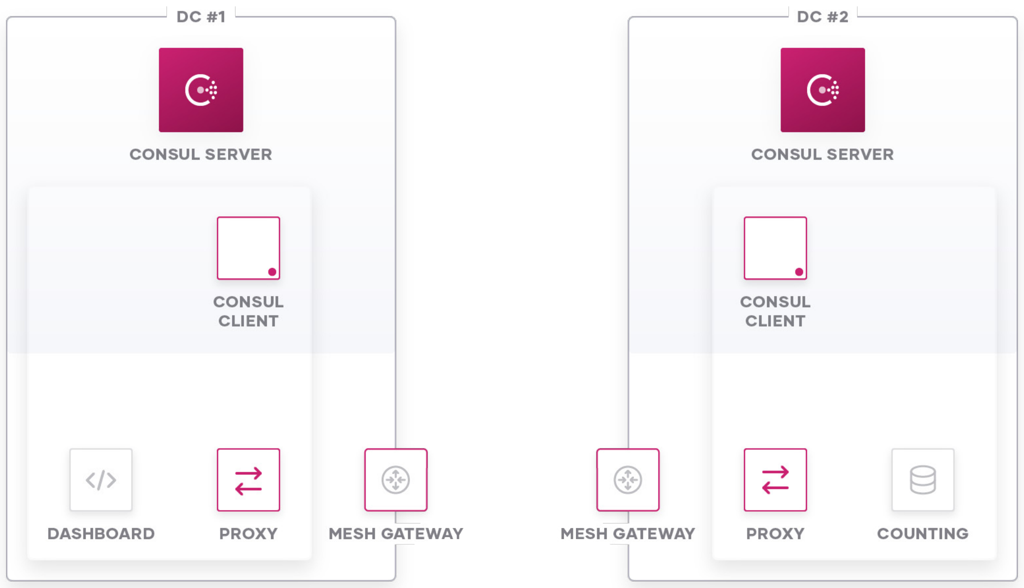


Figure 64: Dashboard—Counting Application setup spread across two Consul Datacenters.

Remark: Bootstrapping of Envoy as a service sidecar proxy within a Consul Multi-DC environment with Mesh Gateways is the same as described in the Consul Single-DC case.

The main difference between the Multi-DC and the Single-DC scenario is that the Envoy cluster for the destination service endpoints will point to the Mesh Gateway when the destination service resides in a remote data center and not to the actual service endpoint itself, as in the Single-DC case. This will be discussed in detail within the next chapters.

Envoy Mesh Gateway Bootstrapping

Because the mesh gateway is also based on Envoy, it can take advantage of the gRPC interface offered by the Consul Client Agent to fetch its dynamic configuration on the fly during runtime.

The Consul Client Agent instantiates the Mesh Gateway and registers it as a service. It configures the Mesh Gateway according to the definition with a public listener port which is used for all Mesh Gateway communications to service sidecar proxies or to remote Mesh Gateways.

The high-level workflow of bootstrapping Envoy as a Mesh Gateway is as follows.

The Operator initiates the bootstrap process by:

- Creating the initial Envoy config with **`consul connect envoy -mesh-gateway`** command
- Start and supervise Envoy process

Tasks handled by the local Consul Client Agent:

- Configure Envoy through gRPC

Further maintenance tasks of Consul Client Agent during the lifetime of the Mesh Gateway:

- Update Envoy cluster configuration if service endpoint status changes

If you compare this high-level workflow of bootstrapping and lifecycling a Mesh Gateway to the process described earlier on how an Envoy based sidecar proxy is bootstrapped and lifecycled, there are some differences in the actual creation and lifecycling, even though both instance types—the Mesh Gateway and the sidecar proxy instances—are Envoy-based.

This is due to the fact that the Mesh Gateway is located at a different point in the network and is being used as an interconnection point between federated Consul Datacenters, hence the need for a different configuration and security posture compared to a sidecar proxy instance.

The main differences between an Envoy based Mesh Gateway and an Envoy-based sidecar proxy instance are:

- A Mesh Gateway will not be equipped with a X.509 certificate for service-to-service authentication making it impossible for a Mesh Gateway to directly connect to a service residing in the service mesh
- A Mesh Gateway only operates on the TCP layer and is not involved in the mutual TLS connection between service sidecar proxies, allowing for an end-to-end authentication, encryption and authorization of service-to-service traffic

- A Mesh Gateway inspects the TLS SNI header to determine where to route traffic
- A Mesh Gateways does not intercept TLS, nor decrypt traffic between service sidecar proxy instances
- All encryption tasks will be handled end-to-end between service sidecar proxy instances even though traffic might traverse a Mesh Gateway to bridge multiple Consul Datacenters
- Mesh Gateways do not enforce Consul Connect intentions and therefore the local Consul Client Agent will not cache any intentions locally

To start Envoy as a Mesh Gateway the workflow is similar to starting Envoy as a sidecar proxy instance:

The first thing that needs to happen is to generate a bootstrap configuration for Envoy. The **`consul connect envoy -mesh-gateway`** command can be used to generate an Envoy bootstrap configuration. The command can either directly start Envoy with that configuration or can write it to disk so that Envoy can be started separately with that configuration, for example using Docker.

The bootstrap config is populated with connection details for the local Consul Client Agent, such as the gRPC port to connect to, which will be used to fetch Envoy's final configuration.

After starting Envoy with this bootstrap configuration, Envoy will initiate a connection to the local Consul Client Agent's gRPC port (TCP 8502 per default, if enabled. Disabled per default) to receive its necessary configuration.

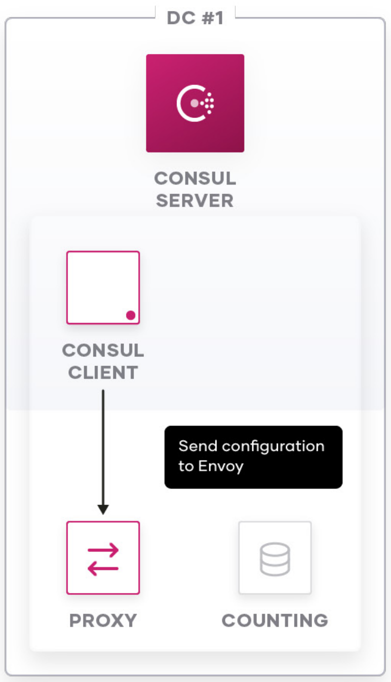
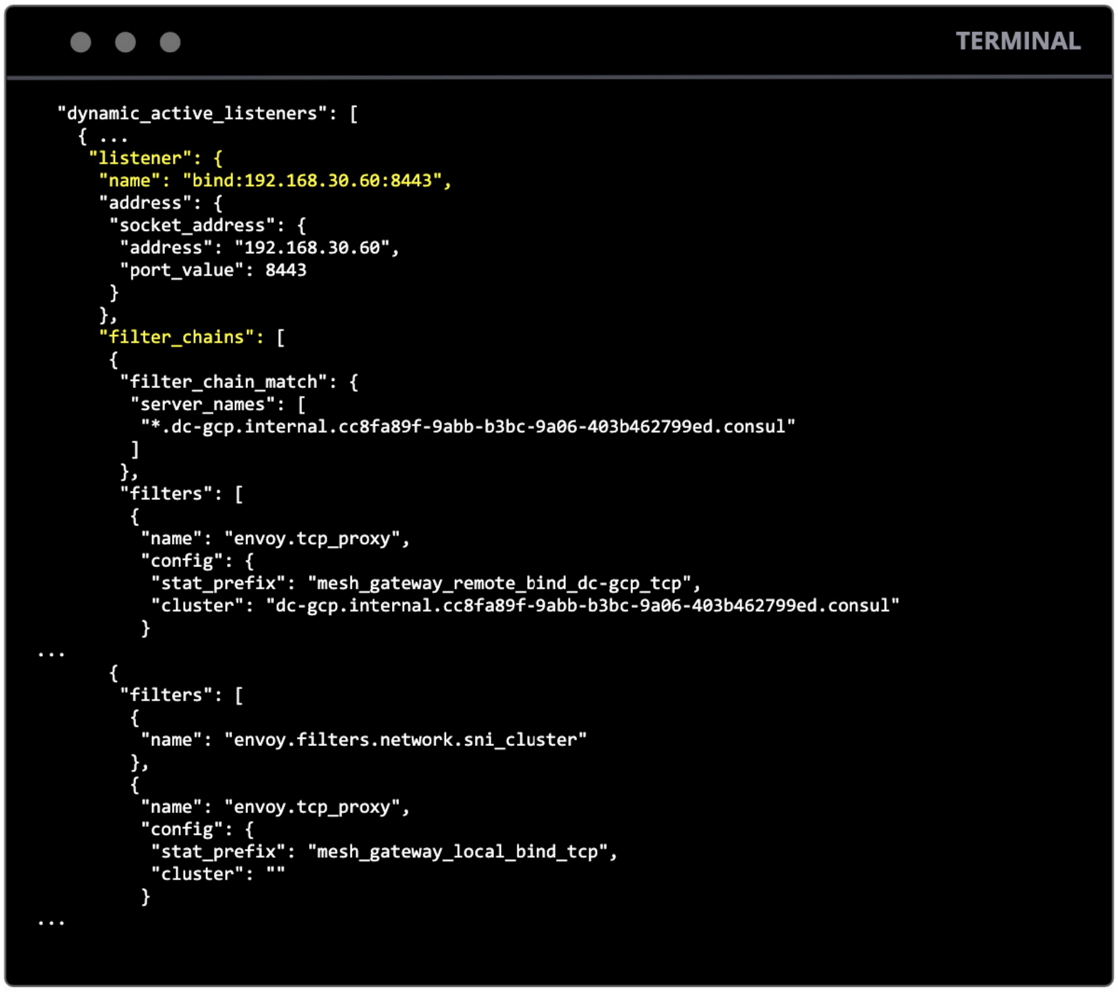


Figure 65: Consul Client configuring Envoy Mesh Gateway through local gRPC connection.

This Envoy config bundle sent through the gRPC channel from the Consul Client to the Envoy Mesh Gateway includes:

- Listener configuration for communication with service sidecar proxies and remote Mesh Gateways
    - Compared to Envoy sidecar proxy instances, a Mesh Gateway only has a single listener configured
- SNI "filter chains" to map inspected SNI header information to either remote Mesh Gateways or service proxies in local Consul Datacenter

The resulting data-plane configuration of a Mesh Gateway will have the dynamic listener configured as well as wildcard SNI routes (filter_chains) for services residing in a remote Consul Datacenter pointing to the respective remote Mesh Gateway of the remote Consul Datacenter:



```
"dynamic_active_listeners": [
 { ...
  "listener": {
   "name": "bind:192.168.30.60:8443",
   "address": {
    "socket_address": {
     "address": "192.168.30.60",
     "port_value": 8443
    }
   },
   "filter_chains": [
    {
     "filter_chain_match": {
      "server_names": [
       "*.dc-gcp.internal.cc8fa89f-9abb-b3bc-9a06-403b462799ed.consul"
      ]
     },
     "filters": [
      {
       "name": "envoy.tcp_proxy",
       "config": {
        "stat_prefix": "mesh_gateway_remote_bind_dc-gcp_tcp",
        "cluster": "dc-gcp.internal.cc8fa89f-9abb-b3bc-9a06-403b462799ed.consul"
       }
...
    {
     "filters": [
      {
       "name": "envoy.filters.network.sni_cluster"
      },
      {
       "name": "envoy.tcp_proxy",
       "config": {
        "stat_prefix": "mesh_gateway_local_bind_tcp",
        "cluster": ""
       }
...
```

Figure 66: Extract of Mesh Gateway Envoy data-plane listener and filter chain configuration.

There are different modes available within Consul service mesh to define which services or sidecar proxies leverage a Mesh Gateway. It is possible to define that traffic should be routed through a local mesh gateway to a remote mesh gateway (**`mode: local`**), or that traffic may be routed directly from the sidecar proxy to a remote mesh gateway (**`mode: remote`**)."

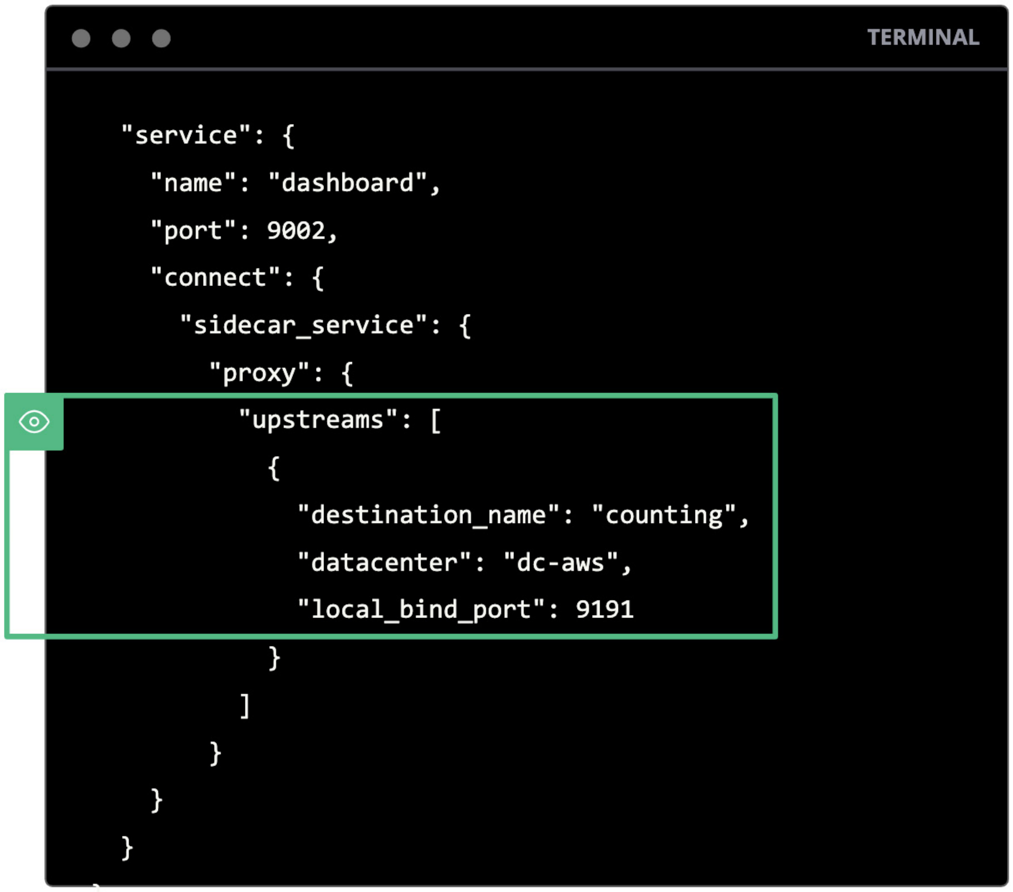
Throughout the following chapters only **`mode: local`** is discussed, but one can easily make the transition to how traffic behaves in **`mode: remote`** by simply not considering the first hop Mesh Gateway in the following examples.

Packet walk Consul Service Mesh Multi-DC with Mesh Gateway

In the following examples it is assumed that the source Dashboard service and the upstream Counting service do not reside in the same Consul Datacenter, but in federated Consul Datacenters.

To allow the Dashboard service to make calls to the Counting service through the service mesh, one of the prerequisites is that one either needs to define the remote Counting service as an upstream service within the Dashboard service definition or to configure an according failover behavior for the Counting service, to allow the service mesh to automatically discover service endpoints in a federated Consul Datacenter when there are no healthy instances available in the local Consul Datacenter.

The first option would look like this, where the Counting service within the 'dc-aws' Consul Datacenter is explicitly defined as an upstream for the Dashboard service.



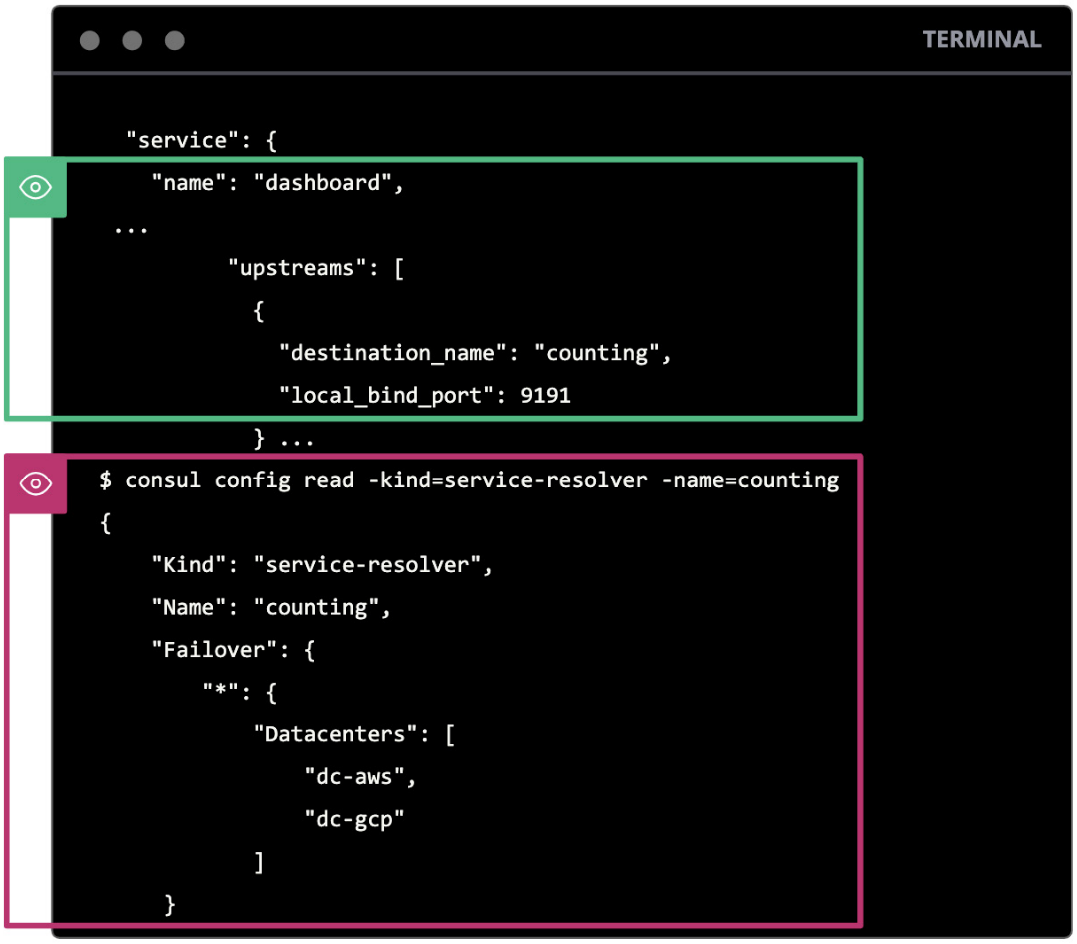
```
"service": {
  "name": "dashboard",
  "port": 9002,
  "connect": {
    "sidecar_service": {
      "proxy": {
        "upstreams": [
          {
            "destination_name": "counting",
            "datacenter": "dc-aws",
            "local_bind_port": 9191
          }
        ]
      }
    }
  }
```

Figure 67: Explicit definition of upstream service in remote data center.

Besides explicitly pointing the upstream service to a remote data center, all other considerations in terms of local sidecar proxy listeners resulting from this configuration, are the same as discussed earlier.

Another option to leverage services in a federated Consul Datacenter as an upstream service would be to define a failover policy for the given upstream service.

The following configuration example does not define the upstream service in a remote data center explicitly, but defines a failover policy by means of a service-resolver which automatically routes traffic to a remote Consul Datacenter when there are no healthy instances of the configured upstream service available in the local Consul Datacenter:



```
  "service": {
    "name": "dashboard",
 ...
          "upstreams": [
            {
              "destination_name": "counting",
              "local_bind_port": 9191
            } ...
$ consul config read -kind=service-resolver -name=counting
{
    "Kind": "service-resolver",
    "Name": "counting",
    "Failover": {
        "*": {
            "Datacenters": [
                "dc-aws",
                "dc-gcp"
            ]
       }
```

Figure 68: Definition of upstream service by name with according failover behavior in service-resolver.

The defined list of data centers is the order of Consul Datacenters to discover the Counting service in.

The initial steps for initiating service-to-service communication between the Dashboard service residing in the local Consul Datacenter and the Counting service residing in a remote Consul Datacenter are basically the same as in the Single-DC example discussed earlier on:

The Dashboard service will initiate a connection to the Counting service by connecting to its sidecar proxy on the configured upstream TCP port (TCP 9191 in our example).

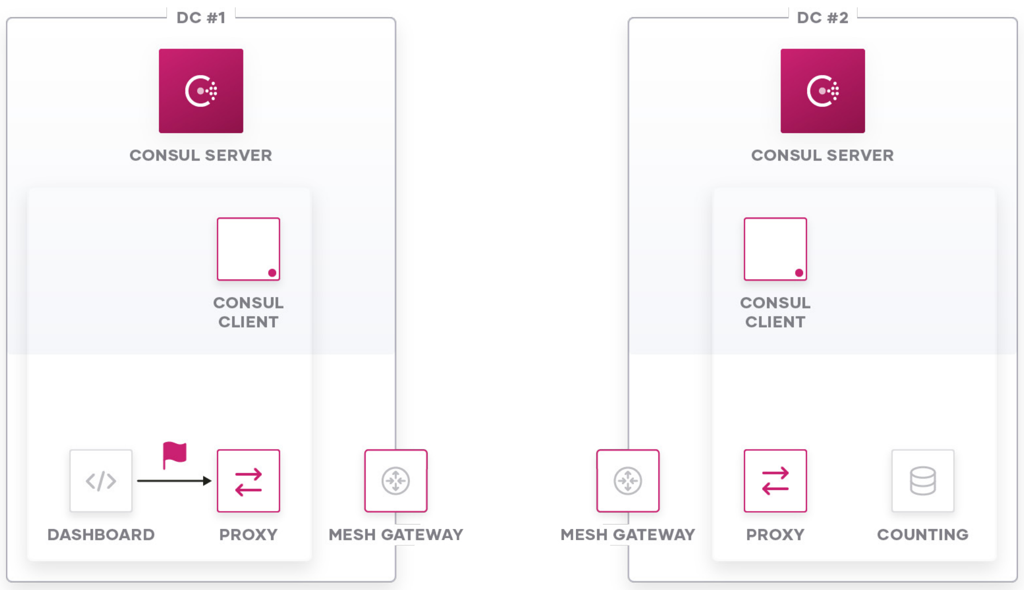


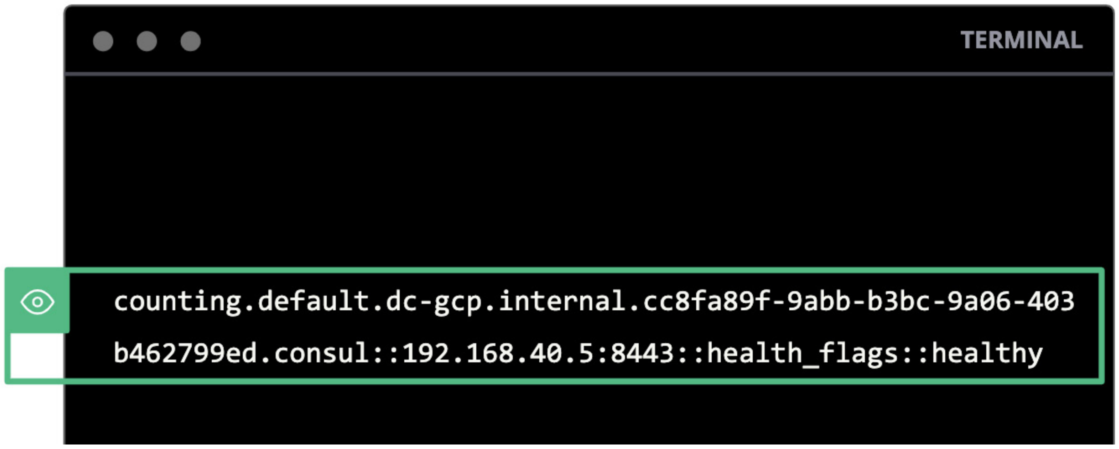
Figure 69: Session initialization of Dashboard service to its local sidecar proxy.

As in the Single-DC example, the sidecar proxy instance of the Dashboard service will map the incoming TCP session on its dynamic listener TCP port 9191 to a dynamic route configuration, which will point to the pre-populated endpoint cluster for the Counting service.

At this point, the session setup differs from the single-DC example.

In the single-DC example, the counting cluster was pre-populated with all available Counting service endpoints within the local Consul Datacenter and Envoy chose one of these endpoints to connect to.

In the multi-DC scenario – either when there are no healthy instances of the Counting service available within the local Consul Datacenter, or when the upstream is explicitly configured to resolve to a different Consul Datacenter, the Envoy cluster contains the address of the mesh gateway.

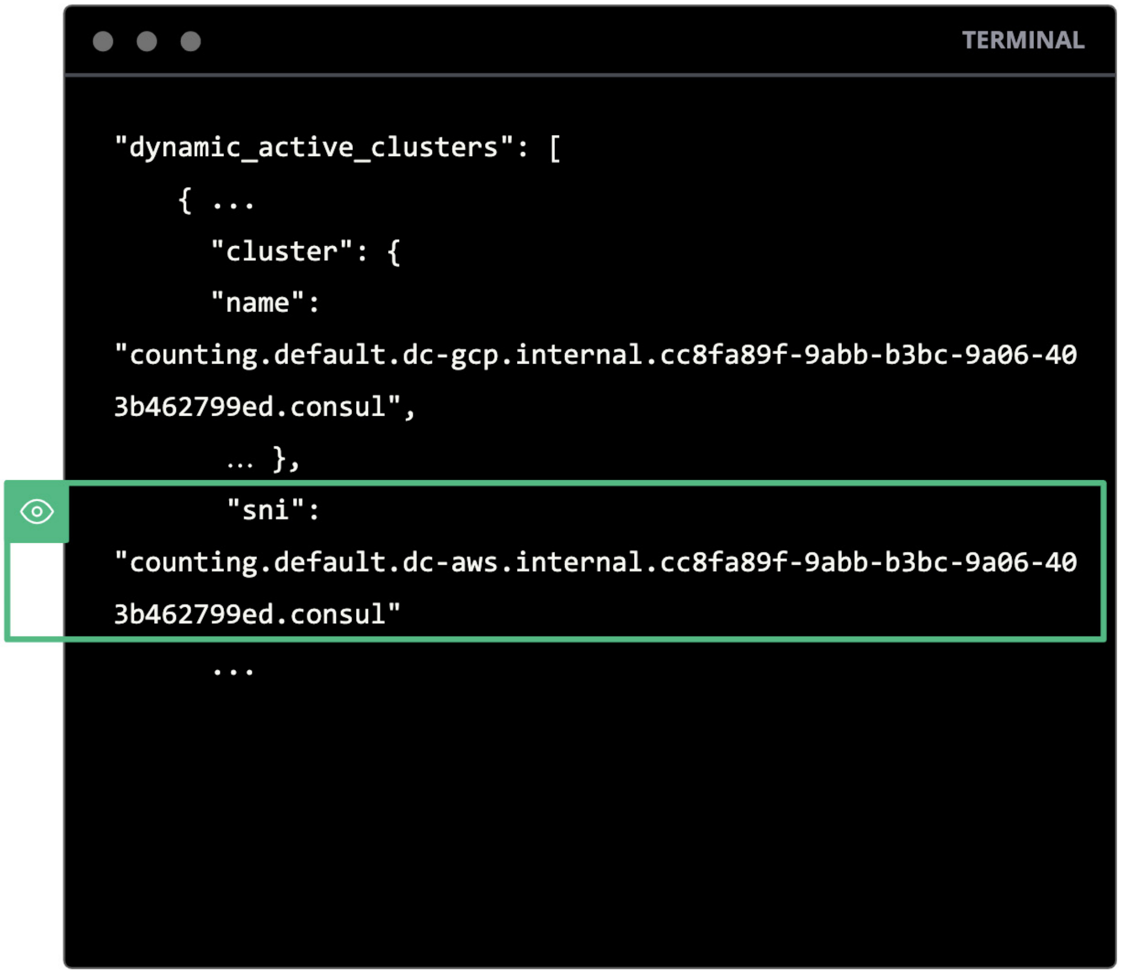


```
counting.default.dc-gcp.internal.cc8fa89f-9abb-b3bc-9a06-403
b462799ed.consul::192.168.40.5:8443::health_flags::healthy
```

Figure 70: Associated Envoy cluster containing local Mesh Gateway.

Remark: There can be more than one Mesh Gateway available in production deployments.

Additionally, the SNI header configuration for the Counting cluster will now have an SNI header mapping to the remote Consul Datacenter. This header will be set accordingly by the source sidecar proxy and will be evaluated by the Mesh Gateway later on.



```
"dynamic_active_clusters": [
    { ...
      "cluster": {
      "name":
"counting.default.dc-gcp.internal.cc8fa89f-9abb-b3bc-9a06-40
3b462799ed.consul",
       … },
       "sni":
"counting.default.dc-aws.internal.cc8fa89f-9abb-b3bc-9a06-40
3b462799ed.consul"
      ...
```

Figure 71: SNI header configuration for "counting" EDS cluster including remote DC name in SNI.

Now that the local sidecar proxy of the Dashboard service knows where to send traffic to and which SNI header to set during the TLS session setup, it will start a TLS session towards the Mesh Gateway:

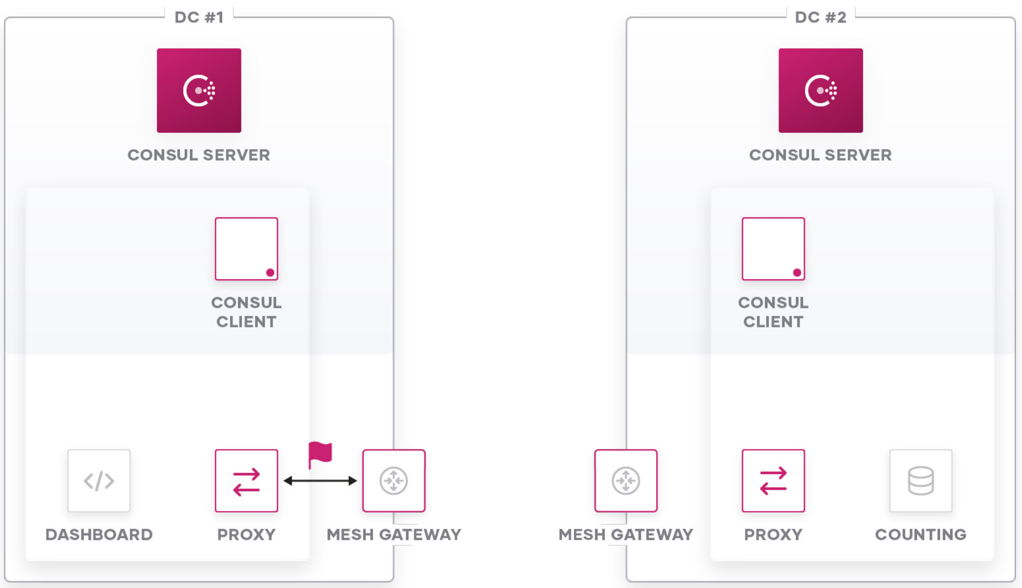
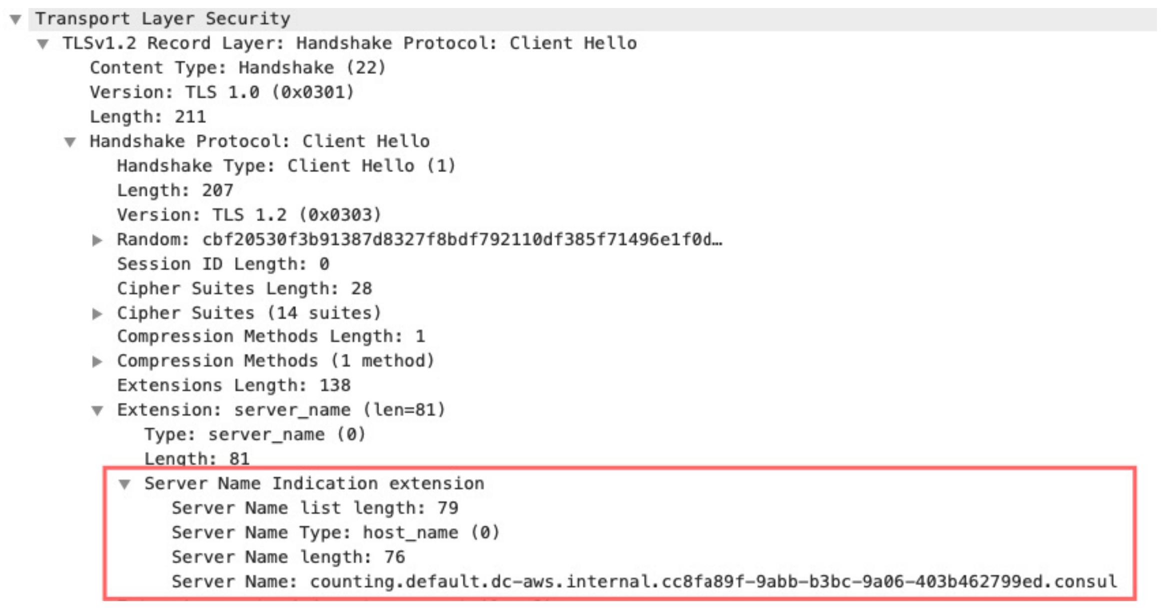


Figure 72: Sidecar proxy to local Mesh Gateway session initialization.



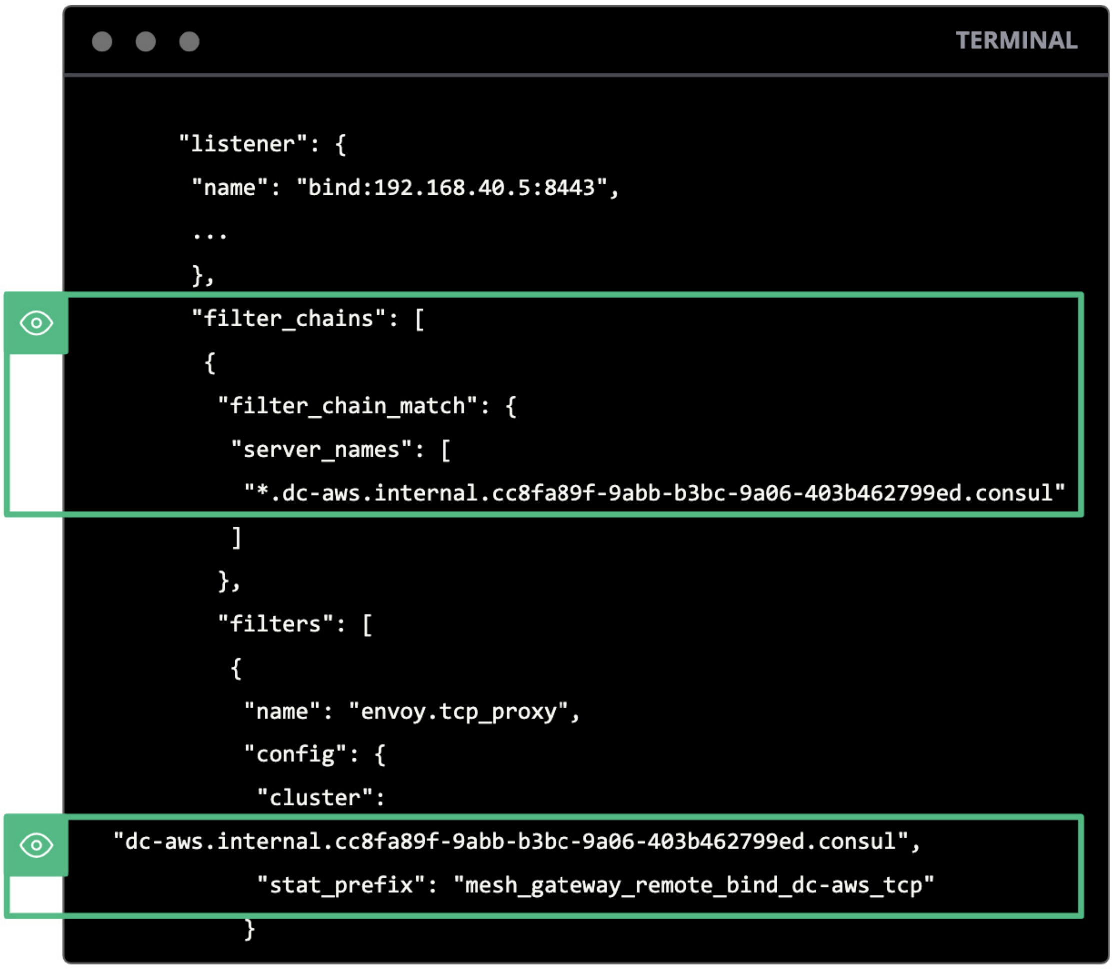
```
▼ Transport Layer Security
  ▼ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 211
    ▼ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 207
        Version: TLS 1.2 (0x0303)
      ▶ Random: cbf20530f3b91387d8327f8bdf792110df385f71496e1f0d…
        Session ID Length: 0
        Cipher Suites Length: 28
      ▶ Cipher Suites (14 suites)
        Compression Methods Length: 1
      ▶ Compression Methods (1 method)
        Extensions Length: 138
      ▼ Extension: server_name (len=81)
          Type: server_name (0)
          Length: 81
        ▼ Server Name Indication extension
            Server Name list length: 79
            Server Name Type: host_name (0)
            Server Name length: 76
            Server Name: counting.default.dc-aws.internal.cc8fa89f-9abb-b3bc-9a06-403b462799ed.consul
```

Figure 73: TLS handshake Client Hello including SNI header.

The underlying TCP session will be destined for the mesh gateway's listener IP and TCP port, so the gateway will accept the TCP connection.

Remember that a Mesh Gateway is not equipped with a X.509 service certificate, hence it will not be able to intercept the TLS session.

During the bootstrapping (or later on through the gRPC channel between Envoy Mesh Gateway and Consul Client Agent) the Mesh Gateway was equipped with SNI filter_chains bound to its listener:


TERMINAL

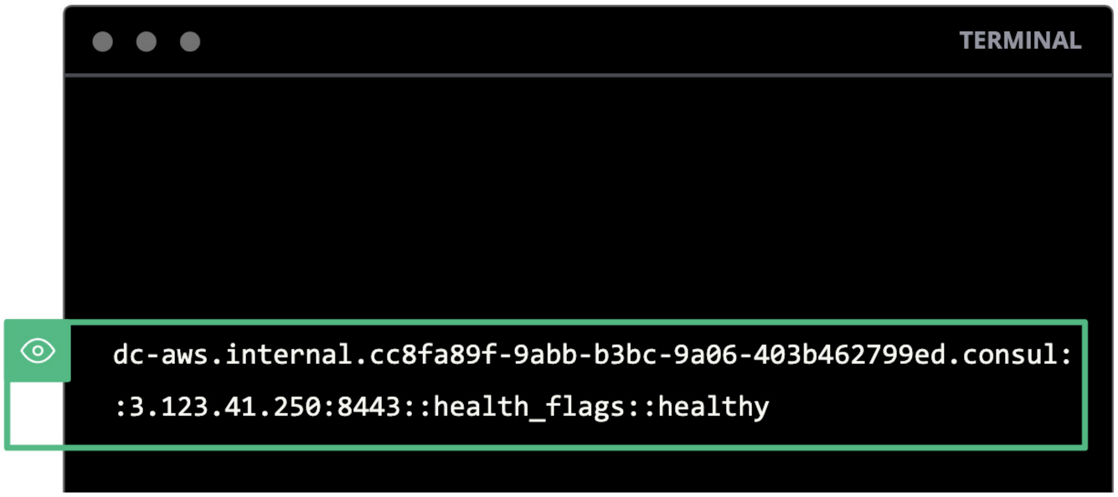
```
"listener": {
 "name": "bind:192.168.40.5:8443",
 ...
 },
 "filter_chains": [
  {
   "filter_chain_match": {
    "server_names": [
     "*.dc-aws.internal.cc8fa89f-9abb-b3bc-9a06-403b462799ed.consul"
    ]
   },
   "filters": [
    {
     "name": "envoy.tcp_proxy",
     "config": {
      "cluster":
"dc-aws.internal.cc8fa89f-9abb-b3bc-9a06-403b462799ed.consul",
      "stat_prefix": "mesh_gateway_remote_bind_dc-aws_tcp"
     }
```

Figure 74: Envoy filter chain configuration on Mesh Gateway.

After accepting the TCP connection from the Dashboard sidecar proxy instance, the Mesh Gateway will evaluate the TLS "Client Hello" (as part of the connection attempt) for SNI headers and match the SNI header against the configured filter_chains.

For each federated Consul Datacenter, there is a wildcard (*) SNI filter_chain installed in the Mesh Gateway pointing to a corresponding Envoy cluster for the respective remote data center.

After mapping the SNI header against the configured "SNI routes" and finding the respective cluster, the Mesh Gateway finds the public IP addresses of the remote Mesh Gateways within the pre-populated cluster:



```
dc-aws.internal.cc8fa89f-9abb-b3bc-9a06-403b462799ed.consul:
:3.123.41.250:8443::health_flags::healthy
```

Figure 75: Associated Envoy cluster containing remote Mesh Gateway.

The local Mesh Gateway will then initiate a new TCP connection to the public listener of a remote Mesh Gateway it found in the respective cluster configuration:

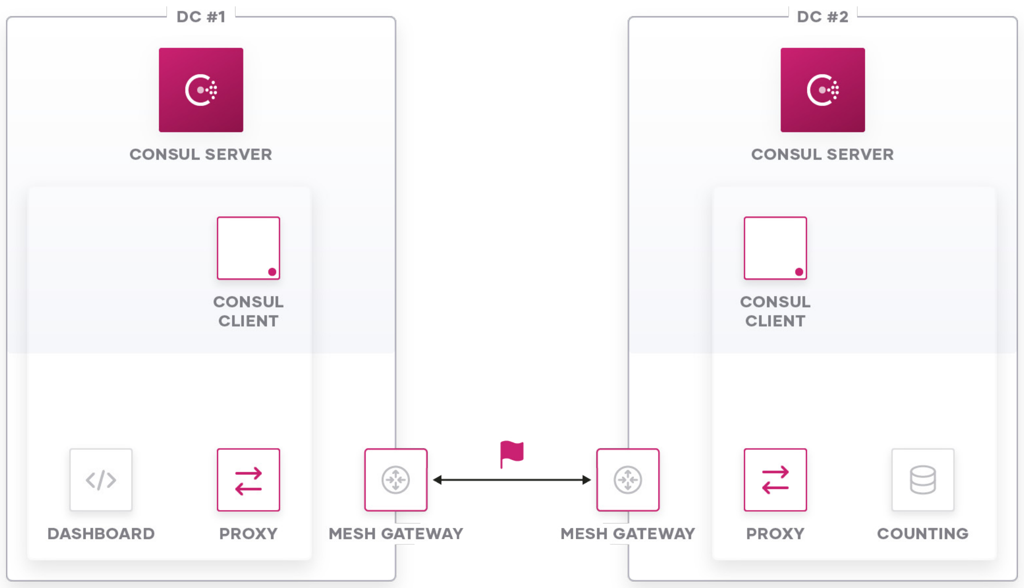


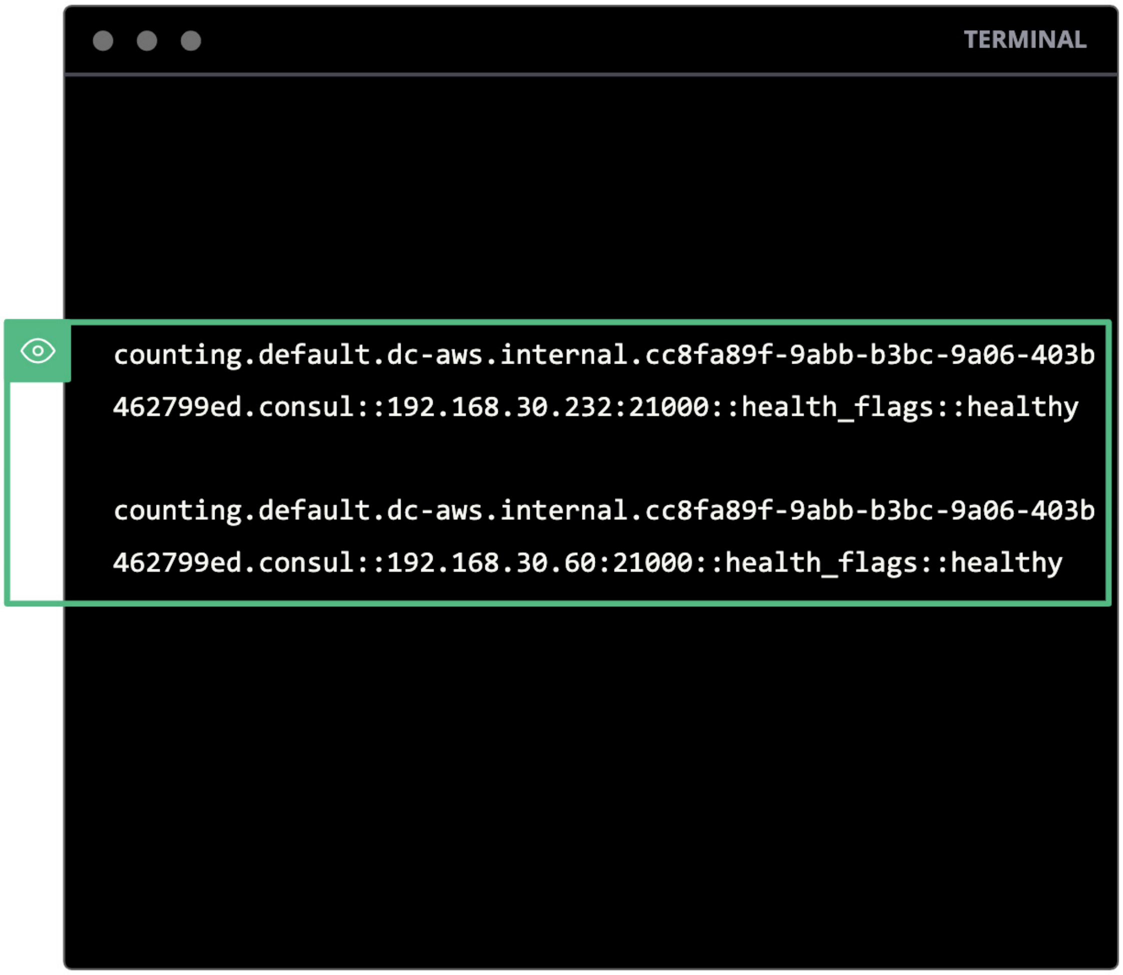
Figure 76: Session initialization of local Mesh Gateway to remote Mesh Gateway.

The TLS connection attempt from the Dashboard sidecar proxy towards the local Mesh Gateway will simply be copied and forwarded from one TCP connection (Dashboard sidecar proxy ← → local Mesh Gateway) to the other (local Mesh Gateway ← → remote Mesh Gateway).

As part of this TCP splicing between the two existing connections, every payload above TCP OSI layer (Layer 4) will be spliced between those sessions. This also includes the initial TLS "Client Hello" including its SNI header originated by the Dashboard sidecar proxy instance.

The remote Mesh Gateway will accept the TCP session attempt of the local mesh gateway, however it is not able to intercept the TLS session as it has no installed X.509 service certificate.

It will inspect the TLS traffic and perform a mapping between the SNI header set within the TLS "Client Hello" from the Dashboard sidecar proxy and map it against its pre-configured dynamic clusters. One of these clusters matches the requested SNI header and will be pre-populated with all available and healthy endpoints of the Counting service within this Mesh Gateway's Consul Datacenter:



```
counting.default.dc-aws.internal.cc8fa89f-9abb-b3bc-9a06-403b
462799ed.consul::192.168.30.232:21000::health_flags::healthy

counting.default.dc-aws.internal.cc8fa89f-9abb-b3bc-9a06-403b
462799ed.consul::192.168.30.60:21000::health_flags::healthy
```

Figure 77: Associated Envoy EDS cluster containing Counting service endpoints.

The Mesh Gateway will choose one of these healthy service endpoints and start a TCP session with this endpoint.

The same TCP splicing logic as discussed before will take place between the service proxy and the Mesh Gateway will splice traffic from the previously established TCP session with the other Mesh Gateway to the TCP session just established with the Counting service sidecar proxy:

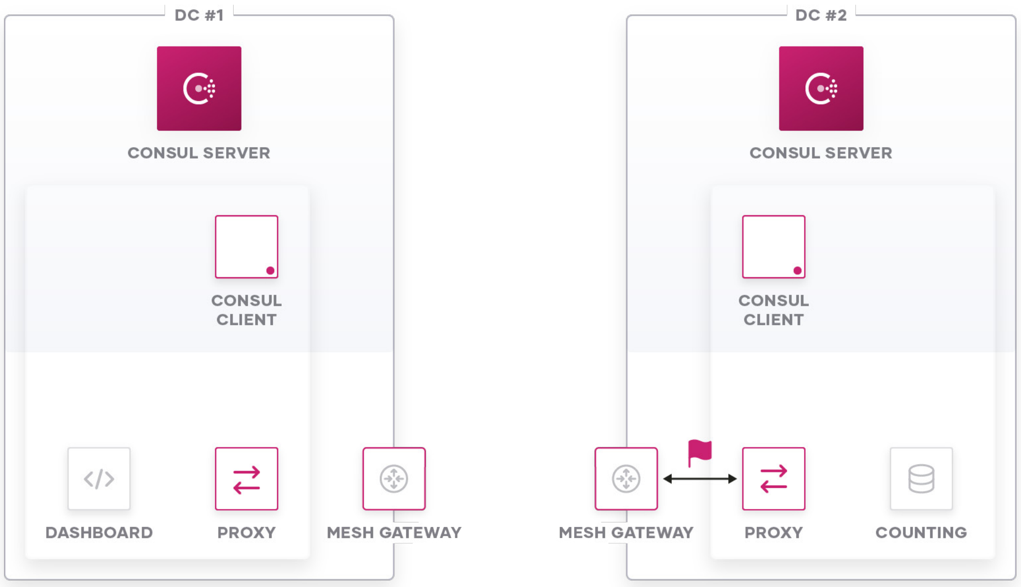
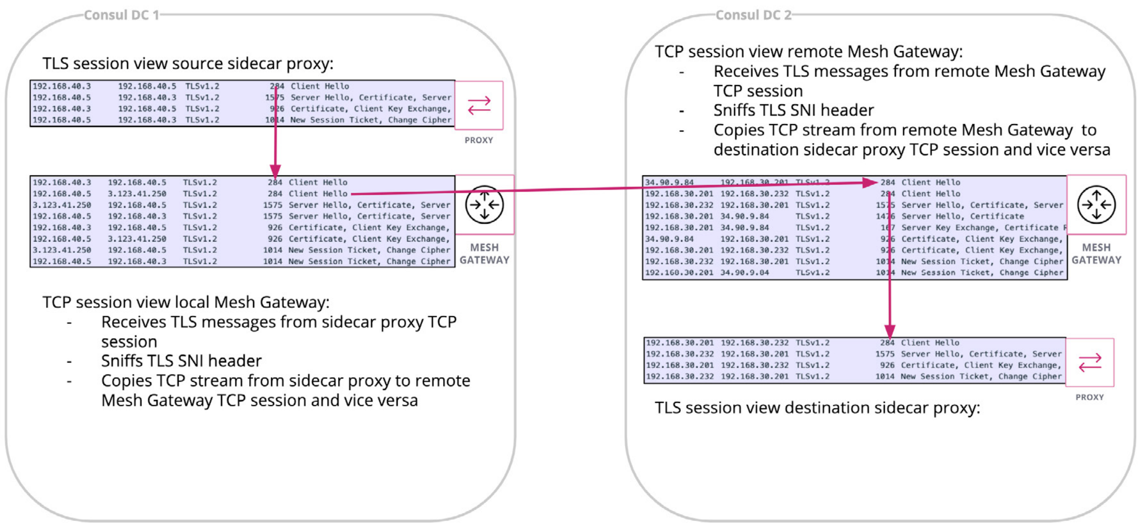


Figure 78: Remote Mesh Gateway to destination sidecar proxy session initialization.

The following might help to better illustrate the different TCP sessions established as part of the end-to-end service communication:



TLS session view source sidecar proxy:

```
192.168.40.3     192.168.40.5  TLSv1.2          284 Client Hello
192.168.40.5     192.168.40.3  TLSv1.2         1575 Server Hello, Certificate, Server
192.168.40.3     192.168.40.5  TLSv1.2          926 Certificate, Client Key Exchange,
192.168.40.5     192.168.40.3  TLSv1.2         1014 New Session Ticket, Change Cipher
```

```
192.168.40.3   192.168.40.5   TLSv1.2          284 Client Hello
192.168.40.5   3.123.41.250   TLSv1.2          284 Client Hello
3.123.41.250   192.168.40.5   TLSv1.2         1575 Server Hello, Certificate, Server
192.168.40.5   192.168.40.3   TLSv1.2         1575 Server Hello, Certificate, Server
192.168.40.3   192.168.40.5   TLSv1.2          926 Certificate, Client Key Exchange,
192.168.40.5   3.123.41.250   TLSv1.2          926 Certificate, Client Key Exchange,
3.123.41.250   192.168.40.5   TLSv1.2         1014 New Session Ticket, Change Cipher
192.168.40.5   192.168.40.3   TLSv1.2         1014 New Session Ticket, Change Cipher
```

TCP session view local Mesh Gateway:
- Receives TLS messages from sidecar proxy TCP session
- Sniffs TLS SNI header
- Copies TCP stream from sidecar proxy to remote Mesh Gateway TCP session and vice versa

TCP session view remote Mesh Gateway:
- Receives TLS messages from remote Mesh Gateway TCP session
- Sniffs TLS SNI header
- Copies TCP stream from remote Mesh Gateway to destination sidecar proxy TCP session and vice versa

```
34.90.9.84     192.168.30.201 TLSv1.2          284 Client Hello
192.168.30.201 192.168.30.232 TLSv1.2          284 Client Hello
192.168.30.232 192.168.30.201 TLSv1.2         1575 Server Hello, Certificate, Server
192.168.30.201 34.90.9.84     TLSv1.2         1476 Server Hello, Certificate
192.168.30.201 34.90.9.84     TLSv1.2          167 Server Key Exchange, Certificate R
34.90.9.84     192.168.30.201 TLSv1.2          926 Certificate, Client Key Exchange,
192.168.30.201 192.168.30.232 TLSv1.2          926 Certificate, Client Key Exchange,
192.168.30.232 192.168.30.201 TLSv1.2         1014 New Session Ticket, Change Cipher
192.168.30.201 34.90.9.84     TLSv1.2         1014 New Session Ticket, Change Cipher
```

```
192.168.30.201 192.168.30.232 TLSv1.2          284 Client Hello
192.168.30.232 192.168.30.201 TLSv1.2         1575 Server Hello, Certificate, Server
192.168.30.201 192.168.30.232 TLSv1.2          926 Certificate, Client Key Exchange,
192.168.30.232 192.168.30.201 TLSv1.2         1014 New Session Ticket, Change Cipher
```

TLS session view destination sidecar proxy:

Figure 79: Splicing between TCP sessions.

The Counting sidecar proxy will finally receive the TLS session attempt from the Dashboard sidecar proxy and will reply accordingly.

As the Mesh Gateways involved in this connection setup simply forward TLS communication between the two sidecar proxy endpoints, the mutual TLS session establishment between the sidecar proxy instances to authenticate and establish an encrypted channel is exactly the same process as described in the Single-DC example.

The final authorization step within the session establishment will be the same as in the single DC example: The destination Counting sidecar proxy will extract the SPIFFE identities out of the X.509 service certificates and send an AuthZ request towards the local Consul Client Agent to verify whether the connection is allowed.

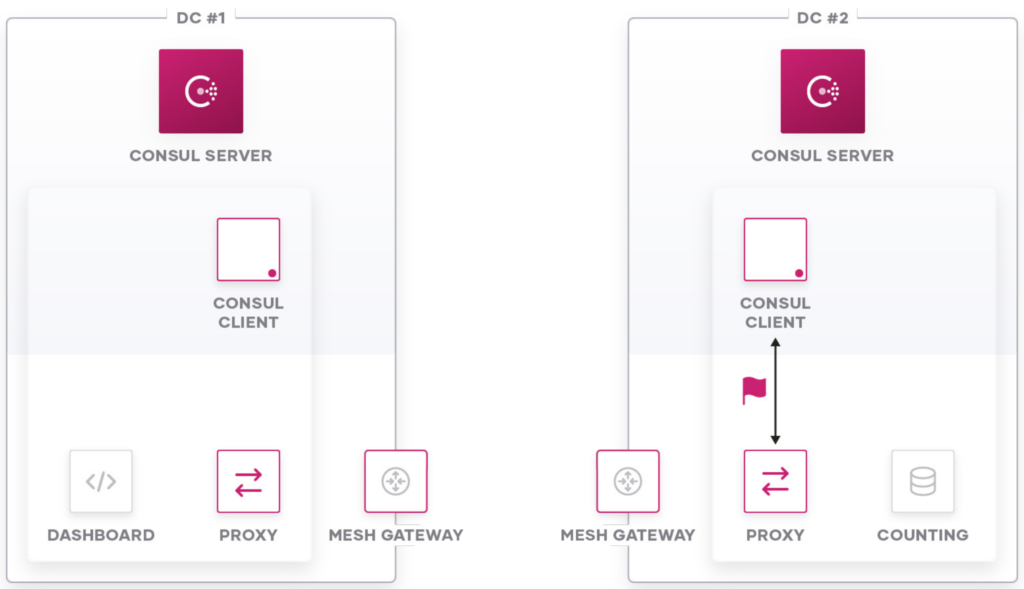


Figure 80: AuthZ callback from Envoy to Consul Client Agent.

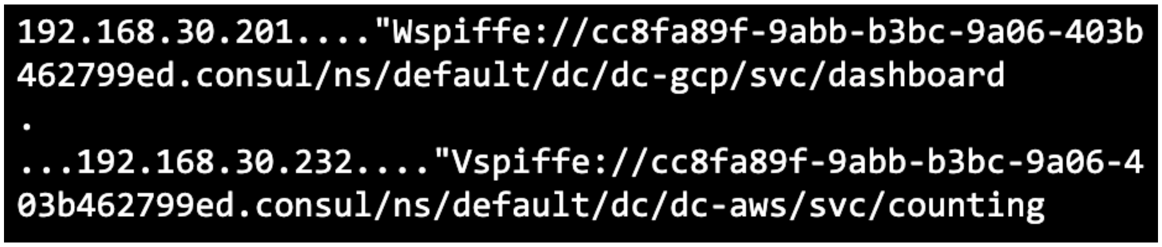

```
192.168.30.201...."Wspiffe://cc8fa89f-9abb-b3bc-9a06-403b
462799ed.consul/ns/default/dc/dc-gcp/svc/dashboard
.
...192.168.30.232...."Vspiffe://cc8fa89f-9abb-b3bc-9a06-4
03b462799ed.consul/ns/default/dc/dc-aws/svc/counting
```

Figure 81: Raw SPIFFE IDs presented from Envoy to Consul Client Agent.

One difference to note is that the Consul Datacenter name is encoded in the SPIFFE identity of the service, so you can identify whether a connection attempt comes from the same Consul Datacenter or a federated Consul Datacenter.

Because the Consul Datacenters are federated and have agreed upon the primary data center, there is a single CA and common root of trust for the sidecar proxies. The establishment of a common root of trust is what makes cross-datacenter mutual TLS possible.

Intentions are replicated between Consul Datacenters which allows for centralized intention management and automated failover of services to another Consul Datacenter without the need to keep intentions manually in sync.

Again, there are two possible replies the Envoy sidecar proxy can receive as an answer to its AuthZ request from the local Consul Client Agent:

Case 1: Intention denies Communication between Services

This is the same behavior as in the Consul Single-DC scenario: The destination Counting sidecar proxy will immediately send a TCP reset towards the source of the connection.

Case 2: Intention allows Communication between Services

The destination sidecar proxy instance will perform the same lookups as discussed in the Single-DC case to find the local application listener port and forward application traffic to the destination service.

After the session is successfully established between two service endpoints, there will be an end-to-end encrypted channel between the proxy instances which is routed via the Mesh Gateways.

The following illustration helps to provide a better understanding of the separate TCP sessions used by the gateways, and the end-to-end session used for the application data.

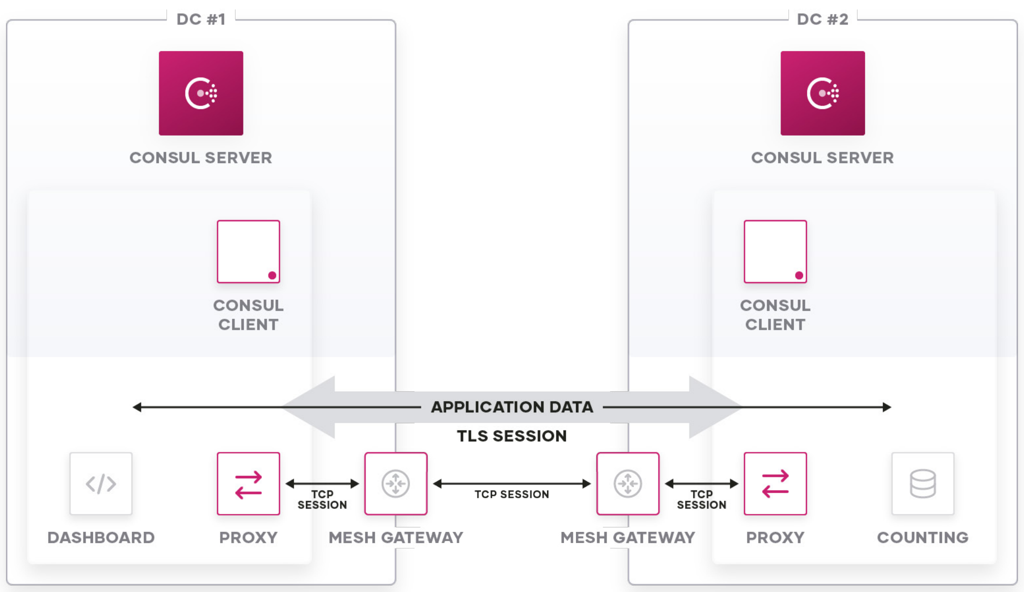


Figure 82: Application data flow vs. mTLS session between sidecar proxy instances vs. TCP sessions

- Application data flows end-to-end between the service instances and data is unchanged for L4 services. For L7 services payload data is unchanged (dependent on L7 Traffic Management configurations, e.g. HTTP path rewrite).
- Application data is TLS encrypted within a single TLS session between the sidecar proxy instances regardless of whether the applications utilize their own encryption. Mesh Gateways will only see encrypted traffic and cannot decrypt the traffic.
- There are five separate TCP sessions involved:
  - 1) Dashboard application → Dashboard sidecar proxy (internal on localhost)
  - 2) Dashboard sidecar proxy → Mesh Gateway DC1
  - 3) Mesh Gateway DC1 → Mesh Gateway DC2
  - 4) Mesh Gateway DC2 → Counting sidecar proxy
  - 5) Counting sidecar proxy → Counting application (internal on localhost)

Interconnecting overlapping IP address space with Consul Mesh Gateways

Consul Mesh Gateways enable interconnecting various Consul DCs with overlapping IP address space without the need to configure NAT or assign unique IP addresses to service endpoints.

As explained in the previous chapter detailing service-to-service communication via mesh gateways, service-to-service sessions do not require end-to-end IP connectivity.

IP connectivity is only used for TCP sessions to the "next-hop" which is a Mesh Gateway from the service sidecar perspective or another Mesh Gateway from a Mesh Gateway perspective.

All traffic routing decisions are based on SNI headers, rather than endpoint IP addresses. All sidecar instances and Mesh Gateways are proxies and not IP routers.

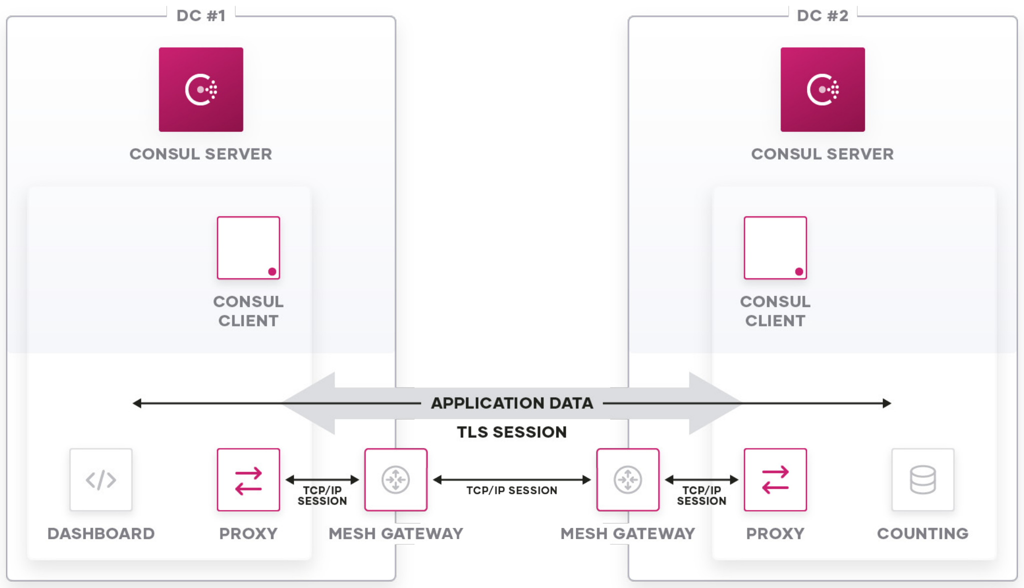
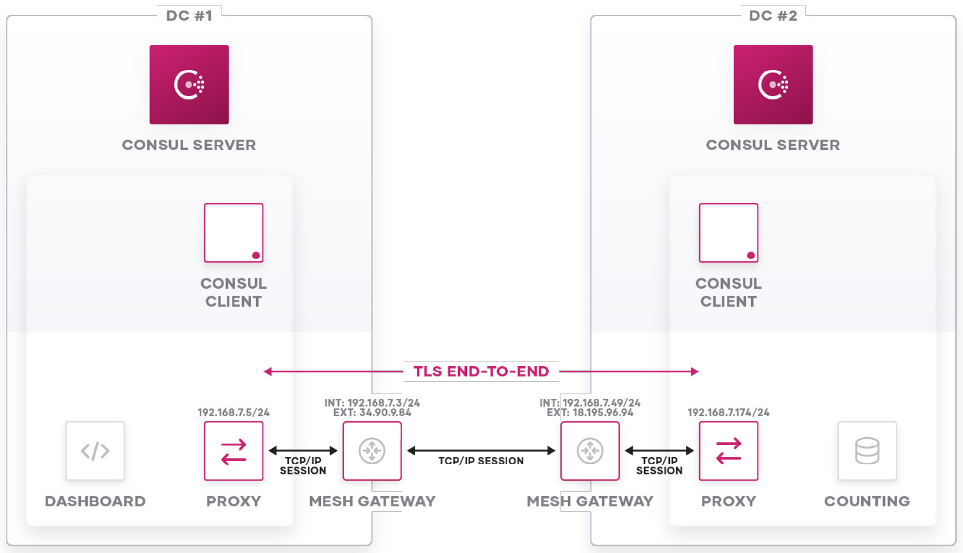
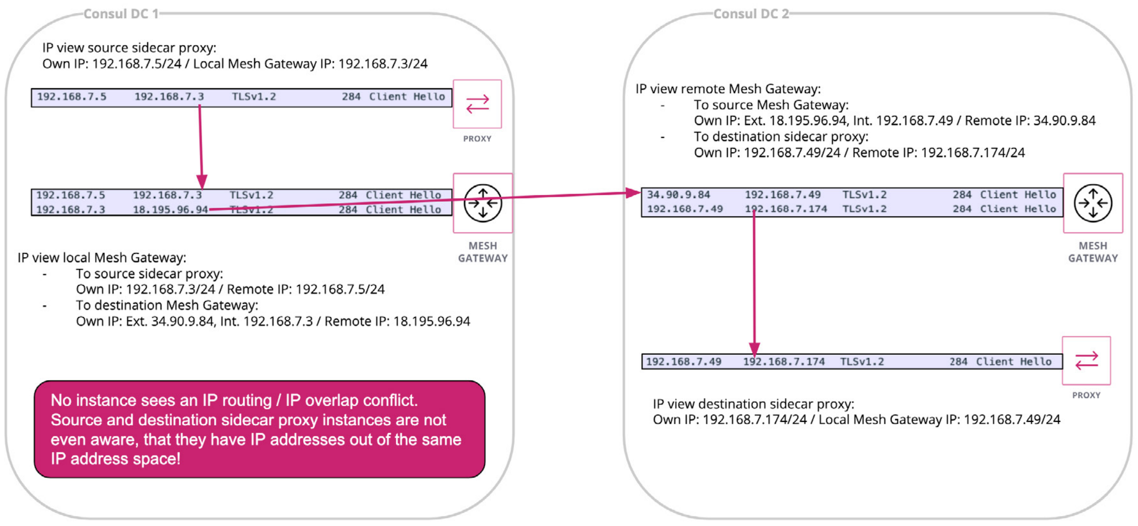


Figure 83: Application data flow vs. mTLS session between sidecar proxy instances vs. TCP sessions.

The following illustrations show the view of the world of various components involved in terms of IP addressing versus TCP session versus end-to-end TLS:



Note: Dashboard proxy is not aware of a possible IP address overlap/conflict it has with remote Counting proxy, as its IP connection only goes to Mesh Gateway IP address, which terminates the TCP/IP session. There is no end-to-end IP connectivity between service endpoints – only hop-by-hop connectivity between mesh gateways and proxy instances.



IP view source sidecar proxy:
Own IP: 192.168.7.5/24 / Local Mesh Gateway IP: 192.168.7.3/24

```
192.168.7.5     192.168.7.3     TLSv1.2           284  Client Hello
```

```
192.168.7.5     192.168.7.3     TLSv1.2           284  Client Hello
192.168.7.3     18.195.96.94    TLSv1.2           284  Client Hello
```

IP view local Mesh Gateway:
- To source sidecar proxy:
  Own IP: 192.168.7.3/24 / Remote IP: 192.168.7.5/24
- To destination Mesh Gateway:
  Own IP: Ext. 34.90.9.84, Int. 192.168.7.3 / Remote IP: 18.195.96.94

No instance sees an IP routing / IP overlap conflict. Source and destination sidecar proxy instances are not even aware, that they have IP addresses out of the same IP address space!

IP view remote Mesh Gateway:
- To source Mesh Gateway:
  Own IP: Ext. 18.195.96.94, Int. 192.168.7.49 / Remote IP: 34.90.9.84
- To destination sidecar proxy:
  Own IP: 192.168.7.49/24 / Remote IP: 192.168.7.174/24

```
34.90.9.84      192.168.7.49    TLSv1.2           284  Client Hello
192.168.7.49    192.168.7.174   TLSv1.2           284  Client Hello
```

```
192.168.7.49    192.168.7.174   TLSv1.2           284  Client Hello
```

IP view destination sidecar proxy:
Own IP: 192.168.7.174/24 / Local Mesh Gateway IP: 192.168.7.49/24

Figure 84: Instance IP addressing vs. TCP sessions vs. TLS session.

Conclusion

We hope this document provides you with a good understanding of how Consul service mesh works and how it can be used to deploy a common service mesh across different cloud and runtime environments.

Consul has many capabilities that enable enterprises to run a single Service Mesh that can span multiple clouds or runtime environments, which we've discussed in detail throughout this white paper. Starting at a foundational level, Consul has proven scalability through it's distributed Control-Plane. To enable the multi-cloud and runtime capabilities, Consul has advanced federation capabilities while still retaining the capability of being a single comprehensive service mesh. This mesh becomes centrally managed across all environments—offering service-to-service authentication, authorization, and encryption on existing platforms without requiring enterprises adopt specific technologies (i.e. Containers/Kubernetes).

Consul service mesh provides a common control plane for application and service networking not only within a data center, but across multiple clouds and runtime platforms, enabling enterprises to adopt a service mesh that supports existing and future application environments.

HashiCorp