Large Scale Data Ingestion using Istio/Envoy

Animesh Chaturvedi
Distinguished Engineer, Palo Alto Networks
@Ani_Chaturvedi / animesh@apache.org

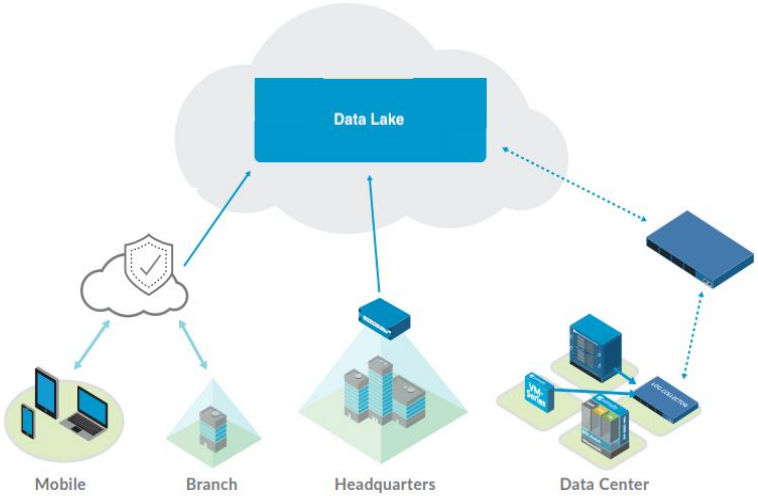
Agenda

- Building a Data Lake
- Why Istio / Envoy
- Challenges
- Fixes
- Results

Building a Data Lake

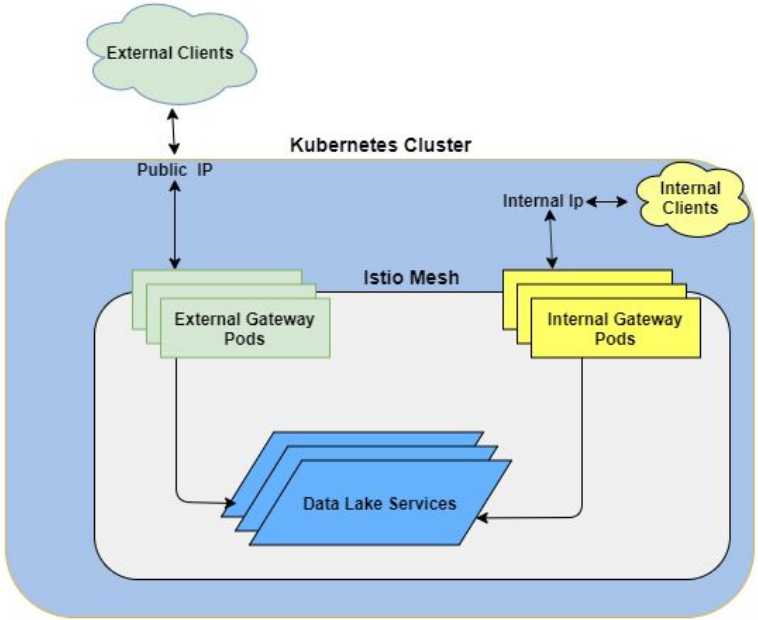
Palo Alto Networks was building a data lake in early 2019

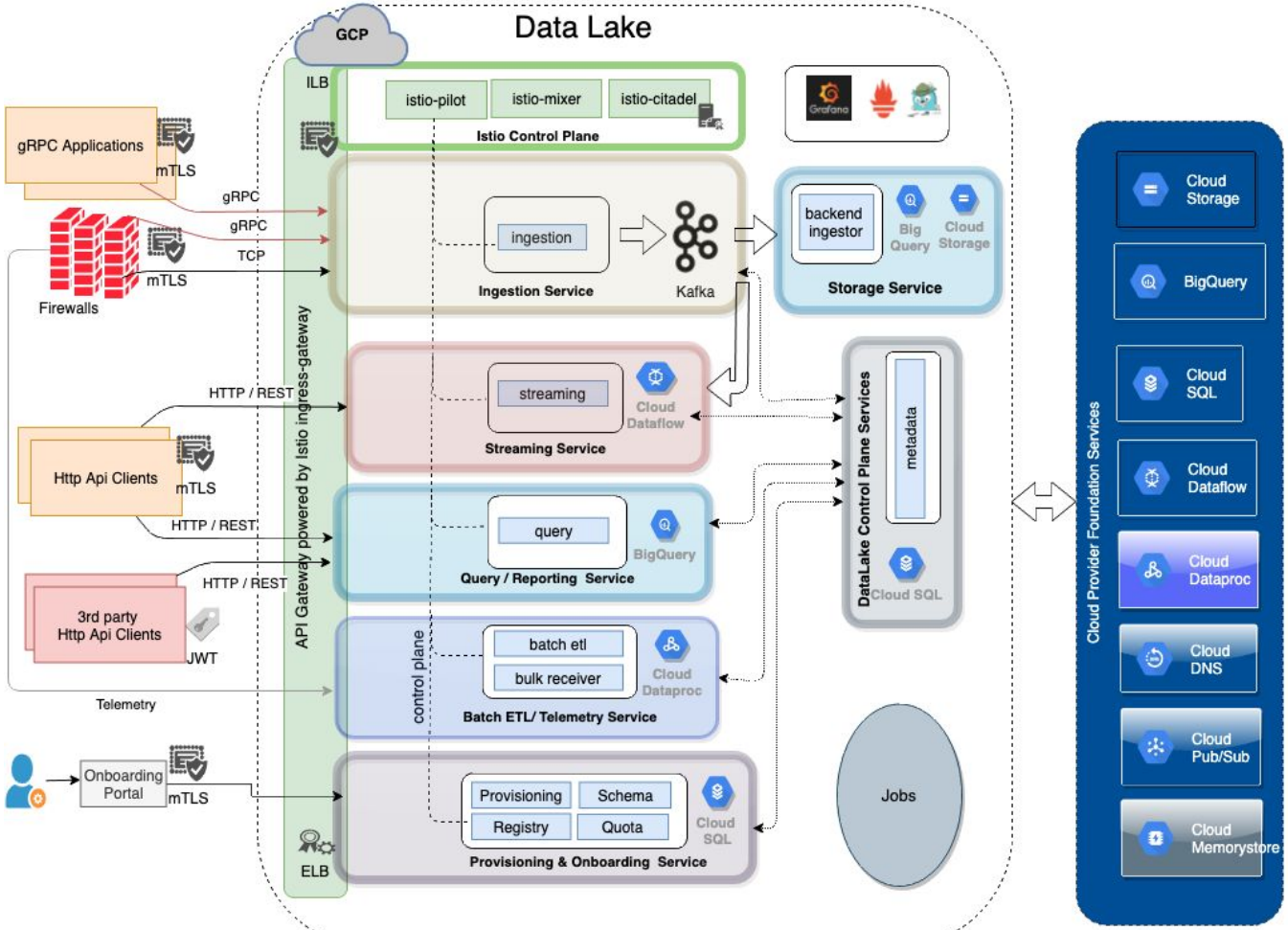
- Store and process logs, telemetry and miscellaneous data from firewalls and applications
- Cloud Native
- API driven
- Massively scalable
  - Few hundred billion events/day
  - Streamed continuously
- Multi tenant
- Microservices architecture
- Efficient Operations
- Best Devops Practices



Building a Data Lake

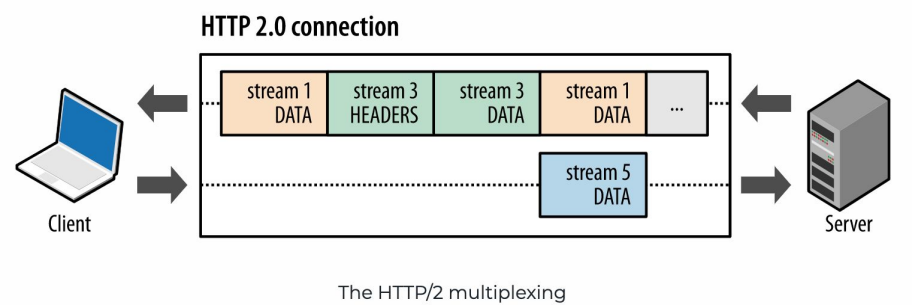
- API Gateway and Service Mesh
- Services
  - Ingestion
  - Query
  - Stream Compute
  - Batch Compute
  - System Services
- Authentication via mTLS / JWT




Data Lake
GCP
ILB
istio-pilot
istio-mixer
istio-citadel
Istio Control Plane
gRPC Applications
mTLS
gRPC
gRPC
TCP
mTLS
Firewalls
ingestion
Ingestion Service
Kafka
backend ingestor
Big Query
Cloud Storage
Storage Service
HTTP / REST
streaming
Cloud Dataflow
Streaming Service
Http Api Clients
mTLS
HTTP / REST
query
BigQuery
Query / Reporting Service
HTTP / REST
3rd party Http Api Clients
JWT
Telemetry
API Gateway powered by Istio ingress-gateway
control plane
batch etl
bulk receiver
Cloud Dataproc
Batch ETL/ Telemetry Service
Onboarding Portal
mTLS
Provisioning
Schema
Registry
Quota
Cloud SQL
Provisioning & Onboarding Service
ELB
DataLake Control Plane Services
metadata
Cloud SQL
Jobs
Cloud Provider Foundation Services
Cloud Storage
BigQuery
Cloud SQL
Cloud Dataflow
Cloud Dataproc
Cloud DNS
Cloud Pub/Sub
Cloud Memorystore

gRPC

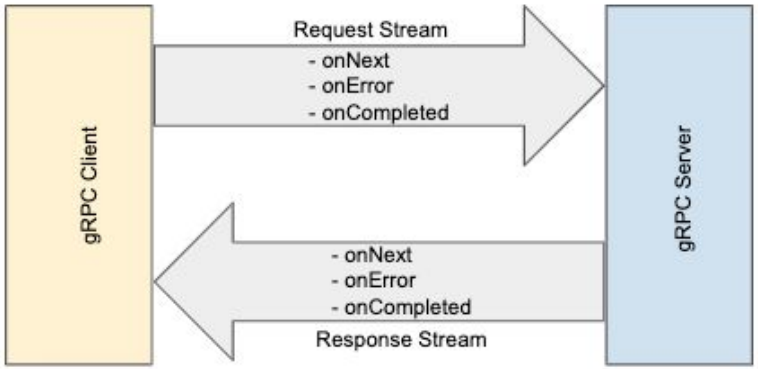
- Created by Google as open source evolution of Stubby
- Uses HTTP/2 as its transport protocol
- HTTP/2 can multiplex many parallel requests over the same connection
- Allows full duplex bidirectional communication
- gRPC provides access to HTTP/2 flow control capabilities



The HTTP/2 multiplexing

gRPC Stream Observer

- Client and Server implement StreamObserver interface for sending and receiving messages



gRPC Streaming modes

#	Dimension	Unary	Bi-Directional
1	Throughput	Relatively Low	**High**
2	Latency	Relatively high	**Low**
3	Overhead	Relatively high	**Low**
4	RPC Call	Synchronous	Asynchronous
5	Blocking	Synchronous RPC call blocks until a response arrives from the server.	**Asynchronous RPC call does not block for any response from the server**.
6	Header	Header per message	**Header per stream**
7	Retries	**Retries per message**	No retries per message. Retry per stream
8	Load Balancing	**Unary RPCs on Channel will be load balanced across all the backends**	The client and server application must implement load balancing

Why Istio / Envoy

- Firewalls are connected all the time and streaming events continuously so we decided to use gRPC
- Envoy was the only proxy that supported gRPC at that time
- Istio was the preferred control plane providing
  - API Gateway
  - Service Mesh
- Istio handles cross cutting concerns for MicroServices deployment
  - Application Networking
  - Security and Policy
  - Observability
- Istio codifies best Devops practices
  - Timeouts
  - Retries
  - Circuit Breakers
  - Fault Injection
  - Load Balancing

It worked until it didn’t

Pipeline worked fine until 200 thousand events per second / 15 billion events per day

But then…

Wedge and get stuck with cascading failures at higher load

Challenges

- At high load critical failure in any components in ingestion pipeline resulted in cascading failures
- Istio 1.1.7
  - Very large connections and streams from gRPC client overloaded Istio
  - This caused Istio telemetry trigger load shedding and limit scaling
  - Citadel / Pilot bugs caused mesh disruption
- GKE Instability
- gRPC bidirectional streaming
  - Not well documented
  - Many performance features are marked experimental
  - In-depth knowledge required for tuning

Fixes: Istio Upgrade

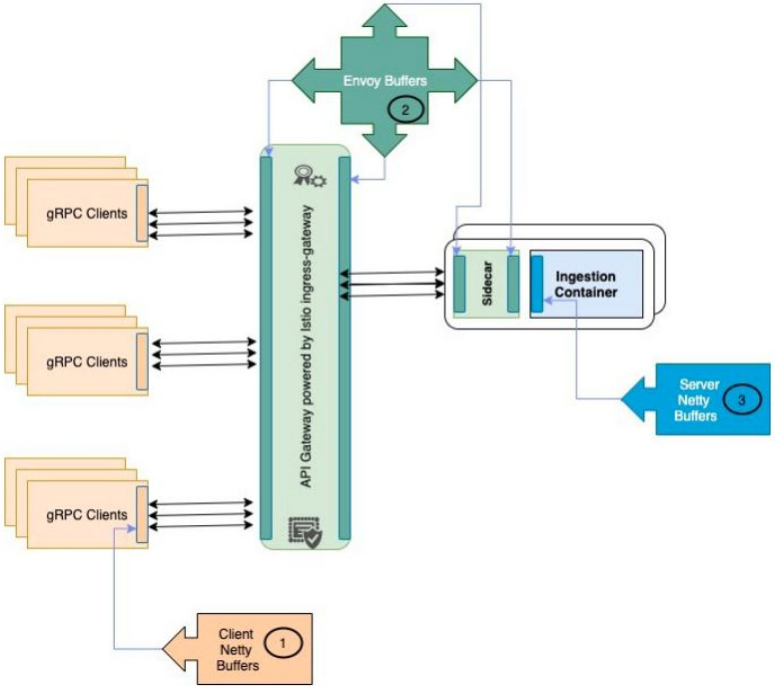
- Upgraded to Istio 1.1.15
  - Fixed several critical issues from Istio 1.1.7 and earlier
  - Turned off Istio telemetry and using Envoy native telemetry
  - Resource tuning for some Istio components
- gRPC client tuning
  - Reduce number of streams and connection
  - Use flow control signals with onReadyHandler

Fixes: Ingestion gRPC Server

- Too many threads causing thread contention: Reduce number of threads
- Large Queues: Enqueuing more work than can be processed
- java.security.Provider.getService() synchronization became a scalability bottleneck at high load: Upgrade to JDK13
- Kafka producer client tuning
- gRPC stream buffer tuning

Fixes: Tune stream buffers

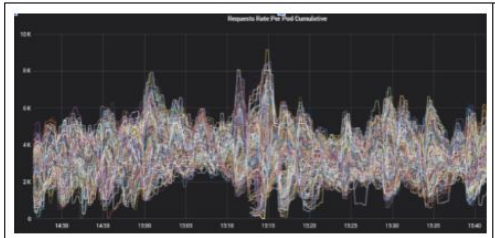
#	Stream Buffers	Description
1	gRPC Client Netty Buffer	Default value is 1 MB and can be configured using NettyChannelBuilder#flowControlWindow. We settled on 64KB buffers for our pipeline.
2	Envoy stream buffers • API Gateway Envoy inbound and outbound Buffer • Envoy sidecar inbound and outbound buffer	The relevant attributes are specified in Envoy as Http2ProtocolOptions • Initial_stream_window_size • Intial_connection_window_size • Max_concurrent_streams Configuring these Envoy attributes via Istio requires setup of EnvoyFilters Default Envoy Http2 buffer window size is 256MB which was very large for our use case and we had to tune these buffers for our application. We settled on 64KB buffers for our pipeline.
3	gRPC Server Netty Buffer in Ingestion Container	Default value is 1 MB and can be configured using NettyServerBuilder#flowControlWindow. We settled on 64KB buffers for our pipeline.



Stream buffers in the pipeline

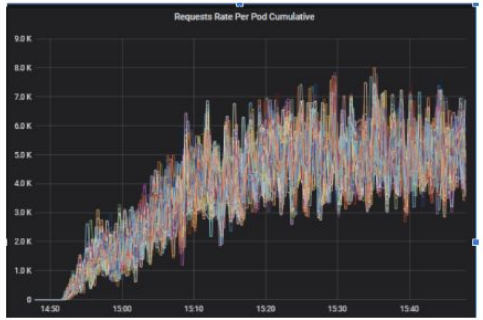
Fixes : gRPC load balancing

- gRPC connections are long lived and results in uneven load distribution across pods and does not play well with Kubernetes HPA
- Expire connections at server
- Tune stream duration



Stream Duration:
Time based 3 minutes

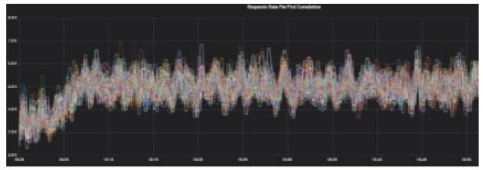
Pod Request Rate
Range: 1000 - 8000



Streams are limited to 1000 requests

Pod Request Rate
Range: 3000 - 7000

Smaller streams results in a much narrower band and distribution across pods resulting in better HPA characteristics



Streams are limited to 500 requests

Pod Request Rate
Range: 4000 - 6000

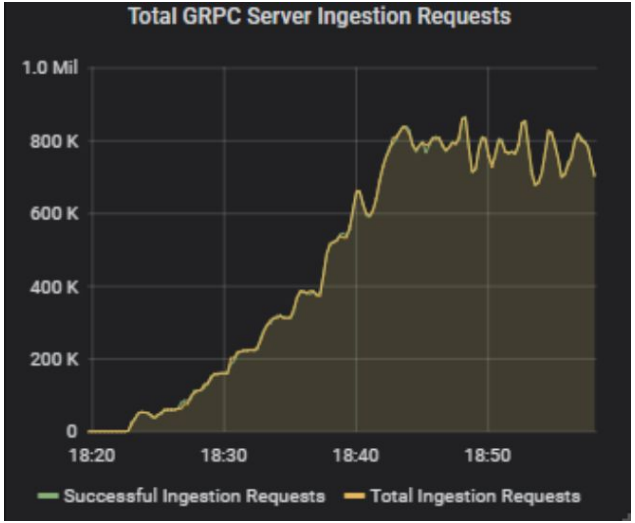
With streams of 500 requests, the band becomes even narrower and dense. This is the selected configuration

Fixes: GKE

- Node Kernel Tuning
  - Low conntrack and threadMax limits: We increased CONNTRACK_MAX to 2 million, CONNTRACK_HASHSIZE to 0.5 million, and THREAD_MAX bumped to 4 million.
- IO Throttling
  - Docker Daemon and Kubernetes became unstable
  - Moved workload to node-pools with SSD
- Node memory exhaustion
  - Workload resource tuning

Results

- Istio has helped us sail through in our journey to build the DataLake
- Resilient and massively auto scalable pipeline
- Handles few hundred billion events in a day
- Operationally efficient
- Launched data lakes in many regions
- Many upstream applications with variety of analytics and AI/ML functions have now integrated with the data lake



Questions ❓

#istiocon @ istio.slack.com

 @Ani_chaturvedi

More details covered in my medium post

Thank you!

- @Ani_chaturvedi
- animesh@apache.org