Simplified Istio (istiod)

("~~Premature optimization~~ Complexity is the root of all evil or: How I Learned to Stop Worrying and Love the Monolith")

Owner: costin, lryan **Working Group:** EnvWG	**Status**: WIP | **In Review** | Approved | Obsolete **Created**: 10/01/2019 **TOC Approvers:** [D]sven, [D]dan, []martin, [D]lin, [D]shriram **Approvers**: [D]sdake@, [D]yongangl@

TL;DR

Simplify the deployment and maintainability for the dominant use-cases of Istio by creating a standalone **istiod** binary that combines the core features represented by the current microservice components - Pilot, Galley, WebHook Injector, Citadel, Node Agent, Pilot Agent, … The internal features are orchestrated internally using code rather than externally via configuration.

In addition, simplify the bootstrapping of authentication from the dataplane to the control plane by using workload JWTs & DNS-cert-generation using Apiserver-signed certs. A separate **istio-agent** will leverage services from **istiod** that can be run per pod, per-VM or as a local server without full RBAC permissions.

Finally this document outlines a set of other simplifications for enablement and configuration flows during install and runtime that reduce installation complexity and behavioral coupling. The simplifications are presented here for illustrative purposes but it is expected that there will be follow-on design documents to finalize details.

Action plan for 1.5

Approval Icebergs

This section serves to highlight the big outstanding issues in this doc

- ~~Istiod as monolith bundling CA behaviors with control plane. Are we OK with monolithic security posture for the default install~~
- Are microservices dead and istiod is used for all use-cases more or less
- ~~Config clarity - Doc needs work to establish and clarify roles and responsibilities of config artifacts while achieving the desired simplification goals.~~

Background

Istio has aggressively followed the microservice pattern of separating out different functional roles within the production stack into discrete deployable units with defined API boundaries. Adoption of this development pattern has not yielded an obvious advantage over a more monolithic approach and is by definition more complex. In particular

- In the vast majority of Istio installations all the components are installed and operated by a single team or individual. The componentization done within Istio is aligned on our internal development team boundaries. This would make sense if Istio was delivered as a managed service for users, this is obviously not the case. The microservice pattern is most effective when it helps distinct teams operate independently, that is not the case here and flexibility has become complex.
- Microservices also help when there are two features with orthogonal scaling or cost dimensions that need to be separated from each other to enable efficiency. There are several examples where this separation makes sense - separating Mixer from other control plane features or separating pilot-agent from PIlot. This is not the case in Istio today for the majority of components however - control plane costs are dominated by a single feature (serving XDS). Every other control plane feature has a marginal cost by comparison, therefore there is little value to separation.
- Features are also separated into different services because they have different security roles within a deployment. This segregation is meaningful if the services being separated do not have equivalent powers. This is not the case today as the powers exercised by the Mutating Webhook, Envoy Bootstrap & Pilot are similar to those of Citadel in many respects and therefore exploits of them have near equivalent damage. In the default installation there is also little separation of user administration privilege, all the components are installed into the same ‘istio-system’ namespace. For deployments that care such segregation will still be supported as deployment model, either by running a separate CA-only istiod instance or by using a 3rd party dedicated CA that can perform certificate signing. Ideally using the Kubernetes CA to sign SPIFFE certs would be one of those options.

In addition to the higher level concerns listed above there is a large amount of deployment complexity involved in deploying a graph of microservices, ensuring it is properly connected and communicating and making changes to it as features evolve. In fact it can be argued that these

costs significantly contribute to the perception that Istio is complex and expensive to install and manage.

Design Goals

This design lays out a set of proposals that radically simplifies the initial and ongoing experience for the human operators of Istio. In particular it will:

- **Reduce the install complexity** - Component dependencies will be encapsulated and hidden from users behind fewer binaries. The universe of configuration options & flags will be reduced significantly as a consequence.
- **Reduce configuration complexity** - A large percentage of the existing configuration surface of Istio exists to facilitate orchestration of the control plane components. Istiod eliminates the need for that by definition. As such it presents an opportunity to make other simplifications to the configuration and operability of the system including canonicalizing best-practice defaults and supported topologies. That effect is a specific goal of this proposal but details are TBD. To elucidate usage examples in this doc will only use mesh.yaml for config, with 'best practice' settings for everything else. A prologue to a more detailed configuration design is added to the appendix.
- **Control plane maintenance** - Operators and installers are less exposed to the vagaries of internal component dependencies and how to version them in live production systems. The ability to run multiple control planes to facilitate canarying also becomes simpler. Workloads select the control plane by using the injection label on the namespace and/or pod annotations.
- **Issue diagnosis** - Fewer components and less cross-component debugging to be performed makes problems easier to find.
- **Increase efficiency and responsiveness** - Contracts between components will not incur transmission overheads and caches can be safely shared as needed. Will improve footprint, startup time etc. significantly.
- **Eliminate needless coupling** - Move Envoy bootstrap generation to the control plane to avoid the need for agents to access privileged or central configuration.

This is similar to the 'hyperistio' experiment in early istio and is aligned with how K8S apiserver combines core services in a single binary.

This proposal does not strictly require the elimination of the existing standalone micro-services, it does however suggest that they must justify their existence relative to **istiod** which would become the de facto default. If a strong argument cannot be made for their retention **(see Appendices)** then they should be eliminated as the cost of supporting multiple solution options for end users and developers is ***very*** high:

- Complexity of options currently supported in values.yaml and helm templates to support control plane deployment and coordination

- Complexity of existing upgrade process and difficulty in shipping a canarying feature.
- Difficulty in resolving configuration topology choices as part of installation & operator design

Componentization will still be maintained internally within the stack for engineering hygiene reasons - testing, API contracts, vendoring etc - but that would not be visible to end-users.

It is not expected that the transition to **istiod** can be done quickly (in terms of adoption, testing, documentation) as much needs to change but the outlined value proposition above is high enough that ***the project should be a priority*** on the Istio roadmap in line with the priority given recently to other high-value usability efforts. Migration to istiod should require no downtime and be gradual, with user able to run istiod alongside normal istio and shift workloads to istiod.

We will also try to minimize the supported configuration surface of istiod to encourage the use of stable features and best practices. To expedite delivery experimental or unused features will not be supported for the early phases of istiod. As istiod matures we may deprecate features or choose to support them in a different manner. A good example is the consul adapter in Pilot which is still in experimental mode, instead of building it into Istiod we could instead ship an MCP producer for it. Another example is the ability of Citadel to save secrets into other namespaces, this feature is deprecated by SDS which istiod would fully support therefore there is little reason to enable that capability.

Related: Simplified SDS installation

Ideally, a user should be able to run "./istiod" (without any parameters) and have a fully functional istio control plane, using Apiserver if a .kube/config is found or a local directory. Combined with the sniffing and other optimizations this should provide basic telemetry and networking out-of-box, with no extra configuration. Similarly, a user should be able to run ./istio-agent and have a local agent that may operate as a control-plane cache or even as a standalone control plane using local files. Istio-agent will delegate to ./istiod or other MCP/XDS servers for the integration with K8S or other config or discovery systems.

Feature Inventory

This section lists out the discrete functional services that form the Istio control plane

1. **Configuration Watch** - Reads one or many of the supported configuration sources (K8S Apiserver, Files or MCP) and watches it for changes. Feeds the stream of configuration to downstream features in the control plane pipeline like xDS serving. Intent is to normalize variations in configuration handling by using the MCP abstraction in-process.

    a. **API Validation** - Validates and writes status back to underlying resources. Also acts as K8S admission control.
    b. **Configuration Forwarding** - Can ingest one or more configuration streams and serve it to another Istio control-plane instance over MCP in both push and pull modes.
2. **Endpoint Watch** - watch endpoints in local and remote clusters. Currently duplicated in Galley and Pilot - but the code in Galley is not optimized and is not used in prod Istio. We'll use the Pilot code, and move the synthetic ServiceEntry generation to use the Pilot endpoints. In future the endpoints may be replicated in each cluster as EndpointSlices - maybe including non-k8s sources like Consul.
3. **CA Root Generation** - Generates the private key and certificate for the root of the SPIFFE CA. Currently in Citadel.
4. **Control Plane Identity** - Currently bootstrapped with a SPIFFE id from CA Root Generation for inter-control-plane communication. Is also bootstrapped with a certificate from K8S CA (Chiron) for injector. See simplified control-plane identity proposal
5. **Certificate Generation -** Generates a private key and certificate to enable secure and identifiable dataplane communication. Excluding the 'save certs and private keys in certs' - only SDS/grpc based cert generation is included. This feature is often decomposed into two parts to enable a stronger security posture
    a. **Private Key Generation** - Done by an agent co-located with the data plane and then used to create a (CSR) certificate signing request
    b. **Cert Signing** - Done by the control plane either acting directly as a certificate authority or by delegating to one in an authenticated way.
6. **Auto-Injection** - Service used by Apiserver as mutating admission controller to inject sidecar into pod specifications.
7. **CNI/CRI Injection -** An alternative approach to injection using CNI/CRI as the hook. Currently not in scope.
8. **Envoy Bootstrap** - Generate startup configuration for Envoy.
9. **Local SDS Server -** Serves credentials and generated certificate based on workload identity to a local Envoy instance. Reads secrets visible to the workloads namespace only. Can delegate to a **Central SDS Server** for centrally managed secrets.
10. **Central SDS Server** - Serves credentials either directly to Envoy or indirectly via a **Local SDS Server.** Often used to access secrets in a 3rd party key-store like Vault.
11. **xDS Serving** - Serves dataplane configuration to running Envoy instances.

Historically the various Istio components provided these features packaged as follows …

- Galley = { Configuration Watch, API Validation, Endpoint Watch (synthetic SE), Configuration Forwarding}
- Webhook = {Auto-Injection, Envoy Bootstrap (mangling)}
- Citadel = { CA Self-signed Root Generation, Cert Signing, Secret generation in all namespaces, Integration with other CAs }

- Node Agent = { Private Key + CSR Generation, Local SDS Server, Integration with other CAs}
- Pilot Agent = { Envoy Bootstrap (unmangling/generation), Env detection, Launch/reload envoy, Readiness/debug support}
- Pilot = { Endpoint Watch, K8S Config Watch, xDS Serving}

In this proposal we will package as follows …

- istiod = {K8S/File Config Watch(galley), API Validation, MCP client and server, Auto-injection, EnvoyBootstrap Generation, Cert signing, CA Root Generation, K8S/Consul Endpoint watching (pilot), xDS Serving }
- istio-agent = { Private Key Generation, Local SDS Server, Launch envoy, Readiness/debug }.
- Removed: Galley Endpoint Watch, Pilot K8s Config Watch, Citadel Secret generation, Node Agent as a DaemonSet. Also all code dealing with config and communication across components.

This proposal makes it possible to deploy a single-component Istio binary, similar to the ‘minimal’ profile we support but with more functionality. This will eliminate a large swath of configuration options and complexity needed by components to talk with each other.

User Experience

This document shows usage of the istiod and istio-agent binaries for important use-cases. **Note command line syntax is used to elucidate behavior but we may use an mesh.yaml as the actual mechanism for config.**

Running inside a Kube cluster

```
Istiod.yaml
...
containers:
  image: istio.io/istio/istiod:latest
  [ no extra CLI or customizations ]
```

The default deployment will by default have the ‘recommended’ config for K8S, similar with 1.4 default install. “Istiod” will detect that is running inside a pod in a cluster, and start as a functioning control plane. It will read standard configuration artifacts such as an optional mesh.yaml from its namespace - typically istio-system.

In this mode services are enabled based on mesh.yaml config. If a mesh.yaml is not created by user - default install will be used.

Running inside a Kube cluster, for a single namespace

```
Istio.yaml:
    - RoleBinding instead of ClusterRoleBinding in the config, no permissions outside of ns

Mesh.yaml:
  Namespaces:
    - mynamespace
```

Istiod will detect that is running inside a pod in a cluster and bootstrap itself as a functioning control plane for the namespace within which it is running.

It will read standard configuration artifacts such as mesh.yaml from this namespace. The mesh config will have explicit options to control which namespaces to watch and manage. In this mode the SDS (CA) can be delegated to a different istiod instance, or may run locally.

Minion cluster polled by a remote istiod via MCP for multi-cluster etc.

In ‘minion’ mode an istiod instance is acting as a delegate for a remote master control plane. It will perform functions that can be handled entirely locally { Envoy Bootstrap, [Workload Certificate Generation (optional)], Configuration Watch, Configuration Forwarding, Auto Injection } but it does not do { CA Root Generation, xDS Serving}. Handling of data plane configuration {xDS serving} is delegated to a remote control plane.

```
Mesh.yaml:
  configSources:
 - address: mcp://mcp.server.address
```

In this case istiod will function as above but will not serve XDS and expects to be the source for cluster configuration changes by a remote istiod via MCP. Envoy bootstrap generated in the cluster will be directed to the specified XDS endpoint.

The configuration is supported in mesh.yaml, as a list of sources to poll. Changes to mesh.yaml are watched, and may change dynamically.

Minion cluster config pushing to a remote istiod via MCP for multi-cluster etc.

```
Istiod.yaml
...
containers:
  image: istio.io/istio/istiod:latest
  [ no extra CLI or customizations ]

Mesh.yaml:
 configSources:
  - address: push://mcp.server.address
```

In this case istiod will function as above but will not serve XDS and will push observed configuration changes in the cluster to a remote istiod via MCP. Envoy bootstrap generated in the cluster will be directed to the master DNS for its XDS endpoint. Note that it is expected that MCP port will also serve XDS unless otherwise specified.

The sources are configurable in mesh.yaml - currently only mcp:// and fs:// are supported, we will extend it to include the MCP push destinations as well (subject to review/separate proposals)

Running outside a cluster

```
./istiod
```

Will use KUBE_CONFIG or $HOME/.kube, with a KUBE_CONTEXT env to specify a non-default *kubeconfig* context.

All services are started including the services needed by a locally running Envoy. The control plane will infer the effective namespace of the control plane from the context user, which must have sufficient permissions to read all necessary configuration resources.

A local mesh.yaml file can specify additional options.

Running with static configuration

```
./istiod

Mesh.yaml:
  configSources:
    - address: fs:///my/config/dir
```

The config is specified in mesh.yaml, using existing fs:// definition.

In this example, a control plane instance is started from configuration resources stored on the local filesystem. The filesystem is processed recursively and all valid configuration sources are loaded. Though not required it is recommended to align directory structure with namespace structure.

All configuration artifacts under the root will be loaded with namespacing achieved by partitioning configuration resources into sub-directories. In this model all services are started, making it possible to run a local istio-agent and Envoy with no external dependencies. This can be used for testing, CI/CD or non-K8S environments.

Running the agent inside a pod

```
ISTIO_ADDRESS=istiod.istio-system
./istio-agent
```

In this example the agent will

1. Connect to istiod.istio-system.svc:15012 and authenticate calls to it using TLS based on Kubernetes Apiserver issued certificate + workload JWT injected into the pod. If injector configured a different 'discovery server', use the platform root of trusts if the discovery server is on port 443 (an explicit option can be added as well - but for the common case we can have a default based on port number). Istiod will generate the bootstrap for envoy and send additional config options.
2. Generate a private key and issue a CSR to istiod, or the external CA if configured by injector or by istiod. CSR is sent using the workload JWT and validated using either K8S cert or a public cert, if the CA is on 443.
3. Bootstrap Envoy based on a configuration provided by istiod
4. Envoy will connect to istiod and listen to xDS. Connection will use injected DNS name, and authenticate with a local JWT token using local root CA config. Backward compatabiity for /etc/certs will be provided.
5. Envoy will connect to the local agent and read key material via SDS

The address is injected by kube-inject or auto-injector, based on the namespace and pod labels and associated Webhook config.

Running the agent locally with istiod running in a Kubernetes cluster

ISTIO_ADDRESS=mypilot.mycluster.io ./istio-agent

Or “systemd restart istio-agent”, using the /var/lib/istio/envoy/sidecar.env file used by the .deb file.

Largely as above except that connectivity to the control plane uses an explicit address. In this case it is also possible to use ACME certificates for the control plane. The agent will still need local JWT token or certificates.

(note that current .deb is using PILOT_ADDRESS, and we’ll support this for backward compat)

Design

Phase 1 - Frankenstein's Monster

A first cut implementation will ‘bolt’ together the existing individual components (Pilot, Galley, Citadel, …) into a single istiod binary and hide the internal complexity of their dependencies by defining an improved user experience around the configuration model.

The current Istio startup is a complicated dance:

- Citadel must start first and generate certificates for the other components.
- Galley start watching Apiserver (or filesystem)
- Pilot needs Galley to be ready before it can start
- Autoinject may start as soon as it has certificates - but after it starts injected pods will not work until Pilot is ready
- Pilot has flags to configure the connection to Galley, depending on MTLS settings
- When SDS is enabled, a daemonset is deployed - which depends on Citadel, and is a dependency of the other components

There are benefits on having each component able to run standalone as a microservice - but it is common practice to combine multiple micro-services in a binary and use in-process calls. Depending on requirements and goals each option has benefits.

In this proposal istiod will:

- ~~Chiron~~ K8S issued certificates generation will start first, if the certificates are not already provisioned for istiod by a user-defined Secret/CA. Istiod will continue to lookup and use Citadel or custom signed certs. Kubelet provisions the Apiserver public key on each Pod automatically, so connecting to Pilot/SDS will work immediately, without dependency on mounting Citadel secrets.
- Citadel **(Root CA Generation, Cert Generation)** will generate the SPIFFE root and perform secret mounting as it does today to support in-place migrations. Root CA generation will be done by the operator, installer - or by istiod using the current leader election or watching the Secret to avoid install conflicts (i.e. what citadel does).
- Galley (**Configuration Watch)** will start at the same time, loading configs from k8s or files
- istiod will use an in-process MCP call to read config into Pilot (**xDS Serving)**, without GRPC serialization (also avoiding the size limit of the message), and possibly wait for 'isReady' from Galley. Since the connection is in-process no authentication is needed.
- **Auto-Injection** will also be in the same process. It is low cost and will be able to take advantage of the caches and config provided by Galley and Pilot.
- The features of istio-agent will be built into istiod as below

… and istio-agent will be built by:

- Moving **Local SDS** & **Private Key Generation** from Nodeagent into istio-agent (nee PilotAgent). This will eliminate the NodeAgent binary from the deployment topology.
- **Envoy Bootstrap** generation from istio-agent (nee Pilot-Agent) will become dependent on reading configuration from istiod instead of processing state stuffed by the injector into the pod environment. This serves several goals:
    - Simplified injector - no need to marshal mesh settings into the pod and unmarshal back. Instead a pod will only need to know the address of istiod.
    - More dynamic startup: right now bootstrap is created at pod startup, stays static.
    - Istiod will have full view of the configs and mesh status - so it can better generate the right config.

The combined component will be self-contained, and multiple versions can run at the same time in the same or other namespaces to enable upgrades and canarying etc.

The initial implementation will serve endpoints as they are today backed by SPIFFE certs on their existing Pilot/Galley/… named K8S Services. This is done to support in place upgrades from pilot-agent+node-agent to istio-agent. These endpoints will be deprecated in favor of the 'istiod.istio-system' endpoint described below.

A separate endpoint backed by a K8S issued cert will be served on 'istiod.istio-system' that exposes all control APIs: XDS, MCP, SDS, Cert Signing, Bootstrap support etc. Any component

in the cluster can use the kubelet-mounted apiserver certificate to connect.

We want SDS to be the default and only option for reading secret material in Envoy. The features of Citadel for writing certificates as secrets to namespaces should be disabled in istiod as a result. We still need to support a migration from that style of deployment as a result. Dealing with SPIFEE signing and distribution of merged root of trusts - will be enabled in istiod. Note that the proposal to holistically enable SDS relies on JWT based authentication to the control plane. Kubernetes has two forms of JWTs (legacy & trustworthy) that it generates depending on the versions. To support migrations we will likely need to support both forms of JWT which is not expected to be particularly expensive to implement. Initial prototype should focus on trustworthy JWT.

Phase 2 - Deep Cleaning

The above phase only mandates a bolting together of the bits and any refactoring strictly necessary to make things work. This phase starts to clean out extraneous dependencies and unsupported use-cases from the stack to make the system more efficient. Some examples of this kind of work:

- Removing K8S code dependencies from xDS serving or Cert Generation so that all configuration is read via MCP either in or out of process
- Removal of dependency on SPIFFE cert generation for control plane connectivity and bootstrap. Rely on K8S or external CA only. Both are supported in Phase 1
- Eliminate the separate K8S Service declarations and reduce to just ‘istiod’
- Move Consul and other adapters to a strictly MCP or K8S API based model (endpoint slice)
- <add here>

Phase 3 - Pipeline Refactoring

Assuming we’ve gotten to this point successfully any further changes to the system will have little to no externally visible impact to the end-user of the system. Work in this phase can broadly be categorized as engineering hygiene, efficiency and maintainability work. There are some patterns that have been adopted in Galley for example in order to facilitate an orderly composition of ‘services’ within a single binary. Such work would be useful as there may be pressure to create streamlined istiod builds because of bloat issues (see Appendix)

Installation, Configuration & Operator Impact

It is expected with this simplified deployment model that a number of major simplifications become possible in installation, operation, upgrade etc. Examples here include:

- Basic control plane unit is now defined by { mesh.yaml, CRDs, cluster roles & bindings, istio-system namespace}. Since istiod will bootstrap based on K8S CA and use this as the basis for control plane communication less sequencing of installation operations is needed. In addition we would ~~constrain the available configuration permutations currently provided in~~ remove values.yaml, and use the much smaller set in mesh.yaml. This should make the new installer more applicable to current use-cases and helm and the operator considerably simpler overall
- Canarying an upgrade can be done with a single deployment unit running in the same namespace as the primary. Canarying becomes a deployment pattern like:
    - Start a ‘istiod-canary’ deployment and service, using a mesh-canary.yaml config
    - Annotate deployment / namespace to indicate that bootstrap should be performed against istiod-canary service. We currently do this using namespace labels or pod annotations.
    - Trigger pod updates to make effective
- Permission requirements in installation are considerably reduced
    - No demonset
    - No PodSecurityPolicy to enable node path mounts
    - No permissions required to read/write secrets in arbitrary namespaces
- Setup for multi-cluster environments can be codified as configured behaviors or ‘modes’ of istiod as opposed to the current lego bricks approach. See ‘minions’ above.
- Setup for non-K8S environments like VMs becomes a single binary using either istiod or istio-agent depending on setup.

Concretely we have the following layering in the configuration artifacts and their responsibilities

- **mesh.yaml** - Configures the behavior of features within istiod. Is schematized and validateable. This resource can be represented as a configmap, CRD or local file as needed by the deployment. It contains no artifacts related to the orchestration of istiod. The initial version will be based on MeshConfig-v1 which is expected to evolve into the more structured schema as defined here (minus the orchestration artifacts)
- **istiod.yaml** - A vanilla K8S deployment artifact that can be used to install and run istiod as a control plane. It should work well with kustomize.
    - A variant of the above intended for use with helm - with empty values.yaml and no templating !
- **istio-operator.yaml (nee istio-control-plane.yaml)** - A configuration used by the operator to orchestrate and manage an Istio deployment. Has artifacts for orchestration

and version management but defers feature configuration to mesh.yaml. Will add ‘ingress’, telemetry, CRDs, webhooks, etc.

In general administrators of istio will either work with an operated install (mesh.yaml + istio-operator.yaml) or an unmanaged install (mesh.yaml + istiod.yaml). No other configuration artifacts should be required for the control plane.

PoC(early, including K8S certs and experiments with upstream envoy instead of istio-proxy): costinm/istiod and https://github.com/istio/istio/pull/17944

Build, Test & Release Impacts

Assuming we achieve a successful POC and a decision to choose istiod as the supported deployment artifact and configuration modality we would expect to see the following changes occur:

- Integration testing framework would become significantly simpler as it would primarily only need to orchestrate istiod, istio-agent, Envoy and any downstream components like Prometheus.
- A similar change would be expected for e2e test. For e2e in a shared cluster it should become simpler to create multiple control-plane instances to test different configurations and topologies
- Test and build times substantially reduced. Currently there is a 5 minute overhead to build all of the docker images (which happens 20x per PR), which we could expect to be reduced to below a minute. Additionally, startup time of Istio is likely to be reduced substantially - this typically takes almost 2 minutes and is run many times per test suite.

Other Impacts

There is no expected impact on telemetry and policy as those features are already migrating to WASM / Envoy plugins.

Kiali and related addons are also not part of this proposal and should not be overly impacted

Operating modes

Istio installer replacement

A ‘istiod.yaml’ file - consisting of plain k8s config, with no helm template - will be used to deploy istiod in a cluster. The config will be an in-place upgrade for the standard Istio install - users who choose this will have a single ‘istiod’ deployment, without the extra values.yaml customization.

VM/Node/Out-of-cluster

Istiod can also be deployed as a .deb or docker image running on a VM or a node, like apiserver - without running directly in the managed cluster. This is ideal for cases where the managed cluster runs tests or priviledged workloads that may take over istio-system.

Remote clusters

A particular mode of out-of-cluster is having Istiod run on a small set of isolated clusters (with proper replication and geo diversity), where no other workload is running, but managing a set of less-trusted clusters where workloads run.

Non-K8S clusters

Groups of VMs or isolated VMs can run the istiod-agent, which is not attempting to connect to K8S and has no deps except MCP and local files. In this mode the config can be provided by remote ‘istiod’ in k8s clusters or alternate sources implementing MCP.

Reliability

Overall the simplifications proposed here should provide a reliability improvement for the system. In particular it should greatly improve the first-use experience where dependency complexity causes issues today. Examples

- The standalone component will depend only on Apiserver, and tolerate as before transient Apiserver failures - but no longer be affected by startup ordering or other components.
- Networking/certificate issues between components and dependency on Citadel are eliminated.
- For memory use, the combined component will eliminate the duplicated caching and serialization and latency between Pilot and Galley.

- There are SPOFs for startup of components. In 1.3 each component may fail and we are also exposed to communication problems (certs, etc) between components. In the new model we eliminate some of the cross-component communication.
- Autoscaling of Pilot will continue to be based on memory/CPU - injector and SDS servers will scale along with Pilot and as such will be highly over-provisioned.

Upgrade

A post-job or ‘replicaCount=0’ will be used to disable the 1.3 injector. The Injector service will point to the combined binary. Alternatively we can leave the old components running, and use operator or manual cleanup.

Performance & Scalability

Most deployments only need 1 Galley MCP server (replica) and 1 injector - but in practice we may run 2-3 for reliability. We run more pilot replicas - few 1000s endpoints per pilot, depending on config size, isolation, etc. User can also use external MCP sources - and they will be used by ‘istiod’ as before. ~~For example, we’re planning to move the Consul adapter as an external MCP source.~~ The ‘file’ and ‘k8s’ config sources are currently implemented in Galley, and ‘k8s’/’Consul’ endpoint sources from Pilot will continue to be in-process for ‘istiod’. In future we may further optimize/refactor - but for this document we’ll attempt to use the most stable and well tested component and minimize behavior changes, the focus is on packaging and startup.

In the new model MCP and autoinject will be exposed from a highly replicated service, so over-provisioned. There is a risk that if Pilot is under stress we may also affect pod startup - however this will result in a delay on pod creation, while Pilot copes with the load, and so helping pilot to recover. MCP will only be used from remote clusters, so it’s less of a concern.

~~The combined binary will still use the envoy sidecar - and may load balance / shed load, including to remote clusters.~~ A Gateway running Envoy can provide load balancing, traffic shifting and other services to ./istiod, so it doesn’t need to run a sidecar itself. For multicluster and mesh expansion we are already exposing the Pilot using the Gateway, for reachability and stable address. The Gateway can be programmed using normal Istio APIs.

In terms of failure modes, the workloads are designed to tolerate failures of Pilot and Galley by continuing to use the current configuration. This will not change.

Injector failures are causing new pods to not start - however if Pilot is down, new pods will not be able to start (no config). So a Pilot failure is cascading to Injector - combining Injector with Pilot doesn’t make things worse.

Certificate generation is similar - if Pilot or Injector are down but Citadel is up - new workloads will fail, so adding parts of Citadel to Pilot doesn't make things worse.

Test Plan

This proposal is making changes to the startup - but no functional changes. All existing integration tests should continue to cover the functionality of all microservice, as well as end-to-end tests. The only observable change should be that only one deployment will run, so tests that exec or scrap individual components will need to change.

Changes

Appendix A - Modes vs Component Binaries

The existing deployment topology of Istio uses a binary-per-component model and this proposal argues that there is a large class of deployments where such a model creates unnecessary complexity. The binary-per-component model is effectively a proxy for binary-per-functional-role within the aggregate control plane responsibilities. An analysis is warranted to assess whether the bundling of features into istiod causes:

- Unsustainable bloat (see Appendix B)
- Latent code represents a security risk
- Coupling of functional roles that must be separated

… in use-cases that are outside the sweet spot identified in the main proposal. Any such use-cases are likely to be drawn from more complex deployment topologies. The following is a list of some more esoteric deployment topologies:

- **Multi-cluster with a centralized control plane**. Istiod's in each cluster push config to a centralized Istiod via client-initiated MCP. The centralized istiod cannot reach into each cluster because of firewalls. This topology would primarily be used when the centralized istiod is managed by a vendor.
- <add here>

Appendix B - Binary Bloat vs Tailored Builds

One concern with a monolithic binary model is whether the packaging of unused dependencies for certain use-cases represents ‘excessive’ bloat. One example use-case would be a VM-only installation based on files or MCP but where the binary still contains K8S API dependencies.

There are a number of viable options that can be taken that represent varying amounts of effort. The first is to simply ignore it because the bloat is considered too small to be consequential and the relative simplicity of distributing and documenting a single binary outweigh other costs.

The alternative is to identify a small number of key use-cases & deployment topologies where the value of streamlined binaries is considered high enough to warrant the additional build, test and release complexity. One such use-case is well established today, the use of istio-agent (nee pilot-agent), as an adjunct to Envoy. This is done because the distribution scale is one-per-pod and the cost relative to a pod is high (e.g. KNative). Similarly we expect to support istio-agent for VMs connecting to a centralized control plane service as a common deployment topology.

This leaves the following potential examples where a stripped down binary may have utility:

- Sidecar agent
- Permissions-free configuration management - only MCP/XDS, using workload identity instead of SA with cluster-wide RBAC
- SDS server - proxy and support for CAs that can handle the JWT directly, proxy for istiod or other SDS servers.
- Pilot generation code - so the Istio config to XDS can be generated locally
- In future: proxy XDS, with local snapshot and patching for advanced use cases.

The only ‘controversial’ point here would be the pilot transformation code and XDS proxy (UDS from envoy, connecting to istiod for configs over MCP and endpoints): it’s relatively easy to add and doesn’t harm too much in binary size. I think there are some use cases - in particular around large scale and ‘untrusted’ clusters. It is also important as a way to validate and test that Istio can work with MCP servers and local mode - most of our testing is based on this already so it’s not a huge overhead to support.

In addition it the nomenclature for a feature-stripped binary should be a derivative of the ‘istiod-xxx’ naming convention that indicates the intended use-case for the binary as opposed to naming the features that have been removed. E.g

- istiod - Kitchen sink
- istio-agent - Just has the sidecar agent included
- istiod-no-k8s - No K8S dependency

Proposal

Start with istiod and istiod-agent as these are well understood use-cases and the expected degree of bloat is low for K8S use-cases. Defer shipping or supporting other tailored binaries until a better assessment of the cost of bloat can be performed.

Changes in deployment topology in K8S may also warrant looking at a tailored binary in the future. E.g. running istiod as a NodeAgent instead of as a centralized service. That transition would be justified based on availability arguments. Likely best addressed in a follow on design document.

Appendix C - mesh.yaml vs command line flags

There are few benefits:

- A file is easy to edit, can have extensive comments for each field
- Version-control friendly
- CLI flags and env are one of the main reasons we need helm - which takes a config file (values.yaml) and converts it to flags with different names
- Config reload is already supported for mesh.yaml and could be supported for options we choose to expose.
- On a VM, changing CLI flags is difficult with systemd. In most cases they end up in a file that is read by the systemd unit.
- It avoids ‘3 different ways to do a simple thing’
- If the config is stored in a configmap as today, K8S dashboard is a very easy UI to edit settings.
- We eliminate the cobra/viper dependency and quite a bit of code that exists just to copy flags to structs. And even more code to copy from a file (values.yaml) to another file (deployment.yaml) so the flags can be copied back into the struct - with slightly different names at each stage.
- Mesh.yaml is already defined as a proto

We will continue to support the env variables, for backward compatibility and for environment detection, as well as for experiments and ‘not ready yet’ features.

We expect 2 classes of env variables:

- well-defined /standard like KUBECONFIG, POD_NAMESPACE, etc - including some that are currently documented for Istio.
- Experimental/customizations/vendor specific options that are not yet fit for the long-term supported mesh config. Those should be clearly documented as not-long-term. Some env variables may be promoted to mesh.yaml once they're stable.

A section in mesh.yaml will include the unstructured 'env:' (as currently used in values.yaml for many components). The library that abstract env-variable reading can merge 'real' env with mesh.yaml overrides.

One feature of K8S is to allow loading a set of env variables from a ConfigMap - Istiod and istio-agent will mount an optional user-maintained config map with env overrides. This will further reduce the need for helm and templates.

Appendix D - Future Deployment Variations

As we evolve this direction there are a number of other topologies that are worth considering:

- **Istio-agent as demonset with CNI/CRI** In this case we package the CNI/CRI handling into the istio-agent binary and run it as a demon within the cluster. In this mode the istio-agent would also provide **{Private Key Generation, Envoy Bootstrap, Local SDS Server}** to Envoy instances running on the same node. There would still be a centralized istiod that the demons would connect to for {**xDS, Certificate Signing}**
    - A variation of the above is to use full **Istiod as demonset with CNI/CRI** - There would still be a centralized istiod that the demons would watch for config via MCP and would use for certificate signing. The central istiod will have access to istio-system secrets and handle global endpoint discovery - might run in 'isolated cluster' for extra security. The per-node would run with lower RBAC privileges, delegating to central istiod for elevated operations. Central istiod can verify the per-node credentials and check that the subject pod is assigned the node. This mode will allow larger scale deployments, since the most expensive part (config generation) will be distributed, with central istiod only distributing configs and endpoints without generation.
- <add here>

The choice to support some of these topologies would influence our decision to build tailored binaries - see discussion on bloat.