Shared with Istio Community

Ambient Workload XDS Evolution

Author: John Howard Louis Ryan Justin Pettit
Date: Mar 14, 2023
Status: In review

Background

TODO

Gaps to fill

- Services should support the full service API
- Metrics should include service information
- Services should be able to have non-Pod endpoints (manually updated Endpoints)
- Service readiness should be properly respected
    - Not as simple as our current logic; publishNotReadyAddresses is one example but there are likely more.
- (Lower priority, optimization) Multiple waypoints causes large churn when scaling up or down
- East west gateway not accounted
- Key needs to be multi-network aware
- Workloads are only based on Pods, not WorkloadEntry
- DNS cannot be implemented, if we want it, since we miss Service hostnames

Intentionally omitted

The following are functionalities that are intentionally out of scope for ztunnel

- DestinationRule
    - Connection pooling
    - Locality LB
    - Subsets
- UDS endpoints
- NONE resolution mode*
    - It can be supported, but done entirely as a control plane operation. Because ztunnel is only matching on IP addresses, the original dst can only be the IP we matched. So we can represent these as normal services with service IP == destination IP.

API Principals

The configuration API consists of relatively high-level types that describe the state of the environment, leaving the logic of *how* that should be processed to the data plane. That is, unlike Envoy XDS types, most of the logic lives in the data plane rather than the configuration itself.

Types

The API consists of a few types:

- Service, which can represent:
    - A grouping of workloads. The Service represents a frontend (name and VIP), and a group of backends (Workloads) which are load balanced across. This is how a Kubernetes Service is represented. The API refers to these as "Workload Services".
- Workload, which represents a single workload instance. This could be a Pod, a VM, or other compute instance behind ztunnel-resolved DNS hostname. A Workload has an address we can reach it at, an (optional) identity, various metadata, etc. It is associated with Policies and Services
- Route, which represents an override of traffic matching some pattern to instead be sent to a different destination.

Key

These types are queryable by *IP Address*. This allows the data plane to lookup information about an IP address on-demand. For example:

- When I get a request to IP X, I can look it up and determine if it's a Service, Workload, or matches a Route, and learn how I should forward the request.
- When I get a request from IP Y, I can look it up and find the metadata associated with it to report telemetry on.

This on-demand nature is not *required*; data planes can (and, at small scales, probably should) perform wildcard subscriptions and receive all information to reduce latency and control plane reliance. However, the API is designed to be able to run in an on-demand mode for use cases where it is desired.

Client Awareness

The API is defined in a manner such that each object is universal. That is, for a given key, all clients can receive the same configuration. This is opposed to being client-aware (like Istio sidecar configuration), where we may give client-specific configuration.

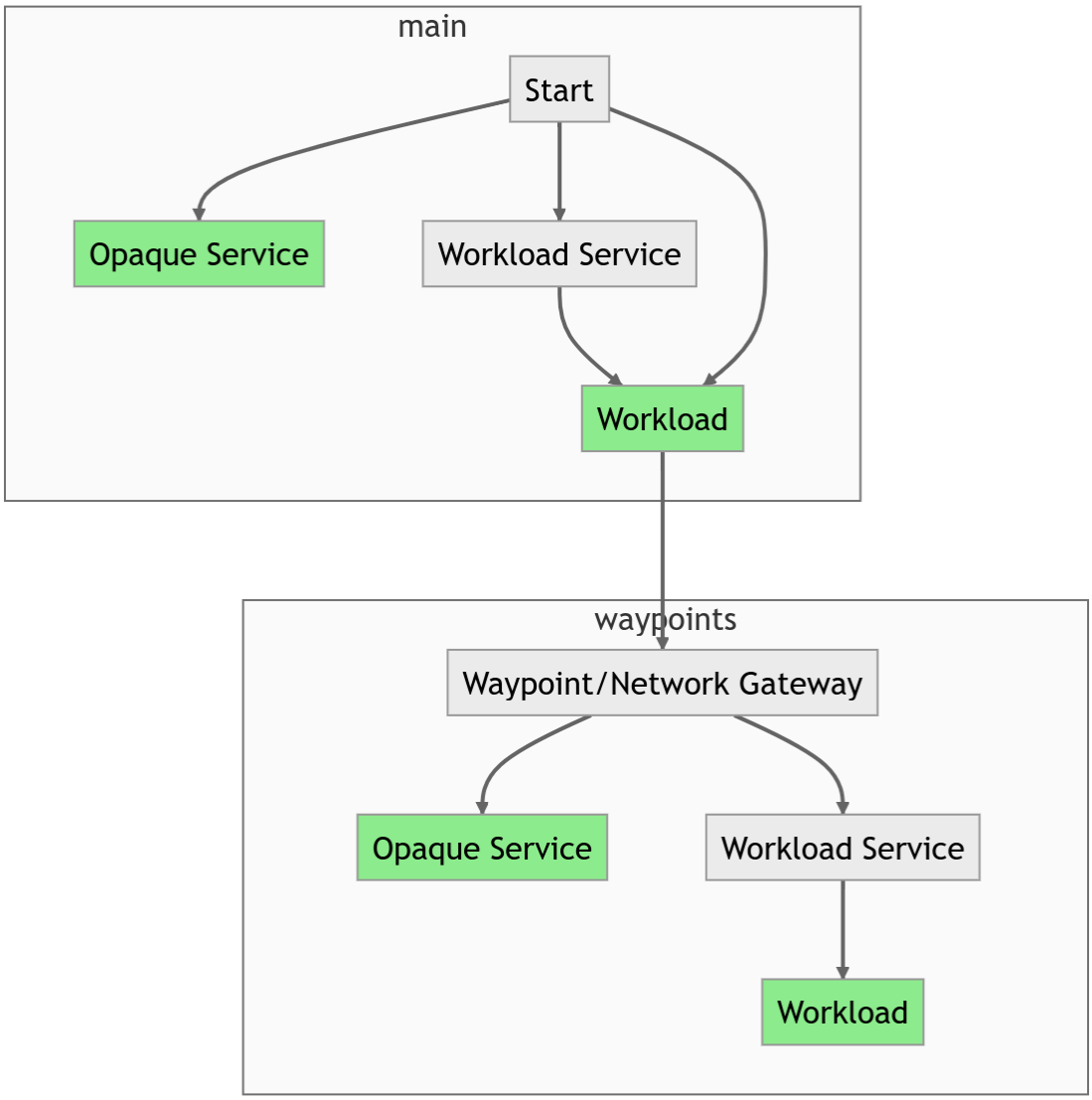
Being client agnostic is both substantially simpler (in implementation and debugging), and allows extremely efficient control plane implementations as caching becomes trivial. Rather than requiring the control plane to compute resources on the fly, it essentially becomes a CDN for a set of precomputed resources.

In practice, this largely means that references are fully qualified in the API. IP Addresses (generally) have a network associated with them, node names have a cluster associated with them, etc.

Note that this would *allow* a control plane to operate in a split-brain mode if it chose to. For example, we may have a list of (network,VIP). By requiring fully-qualified references, we leave open the possibility for a control plane to elide some references based on the client, while still allowing other implementations to include all references.

Graph

These types together form a logical graph. At runtime, the data plane makes traffic routing decisions based on traversing this graph.



A representation of the graph traversal of the API. Green nodes are "terminal".

In text form:

- When the data plane handles a connection, it resolves the IP address of the target destination to one of the declared types.
    -
    - If it's a Service we continue to traverse and select (via some load balancing algorithm) an appropriate Workload within the Service.
    - If it's a Workload (or we traversed here from a Workload Service), then we need to send it to this specific Workload. In some cases, this is terminal. However, if there is a waypoint proxy then we must traverse it, so we will continue the graph traversal. If there is a network gateway defined, we will traverse it only if it's in another network.

- If we need to traverse a waypoint or network gateway, we drop into the subgraph below. Note: these are two distinct paths, but since the graphs are the same they are consolidated for simplicity.
    - A Gateway always refers to a service
    - Like above, we traverse again to select a specific workload within that Service.
    - Notably, this full graph forms a possible loop. The data plane is expected to reject configuration with a loop. That is, a workload that is a part of a waypoint service should not have a waypoint itself, and similar for a network gateway. This ensures there is no looping possible.

An example of a full end-to-end traversal:

- Client calls `curl service-a`
- Data plane looks up the IP, and finds Service `service-a`.
- Data plane picks a specific workload that is a part of `service-a`, `workload-a`.
- `workload-a` contains a waypoint `1.2.3.4`, so we need to extract information about that waypoint as well.
- Data plane looks up the `1.2.3.4` IP to find the Service associated with it. It finds Service `waypoint-svc-1`.
- Data plane picks a specific workload that is a part of `waypoint-svc-1`, `waypoint-workload-1`.
- Data plane constructs a request with:
    - Destination IP: `waypoint-workload-1`'s IP
    - Protocol: HBONE (as found in `waypoint-workload-1`'s protocol specification)
    - HBONE `:authority`: `service-a`'s IP

Proposal

Below is the proposed new/modified API. Changes are highlighted.

The primary differences are:

- A new type, Service, is introduced to give dedicated information about a Service, so we don't need to inline it all into a Workload. This will expand over time to support all of the Service spec, but for now is kept minimal.
- A new type, Address, is introduced to join Workload and Service under a unified resource. This is mostly useful for on-demand lookups; without that requirement, we could just open up 2 streams.
- A new type, Route, is introduced to start Egress routing. This is currently under specified; I included it for completeness, but we need to add Egress to Ambient APIs (public) before we commit to this. I recommend we hold off on the Route API for the short term.
- Waypoints are represented by a new GatewayAddress type, which will now refer to a Service instead of a list of workloads

- An explicit `network_gateway` field is added.

```
// Address represents a unique address.
//
// Address joins two sub-resources, Workload and Service, to support querying by IP
address.
// Address is intended to be able to be looked up on-demand, allowing a client
// to answer a question like "what is this IP address", similar to a reverse DNS
lookup.
//
// Each resource will have a mesh-wide unique opaque name, defined in the
individual messages.
// In addition, to support lookup by IP address, they will have *alias* names for
each IP the resource represents.
// There may be multiple aliases for the same resource (examples: service in
multiple networks, or a dual-stack workload).
// Aliases are keyed by network/IP address. Example: "default/1.2.3.4".
//
// In some cases, we do not know the IP address of a Workload. For instance, we may
simply know
// that there is a workload behind a gateway, and rely on the gateway to handle the
rest.
// In this case, the key format will be "network/resource-uid". The resource can be
a Pod, WorkloadEntry, etc.
// These resources cannot be looked up on-demand.
//
// In some cases, we do not know the IP address of a Service. These services cannot
be used for matching
// outbound traffic, as we only have L4 attributes to route based on. However,
// they can be used for Gateways.
// In this case, the key format will be "network/hostname".
// These resources cannot be looked up on-demand.
message Address {
  oneof type {
    Workload workload = 1;
    Service service = 2;
  }
}

// Service represents a service - a group of workloads that can be accessed
together.
// The primary key is "namespace/hostname".
// Secondary (alias) keys are the unique `network/IP` pairs that the service can be
reached at.
message Service {
  // Name represents the name for the service.
```

```
  // For Kubernetes, this is the Service name.
  string name = 1;
  // Namespace represents the namespace for the service.
  string namespace = 2;
  // Hostname represents the FQDN of the service.
  // For Kubernetes, this would be <name>.<namespace>.svc.<cluster domain>
  string hostname = 3;
  // Address represents the addresses the service can be reached at.
  // There may be multiple addresses for a single service if it resides in
  // multiple networks, multiple clusters, and/or if it's dual stack.
  repeated NetworkAddress address = 4;
  // Ports for the service.
  // The target_port may be overridden on a per-workload basis.
  repeated Port ports = 5;
  // Optional; if set, the SAN to verify for TLS connections.
  repeated string subject_alt_names = 6;
}

// Workload represents an individual workload.
// This could be a single Pod, a VM instance, etc.
// The primary key is UID.
// Secondary (alias) keys are the unique `network/IP` pairs that represent the
workload.
message Workload {
  // UID represents a globally unique opaque identifier for this workload.
  string uid = 0;
  // Address represents the address for the workload. Combined with `network`
below,
  // this should be globally unique.
  // If multiple addresses are set, they are assumed to be fungible; this is
typically
  // used to provide both an IPv4 and IPv6 address for the same workload.
  // Each workload must have at least either an address or hostname; not both.
  repeated bytes addresses = 1;

  // The hostname for the workload to be resolved by the ztunnel.
  // DNS queries are sent on-demand by default.
  // If the resolved DNS query has several endpoints, the request will be forwarded
  // to the first response.
  // Each workload must have at least either an address or hostname; not both.
  string hostname = 21;

  // Network represents the network this workload is on. This may be elided for the
default network.
  // A (network,address) pair makeup a unique key for a workload *at a point in
time*.
  string network = 2;
```

```
  // Protocol that should be used to connect to this workload.
  TunnelProtocol tunnel_protocol = 3;

  // If set, this indicates a workload expects to directly receive tunnel traffic.
  // In ztunnel, this means:
  // * Requests *from* this workload do not need to be tunneled if they already are
tunneled by the tunnel_protocol.
  // * Requests *to* this workload, via the tunnel_protocol, do not need to be
de-tunneled.
  bool native_tunnel = 4;

  // Namespace represents the namespace for the workload.
  string namespace = 5;
  // The SPIFFE identity of the workload. The identity is joined to form
spiffe://<trust_domain>/ns/<namespace>/sa/<service_account>.
  // TrustDomain of the workload. May be elided if this is the mesh wide default
(typically cluster.local)
  string trust_domain = 6;
  // ServiceAccount of the workload. May be elided if this is "default"
  string service_account = 7;

  // If present, the waypoint proxy for this workload.
  // All incoming requests must go through the waypoint.
  GatewayAddress waypoint = 8;

  // If present, East West network gateway this workload can be reached through.
  // Requests from remote networks should traverse this gateway.
  GatewayAddress network_gateway = 9;

  // Virtual IPs defines a set of virtual IP addresses the workload can be reached
at.
  // Typically these represent Service ClusterIPs.
  // The key is an IP address.
  map<string, PortList> virtual_ips = 10;

  // A list of authorization policies applicable to this workload.
  // NOTE: this *only* includes Selector based policies. Namespace and global
policies
  // are returned out of band.
  // Authorization policies are only valid for workloads with `addresses` rather
  // than `hostname`.
  repeated string authorization_policies = 11;

  // Weight of the workload when load balancing across a service. Higher weights
receive more traffic.
  uint32 weight = 12;
```

```
  // Current health status of the workload.
  WorkloadStatus status = 13;
}

message WorkloadMetadata {
  // Name represents the name for the workload.
  // For Kubernetes, this is the pod name.
  // This is just for debugging.
  string name = 1;
  // CanonicalName for the workload. Used for telemetry.
  string canonical_name = 10;
  // CanonicalRevision for the workload. Used for telemetry.
  string canonical_revision = 11;
  // WorkloadType represents the type of the workload. Used for telemetry.
  WorkloadType workload_type = 12;
  // WorkloadName represents the name for the workload (of type WorkloadType). Used
for telemetry.
  string workload_name = 13;
}

message Locality {
  // Name of the node the workload runs on
  string node = 1;

  // The cluster ID that the workload instance belongs to
  string cluster_id = 2;
}

enum WorkloadStatus {
  // Workload is in an unknown state.
  UNKNOWN = 0;
  // Workload is healthy and ready to serve traffic.
  HEALTHY = 1;
  // Workload is unhealthy and NOT ready to serve traffic.
  UNHEALTHY = 2;
  // Workload is draining (or terminating) and will soon be unable to serve
traffic.
  DRAINING = 3;
}

enum WorkloadType {
  DEPLOYMENT = 0;
  CRONJOB = 1;
  POD = 2;
  JOB = 3;
}
```

```
// PortList represents the ports for a service
message PortList {
  repeated Port ports = 1;
}

message Port {
  // Port the service is reached at (frontend).
  uint32 service_port = 1;
  // Port the service forwards to (backend).
  uint32 target_port = 2;
}

// TunnelProtocol indicates the tunneling protocol for requests.
enum TunnelProtocol {
  // NONE means requests should be forwarded as-is, without tunneling.
  NONE = 0;
  // HTTP means requests should be tunneled over HTTP.
  // This does not dictate HTTP/1.1 vs HTTP/2; ALPN should be used for that
purpose.
  HTTP = 1;
  // Future options may include things like QUIC/HTTP3, etc.
}

// GatewayAddress represents the address of a gateway
message GatewayAddress {
  // address can either a hostname (ex: gateway.example.com) or an IP (ex:
1.2.3.4).
  // Note that both of these are just lookup keys for Service
  oneof address {
    NamespacedHostname hostname = 1;
    NetworkAddress address = 2;
  }
  // port to reach the gateway at
  uint32 port = 3;
}

// NetworkAddress represents an address bound to a specific network.
message NetworkAddress {
  // Address presents the IP (v4 or v6).
  bytes address = 1;
  // Network represents the network this address is on.
  string network = 2;
}

// ROUTE WILL BE DEFERRED UNTIL USER FACING EGRESS IS DEFINED
// Route defines an L4 route rule, allowing rewriting requests to alternative
```

Control Plane Implementation

While Workload is mostly unchanged in protobuf form, the control plane implementation will be enhanced. Currently, we read only Pods, and join with Services to produce VIPs. This leads to missing use cases (ServiceEntry, WorkloadEntry), and buggy logic (Pod+Service=>Endpoints logic is not correct).

This behavior will be optimized to read all Pod and WorkloadEntry to build up the base Workload struct. This will then be joined (by IP+Network key) with endpoints (which could be Endpoints, EndpointSlice, and/or synthesized endpoints from Service-selects-WE and SE-selects-Pod) to determine Service association.

For endpoints that do *not* have a Pod or WorkloadEntry (for example, Kubernetes services without selectors), we will also create a Workload. This is needed to support manually set Endpoints for a Service. These Workloads will be fairly minimal; they will always be TCP, have no telemetry metadata, etc. Importantly, however, they will have an IP address and be associated with a service.

The waypoint address will be changed from a list of workload IP addresses to the waypoint service. This is a control plane/XDS change, not a behavioral change; the ztunnel will still resolve this service to a set of workloads to load balance over. That is, it will not send directly to the VIP. This new API also allows the waypoint address to be an opaque Service without an IP

or workloads associated. This would support future use cases like `waypoint: waypoint.example.com`. Since this is not an urgent use case, the ztunnel implementation should likely be deferred.

Did we meet the requirements?

From above, the requirements:

- Services should support the full service API
  - **ENABLED:** this doesn't *solve* this, but gives us the extensibility to do so easily in the future.
- Metrics should include service information
  - **SOLVED**: we now have the full Service name, namespace, and hostname to report in metrics
- Services should be able to have non-Pod endpoints (manually updated Endpoints)
  - **SOLVED:** we now will include these as (minimal) Workload resources
- Service readiness should be properly respected
  - **SOLVED:** Endpoints will be consulted for readiness purposes
- (Lower priority, optimization) Multiple waypoints causes large churn when scaling up or down
  - **SOLVED**: waypoints will be represented as stable service addresses
- East west gateway not accounted
  - **SOLVED**: new field added
- Key needs to be multi-network aware
  - **SOLVED**: network is part of the key
- Workloads are only based on Pods, not WorkloadEntry
  - **SOLVED**: WorkloadEntry will be included in the source information for Workload
- DNS cannot be implemented, if we want it, since we miss Service hostnames
  - **SOLVED**: we now have the full service hostnames to return in DNS responses

Why not…

"Next Hop"

Instead of a `Waypoint Address` and `Network Gateway` config, we could have a more generic `Next Hop` field. However, the two are not interchangeable. For waypoints, ztunnel is also required to *enforce* traffic going through the waypoint. For network gateway, there are some additional protocol requirements (Multi-network Ambient).

Additionally, the `Next Hop` is client aware; we have a different answer depending on the network ztunnel is in. This makes the control plane implementation less efficient, as we cannot precompute the XDS resource.

As a result, it makes sense to split these out.

Overlapping IPs within a network, Option 1

Some users are configuring split-brain control planes, where each cluster only knows the number of endpoints behind a service in another network. For example:

```
kind: WorkloadEntry
metadata:
  name: service-a
spec:
  address: <eastwest-gateway-address>
  network: remote
  weight: 12
---
kind: WorkloadEntry
metadata:
  name: service-b
spec:
  address: <eastwest-gateway-address>
  network: remote
  weight: 4
```

This model doesn't work well with IP-keyed resources. However, Istio has defined a network as having a non-overlapping IP space, and already enforces this in a variety of places (even in sidecars).

The recommendation is to instead represent this as a workload with an *unknown* IP address. This is introduced as a workload keyed by Network+UID instead of Network+IP in the Workload API. These IP-less workloads are unique in that they can only be accessed as backends to a Service; they cannot be the source of traffic (as we cannot identify them) nor the destination of direct-to-pod traffic (again, we cannot identify them).

Note: this will require making some fields in the workload optional, e.g.

- address(es)
- Node
- Likely others (including in the future)

The new weight field added to the workload proto can be used to signal replica count in the remote cluster if desired. (i.e., the workload here with unknown IP can really represent a group of remote workloads)