Istio Auto Multi-Tenancy 101

mtail@google.com
Last Update: November 22, 2016

Introduction

This document provides a brief introduction on how to use the Istio Auto Multi-Tenancy (AMT) feature, which makes it easy to turn a single-tenant application into a multi-tenant service. We walk through the main features and basic architecture, then describe the steps necessary to use AMT.

This document assumes familiarity with the Istio Architecture and the Plori Service Broker model.

Benefits

Istio's AMT feature ingests one or more containers for a single-tenant application, a deployment script, along with some configuration state, and wraps the containers in sufficient infrastructure to expose a fully-functional, monetizable, multi-tenant service.

This approach eschews much of the inherent complexity of multi-tenant software design. It enables developers to continue creating single-tenant software, yet take advantage of multi-tenancy benefits including:

- Ease of consumer provisioning
- Flexible monetization story
- Auto-scaling
- Consolidated ops and analytics

All of these features are delivered while maintaining robust tenant-level state and performance isolation. The Automated Multi-Tenancy PRD provides additional details on these and other benefits of this approach.

Features

The essential characteristics of the AMT feature include:

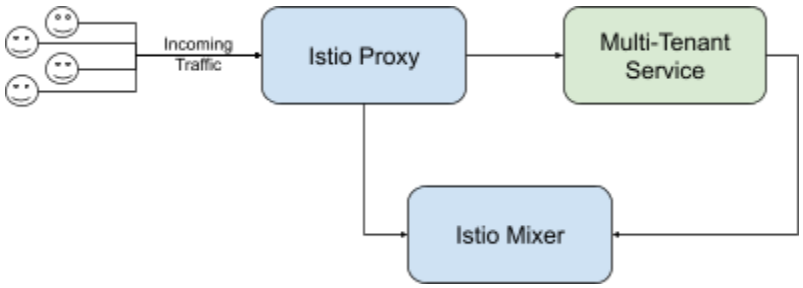
- Quickly turn single-tenant applications into multi-tenant services without the need for code changes.
- Automatically handle provisioning of tenants.
- Automatically add API-level and network-level telemetry to enable the service provider to effectively manage the service.
- Provide flexible billing options to quickly monetize the service.
- Deliver strong isolation guarantees to prevent individual tenants from interfering with one another, thus preventing noisy-neighbor problems and helping to prevent data leaks.
- Provide tenant-specific rate limiting quotas.

The Automated Multi-Tenancy PRD provides a detailed set of feature requirements for an eventual product offering in this space.

Basic Architecture

We assume the reader is familiar with the basic Istio runtime architecture and composition as outlined in go/Istio-runtime-101. The design we describe here is Kubernetes-centric.

Here's how a classic multi-tenant service fits into the Istio runtime:



The goal of Istio AMT is to take a single-tenant application and turn it into a multi-tenant service. The AMT infrastructure allows the single-tenant application to receive traffic from multiple tenants, to report telemetry and billing to multiple tenants, and to allow the service operator to treat multiple deployments of the application as a single service.

AMT creates a new deployment of a service whenever a new tenant is introduced. AMT automatically manages these deployments, and automatically routes incoming traffic to the appropriate deployment.

AMT supports two distinct operational modes for the services it manages:

- **Private IP**. In this mode, every tenant of the multi-tenant service uses distinct IP addresses to communicate with the service. This mode is only usable for environments where IP addresses abound. More specifically, this mode is appropriate for in-cluster usage, tightly-coupled cloud deployment scenarios (using something like XPN/XON), but not for the open Internet.

- **Shared IP**. In this mode, all tenants of the multi-tenant service use the same set of IP addresses to communicate with the service. Since this mode uses a fixed set of IP addresses, it can readily be used for off-cluster traffic.

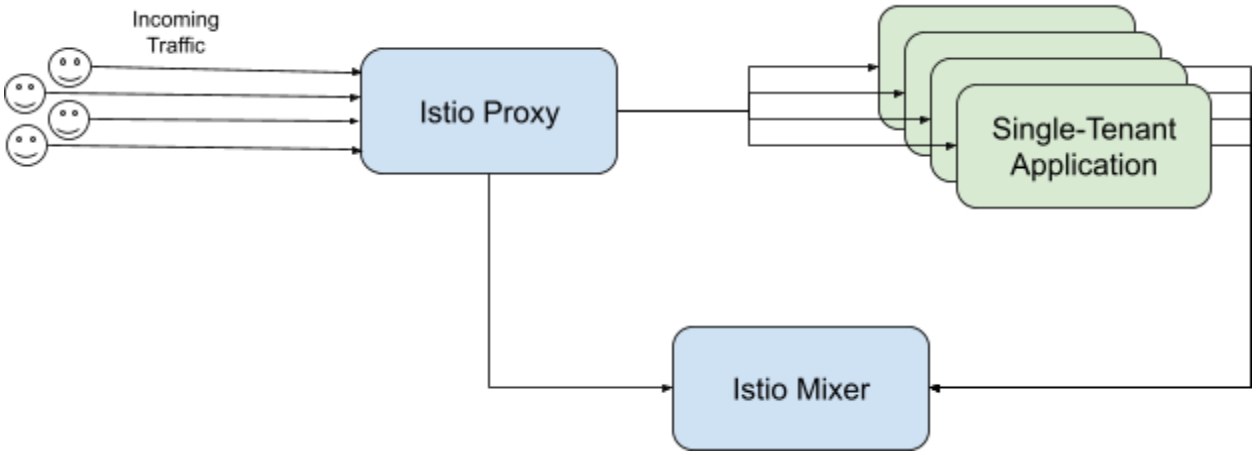
AMT classifies incoming traffic to a service into three distinct routability categories:

- **L4 Routable**. Uses transport-level authentication such as TLS mutual auth, SPIFFE, or LOAS, to route any incoming traffic to the right destination. Whether L4-level routing is enabled depends on whether Istio as a whole supports suitable transports.

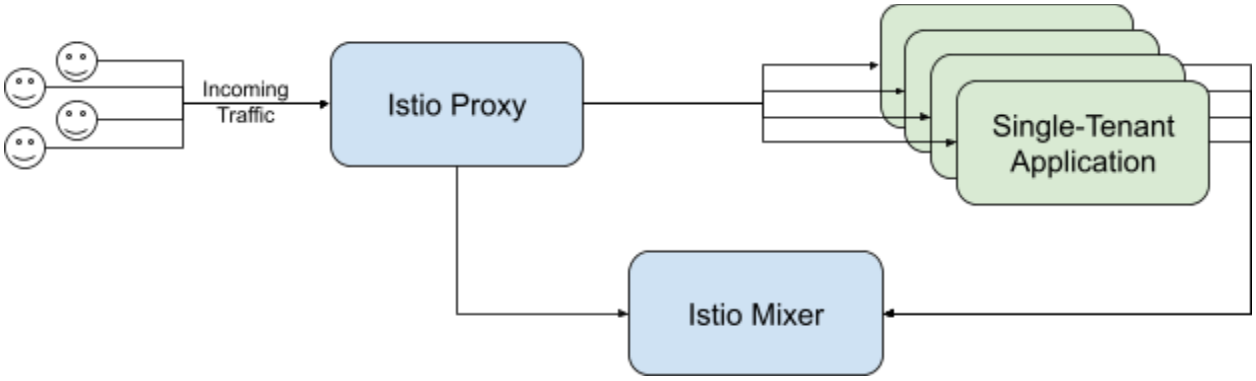
- **L7 Routable**. Uses application-level authentication such as JWT or API Keys, to route any incoming traffic. Although HTTP and gRPC are the only protocols likely to be supported natively by Istio, it should be possible to easily extend the infrastructure to support other application protocols as well over time.

- **Unroutable**. If neither L4 or L7 authentication is available, then incoming traffic is deemed unroutable by Istio.

The specific combination of operational mode and routability determines the ultimate topology of an AMT-based Istio service. Unroutable traffic can only be used with the private IP operational mode and in such a situation Istio deploys one proxy with multiple IP addresses (one address per application deployment):



When the proxy knows how to route incoming traffic, then it's possible to share a single IP address between application deployments such as:



The above architecture diagrams show the semantic composition, not the physical composition in terms of Kubernetes clusters and pods. The exact topology of a deployed application varies from application to application, driven by the deployment script created by the application developer. There could be one or more standalone clusters per application deployment or perhaps only a pod per deployment located in a communal cluster. Specific tradeoffs among cost efficiency, required level of security, and so forth will determine the best topology for any specific application.

The Mixer

In the above diagrams, the mixer receives traffic from various independent application deployments. Being designed for single-tenancy, the application deployment doesn't supply the mixer the tenant information that it needs. the mixer copes with this problem by automatically discovering the application deployment sending it traffic and establishing tenant identity from there.

Service Brokers

Service brokers are responsible for creating and managing application deployments. As a service consumer creates a service instance via the service broker API, the service broker implementation creates an application deployment and configures Istio to accept and properly route traffic for each deployment.

Through configuration, the developer defines the specific service broker to use for their service, based on the deployment tool used to deploy the application (a service broker for Chef, one for Puppet, one for Deployment Manager, etc)

Monetization

AMT-based services can be monetized using four distinct approaches:

- **Network-Based Billing**. Based on metrics generated by the Istio proxy, it's possible for a service to charge for network traffic ingress and egress. This can be enabled purely through configuration changes without having to change the service code.

- **API-Based Billing**. For HTTP and gRPC protocols, the metrics generated by the Istio proxy can be used to charge per API call into the service, including different amounts for different methods. Just as for network billing, this can be enabled through configuration changes.

- **Subscription-Based Billing**. This requires cooperation from a marketplace provider. Through the marketplace, subscriptions can be established independently from the service proper.

- **Custom Billing**. The service can make calls to the mixer to charge the tenant. This approach is not automatic; it requires explicit changes in the service's code.

Incremental Istio Changes

Implementing AMT requires a modest investment relative to the core Istio runtime model. The primary work is shown here:

- **Routing**. The proxy needs to understand how to handle L4 or L7 routing to direct incoming traffic to the right application deployment.

- **Mixer Identity Recovery**. When the mixer is called without suitable identity information, it needs to recover tenant identity.

- **Proxy Deployment Topology**. The Istio Manager needs to understand how to deploy the proxy to understand the difference between routable and unroutable configurations and deploy the proxy accordingly.

Using Auto Multi-Tenancy

Using AMT is a small incremental change relative to the normal Istio runtime flow around multi-tenant services as described in go/Istio-101.

The steps involved in using AMT are:

- **Containerize the Application**. The application developer needs to package the application as one or more containers. This enables the application to run in the Kubernetes environment.

- **Create Deployment Scripts**. Once application containers exist, the service provider must create deployment scripts that can take these containers and get the application running in a Kubernetes environment. These deployment scripts can be written using any of a set of deployment tools like Helm, TerraForm, Deployment Manager, and others. The scripts need only be concerned about the application proper, deploying the Istio proxy, the multiplexer, and the mixer are the responsibility of the Istio runtime.

- **Create a Cluster**. The service provider needs to create a Kubernetes cluster in which the various instances of the application will be deployed to form the multi-tenant service.

- **Enable Istio**. The service provider follows the procedures described in go/Istio-101 to enable Istio for the cluster.

- **Publish a ManagedService Resource**. Again, as described in go/Istio-101, the service provider must publish a ManagedService resource to Kubernetes in order to activate the

runtime for the service. It only takes a few lines of additional configuration within the resource in order to enable AMT:

```
// Enable a service broker for this service.
// The service broker will be available to receive incoming
// requests and will list the current service in its catalog.
bool enable_service_broker;

// Provide the location of the provider-authored deployment
// script. This will be used by the service broker to
// deploy a new instance of the application as necessary.
string application_deployment_script;

// Provide the type of the deployment tool needed to
// execute the deployment script. This can be one of
// the well-known tools supported by the
// implementation of the service broker.
DeploymentTool application_deployment_tool;

// Configures whether the service should be configured for a
// single shared IP address or for per-instance IP addresses
bool enable_shared_ip;
```

- **Manage the Service**. Finally, once the ManagedService resource has been published, the service provider can use the normal CLI and GUI tools to manage the service, like any other service as described in go/Istio-101.

Tenant Isolation

The architecture we describe here is orthogonal to the ultimate topology of the individual application deployments. The deployment scripts written by the service provider can decide to house some or all application deployments in different clusters for example. AMT doesn't care how these deployments are realized, so long as there is IP connectivity between the proxy and the individual deployments.

The common composition is expected to use a single communal cluster and one Kubernetes Deployment for each application deployment, in a new namespace. This provides good basic isolation and binpacking benefits.

Some variations are possible. For example, using labels, it's possible to control which nodes a given deployment uses relative to other deployments. This could enable a model where small customers share a common set of nodes, while special high-value customers can be isolated to separate sets of nodes.

Updates

The update model provides for the gradual rollout of both container and configuration changes within an AMT-managed cluster. The model is built around Kubernetes's existing Deployment-based rollout approach. The general flow is this:

- Developer pushes new containers and configuration to the Istio Manager, which initiates a rollout.
- The Istio Manager configures the Service Broker such that any new instantiation of the application will use the new containers and configuration.
- The Istio Manager communicates with the Service Broker and has it initiate a global upgrade script provided by the developer. This allows the developer to do one-time global changes to the cluster, such as updating database schemas.
- The Istio Manager iterates through every instance of the application and instructs the Service Broker to update each in sequence.
- For each application, the Service Broker triggers a developer-provided upgrade script which is responsible for doing the work necessary to update a single instance of the application. The typical upgrade script merely triggers Kubernetes to update the current Deployment.
- When upgrading a Deployment, Kubernetes iterates through the pods that are part of the deployment and restarts them with the new containers and config.
- After each instance of the application is upgraded, the Service Broker invokes a developer-supplied post-upgrade script that does whatever cleanup is needed.
- After all instances of the application have been upgraded, the Service Broker invokes a developer-supplied global post-upgrade script.
- Throughout this process, the normal Kubernetes health monitoring mechanism is used to determine whether individual pods being brought up are healthy or not. Overall deployment of an upgrade can abort and be rolled back if failures are discovered.

As described here, all customers of a service would eventually be upgraded to new versions of containers and config. It would be fairly simple to introduce configuration controls such that specific customers can stay on specific versions, while others are upgraded.

Billing

In order to enable an AMT-based service for billing, the service must be registered with one or more payment systems. The registration process is no different for AMT-based services than normal multi-tenant services. Registration enables a service to charge customers and determines the price sheet to use. The registration process is outside the scope of this document and is TBD.

Once an AMT-based service is registered, simple configuration changes make it possible to do network-based or API-based billing. The configuration indicates the set of metrics to publish to the payment system(s). Once this configuration is deployed, the Istio runtime will automatically forward the metric data to the payment systems. The payment systems will apply the appropriate charging model to determine a monetary amount to charge the user.

Service code can be modified to do custom billing as well. The service code then makes calls to the mixer to report particular billing events, which the mixer forwards to the appropriate payment system(s).