Shared with Istio Community

Istio over Zero-VPN

Owner: costin,andracis
Work-Group: Environments, Networking
Short self link: https://goo.gl/efPVv8
Reviewers:

Status: **WIP** | In Review | Approved | Obsolete
Created: 10/2017
Release Version: 0.3, 0.4, 0.5
Approvers: louiscryan [], sven [], rshriram[]

This document is following the Istio Design Review Process.

TL;DR

Define requirements and implementations options for Istio communication in networks without direct L3 connectivity between services - when some resources are behind NATs, firewalls, or have multiple isolated VPNs or no VPN capabilities.

We discuss multiple options, including cases where we do not control the Gateway, i.e. existing gateway or LB is used, but main priority is the case where Istio is used as Gateway.

Overview

As we expand Istio to multiple hybrid environments and multiple clouds we need to support cases where a private network or a VPN between endpoints is not possible or desirable.

Goals:

- P0: Allow mesh expansion in cases where the requirements of “direct L3 IP connection between services” can’t be met.
- P0: All communication must be secure (encrypted & authenticated in both directions).
- P1: Allow admin to selectively control which services are exposed to remote clusters using service-level Istio configs.

Non-Goals:

- Some of the solutions do not provide end-to-end security, and assume the gateway is trusted. Opportunistic encryption and other solutions for e2e will be discussed in separate documents.

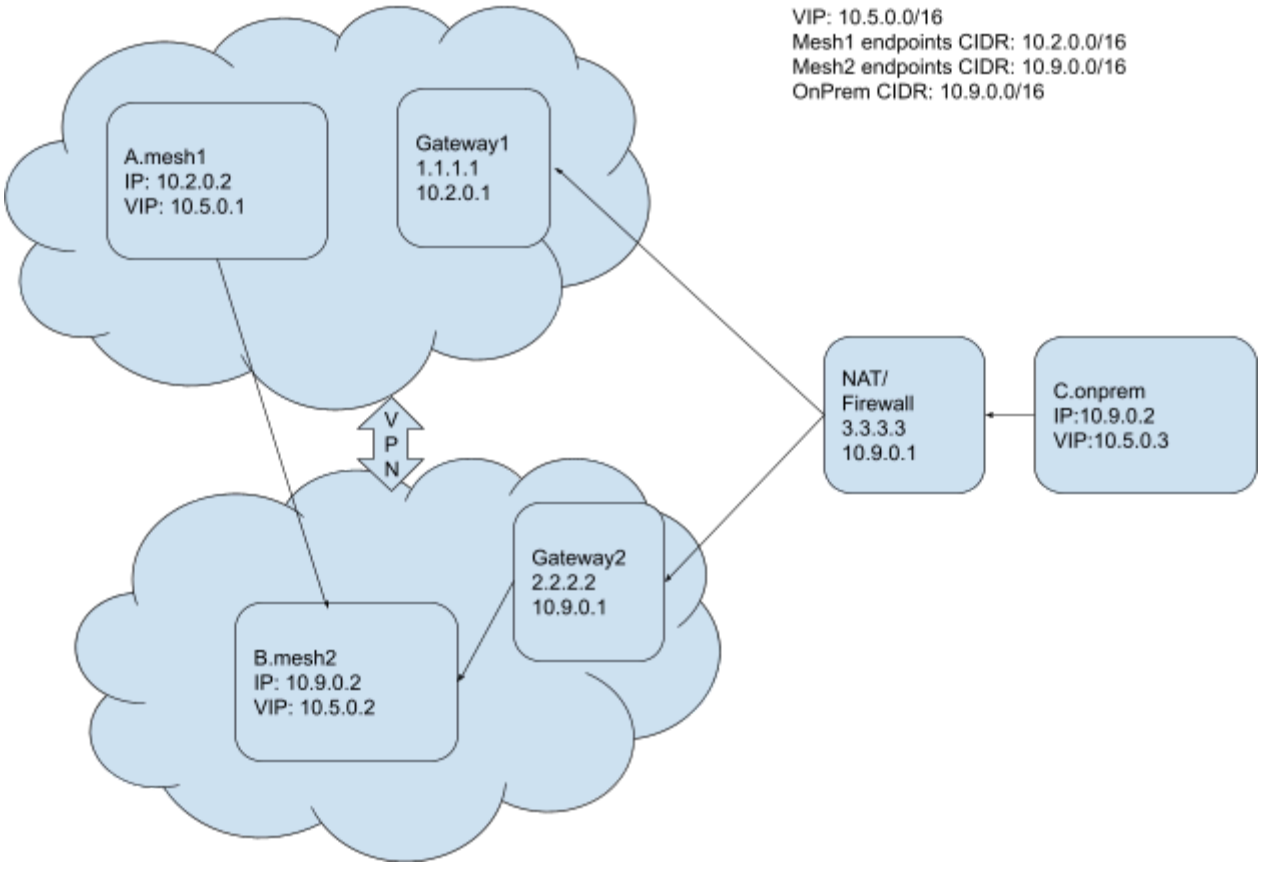
There are many use cases and configurations, each with specific solutions, but in general given

the requirements - NAT or firewall - we must route the requests through some gateway, either the Istio Mesh Load Balancer/Gateway or an equivalent gateway.

This document attempts to list possible options and their use cases - it goes into more details for the simplest solution that we may implement first, while the intent is to gradually support multiple environments and solutions.

In addition to the connectivity itself, the challenge in all discussed solutions is providing a 'split' DNS/naming - there are several ways to handle custom naming, addressed in separate document. In the example below, "C.onprem" will get a "Gateway2" as the list of endpoints for "B.mesh2" service - while "A.Mesh1" will get the actual endpoints of B.mesh2.

Each endpoint has a private address (like 10.9.0.2) and belong to one or more services, identified by a VIP (ClusterIP in case of K8S - can also be a manually allocated address). The VIP is core to Istio functionality[1] - it is required for intercepting the traffic, and acts as a primary key to identify services.



[1] Headless services - where each endpoint can be addressed individually - are not covered in this document, special extensions for this case will be discussed in a future version.

Use cases

The main use case is an environment where there is not a private network, VPN or VPC connecting all the clusters/clouds. It may happen when the cloud provider does not provide VPN, or on-prem has specific firewall or NAT requirements.

A less common case is when some services run outside of the cluster or on-prem ‘production’ network - for example isolated servers. The design also allows the mesh to expand to IoT - like a badge, light, sensor controllers and other equipment. While IoT and ‘isolated servers’ could also use the regular Ingress to make calls into the mesh, it is typically hard for cloud services to make calls to the service ‘outside’ (most of the times it requires mqtt or similar protocol, and a reverse call). In general IoT and ‘isolated’ servers have special security challenges - and are very likely to benefit in many ways from adopting Istio (to be discussed in separate design).

Headless and load balancing

In all solutions discussed, the ‘gateway’ will receive an identifier, using a specific protocol, allowing it to connect to the service and endpoint.

The identifier is set by the client sidecar - and should include information needed to identify both the Service and endpoint.

As a starting proposal, we can use: ENDPOINT.CLUSTER

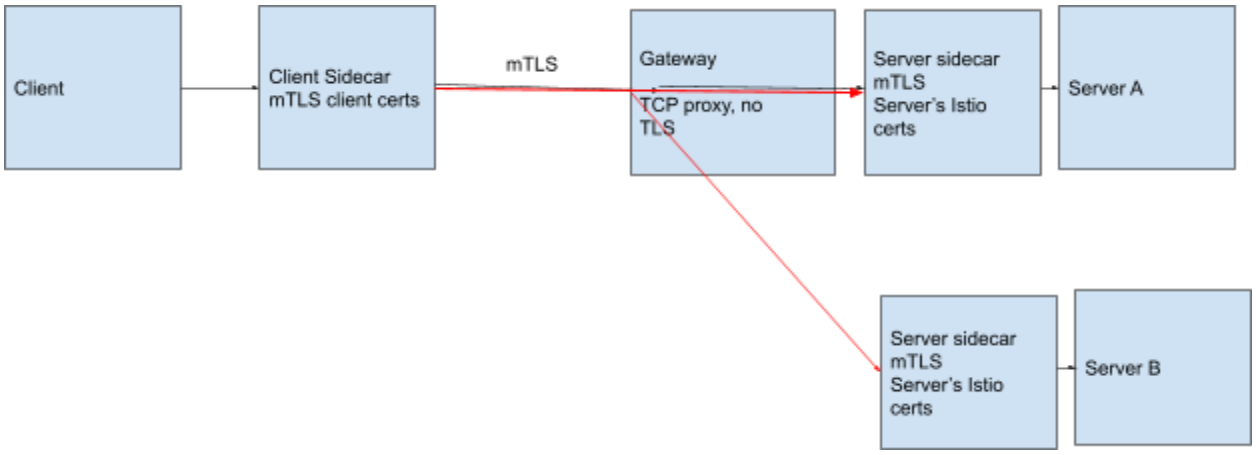
As an optimization we can use the actual (local) endpoint IP, or we can use the endpoint index to match the headless approach.

The load balancing and routing decisions are identical with those over private network/VPN/VPC: the client sidecar only needs to ‘adjust’ the endpoint connection, instead of opening a TCP connection to the endpoint IP it’ll need to open (or reuse) a TCP connection to the Gateway IP, and include the endpoint and cluster info as metadata. Except the extra bytes sent in the first packet (and in some cases the extra multiplexing), the ‘external’ services should be indistinguishable and operate in the same way with same features with the ‘internal’ services.

For “TLS termination in Gateway” options, the gateway may do additional processing (with associated latency/scalability costs) - but so far we don’t have a clear use case.

Options

In all cases, the communication uses mTLS - this is one of the core requirements, it is not acceptable to send unencrypted traffic outside of the private network (and usually even inside the private network).
As such, the Host/authority is not visible until TLS handshake is finished - we must either do the handshake or rely on external signals.



TLS termination in the destination service

This category of options use the Gateway as a simple TCP proxy, terminating the mTLS at the destination service. The Gateway doesn't decrypt or make any security decision.

Pros:

- End-to-end security
- Less overhead on Gateway
- Less complexity
- Gateway doesn't need to be trusted

Cons:

- Routing to the destination service is more complicated and non-standard

P0: TCP proxy with separate ports

The first option is to manually configure the Gateway as TCP proxy, allocating a separate port for each of the service that needs to be exposed outside of the cluster. Not all services need to be exposed - and the configuration can be automated by Istio.

The Gateway acts as a plain TCP proxy - will not terminate TLS, but forward the request to the service using the port. The ports in the gateway don't need to match the service ports.

Pros:

- Very simple - and already works (using manual configuration - only needs to be automated by Pilot)

Cons:

- Port is 16 bits - so max 64k services
- Difficulties in allocating one port per service.

Best for:

- Small number of services

P1: TCP proxy with single port using SNI-based routing

An optimization for the above setup is to use the SNI part of the TLS handshake to determine the service name, and map it to an Istio service. This allows using a single port for all services, which is more scalable and elegant - and requires minimal changes in Envoy compared with plain TCP proxy.

(note: lryan@ suggested the idea, it takes advantage of a little known behavior in the TLS handshake protocol, the fact that ClientHello sends the original hostname in clear text, to allow the server to present the matching certificate).

This approach requires relatively small changes in Envoy TCP proxy code for listener, however the cluster code (for the client's sidecar) must ensure that a SNI header is sent, and use the service name. Normally SNI headers are sent only for connections with a host, and need to be explicitly set (one liner). The Envoy cluster code initiates connections to the IP resolved by EDS, but should have access to the original host.

Pros:

- Doesn't require an additional protocol or changes - SNI is sent anyways at the start of the connection

Cons:

- Non-standard, original solution

- Requires care in parsing the ASN1/DER

Best for:

This is likely the best option for TCP proxy, with minimal changes in application code. This is the recommended solution.

P2: TCP proxy using HTTP2/gRPC streaming

In this mode a dedicated HTTP2 or gRPC interface is used explicitly, with the service ID passed explicitly in the gRPC request or as a HTTP2 header, and the raw bytes encrypted TCP payload forwarded as bi-directional streaming.

Pros:

- Well tested/widely used in some corp environments.
- Uses the native http2/gRPC support in Envoy
- **Multiplexed connections**
- Will work naturally with QUIC, once added to Envoy (and benefit from UDP)
- Allows more flexible metadata
- Easy to integrate with apps supporting Istio natively, without sidecar

Cons:

- Likely to require more time/code to implement and test.
- Not yet a standard solution

TCP proxy with single port using CONNECT or SOCKS

Another option is to extend the TCP proxy in Envoy with HTTP CONNECT and/or SOCKS support. The TLS connection will be tunneled over CONNECT or SOCKS - the gateway will use the destination info (host) to locate the service.

CONNECT is the standard solution of https proxies - support for CONNECT in Envoy will allow a broad set of applications to use Envoy without iptables. SOCKS is the equivalent solution for plain TCP. In both cases the ‘authority’ is sent at the start of the connection.

In this case the sidecar of the external service will need to also be changed to use CONNECT or SOCKS on its cluster connection.

Pros:

- Standard protocols for proxies
- Well known/supported
- May work with unmodified applications

Cons:

?

Best for:

- Existing external apps that implement their own mTLS (without sidecar), and already have socks or connect support.

TCP proxy with single port using HAProxy's PROXY protocol

The PROXY protocol supports both TCP and HTTP, and has additional features not present in CONNECT/SOCKS - but shares the same characteristics. Envoy supports it for listeners (use_proxy_proto) - but it will need special changes since the inbound information can't be trusted, and should only be used for routing. Also it doesn't appear that cluster (outbound) support for PROXY protocol is available.

The CONNECT/SOCKS solution would fit better in cases of unmodified applications doing custom mTLS (without sidecar), since this is built-in/configurable. The PROXY protocol is best fit for integration with other proxies.

Pros:

- Partially supported by Envoy and other proxies
- Like Connect, it can pass additional data (source IP)
- Consistency - normal TCP Gateway may use the same protocol to pass source information.

Cons:

- Less commonly used compared with socks/connect

Best for:

- Infrastructure already using or migrating from haproxy

TCP proxy using Websocket

This is a variant of the above, using the existing Websocket protocol to route the bi-directional stream of bytes (consisting of the encrypted mTLS connection).

Pros:

- No changes to Envoy
- WS is widely used as a TCP proxy
- Routing and metadata are easy
- Will make the Websocket support in Envoy more powerful.

Cons:

- The destination service sidecar must terminate Websocket handshake and transform it into TCP proxy. (this would be a good general feature - there is an Envoy issue asking for this)

Best for:

Apps already using Websocket as a generic transparent proxy.

Foreign LB as TCP proxy

In cases a cluster doesn't allow customization or provides specialized gateway, it is possible to either tunnel traffic over Websockets or similar protocol, terminating mTLS on the actual service.

Pros:

- Works with other LBs, doesn't require Istio gateway

Cons

- We have less control, and may need to work around any limitations

Best for:
In cases where we don't have Istio gateway, or if a foreign LB provides special features missing in Istio gateway, or is required.

NATs with port forwarding

A special case is a service behind a NAT. This can be covered by any of the solutions above, but the gateway will need to open a port forward in the NAT. The main difference is that the gateway endpoint to be registered is the address of the NAT, not the address of the gateway. Additional code to do address discovery is needed.

Cons:

- Performance, scalability, hard to configure

Best for:
If you have no other choice, or use it from home or small office.

NATs without port forwarding

In extreme cases, the service may not be able to open an inbound port in the NAT. A number of protocols exist allowing a 'reverse Gateway', where the gateway server will open a connection as client to the main networks gateway. For example SOCKS allows this, as well as few new protocols. The behavior will be similar with the other solutions, but an additional service will need to maintain the reverse channel and the registrations will use the IP address of the tunnel server.

Cons:

- Poor performance, requires external service

Best for:
If you have no other choice (and can't even open a TCP port in the firewall). For example for development/debugging. Should not be used for production.

TLS termination in gateway

In this case, the gateway will handle mTLS authentication:

- It must have a certificate that is recognized by the client - either provide the expected service account cert based on the SNI, or clients must accept the gateway certificate as a valid secure name for all services behind the gateway
- It must request mTLS client certificate - this typically means that port can't be shared with normal 'browser-based' Gateway. (a browser accessing the secure port would get a prompt for certificate)
- It must propagate the mTLS client certificate to the service - in case of TCP proxy using one of the solutions for passing metadata (haproxy protocol, etc).
- The service must verify the 'delegated' call is from the gateway (to prevent other services from impersonating callers)
- The gateway must be 'trusted' - this class of solutions do not provide end-to-end security, the gateway will have access to clear-text. Additional layer of encryption can be added - but it requires additional complexity (and would effectively turn this into one of the 'TLS termination at destination' discussed above).

This is a more complex solution compared with the case where the TLS was terminated by the destination service. It also requires an additional decryption and encryption at the gateway, which may have latency/cpu impact.

For routing, in case of HTTP, the gateway can forward the request to the actual service using the Host header.

For TCP, we still need a similar 'one port per service' is required, or SNI (with specialized envoy/gateway code to use the SNI for routing, by default it's only used to select the server certificate).

The main problem in this setup is mTLS and identity delegation and trust: the client will receive the gateway cert, while the server will see gateway as client.

Istio Ingress

Current Istio ingress can be used with changes in the secure naming configuration. Extensions for passing the identity may be needed, but Envoy seems to already provide this for HTTP.

Cons:

- Requires implementation/design of secure delegation
- Requires Trust in the ingress, if Gateway is compromised the effects are broad.

Foreign LB TLS and HTTPS termination

In cases the cluster inbound has to use a specialized gateway that doesn't allow websocket or TCP, it is possible to terminate the TLS on the 'foreign' gateway. In general most LBs do not

require client certificates - and are unlikely to support Istio certificate format - the client sidecar will need to generate a JWT token and use regular TLS, and the server sidecar will need to process the JWT token forwarded by the gateway.

Cons:

- May not support mTLS
- JWT is less secure
- May be even harder to trust than Istio gateway operated by the developer.

Access control

The services exposed via the gateway can be accessed by anyone on the internet - but only if the client has a valid mTLS client certificate issued by Istio CA. We explicitly set as a requirement that all communication over zero-VPN networks will use mTLS or equivalent “encryption and mutually authentication”.

It is possible to use annotations to whitelist/blacklist services that can be exposed via the gateway, as an additional layer of control, so some services will not be ‘meshed’ outside of the zero VPN.

Beyond corp goes into many details for one similar deployment and the security implications.