A Stream Data Platform For Video Delivery Telemetry

Michael Bevilacqua-Linn
Comcast

Abstract

As video delivery systems transition to all IP, they are becoming massively scaled distributed systems. Comcast's video delivery systems span hundreds of sites, and failures in one site can have unexpected negative imapcts on video delivery. When troubleshooting these sorts of systems, it's useful to have as broad, but often shallow, operational view of the system as a whole.

In addition, these systems generate data essential for business intelligence, capacity planning, reccomendations, and a whole variety of other essential functions. Without a system to methodically collect data from across this infrastructure, data collection is usually done via a set of ad-hoc integrations, usually with log files, or, at best, custom telemetry collection schemes. This leads to a stew of ever-shifting data formats, which much be parsed and reconciled to make sense of the system as a whole

In this paper, we present an architecture for a stream data platform which allows us to comphrehensively collect high value data, for both operational and other business purpouses, at large scale. In addition, we present a method of defining and evolving schemas for this data.

INTRODUCTION

Large scale distributed systems are notoriously tricky to reason about and debug. Our current methods generally assume that only the engineering or operations team responsible for an individual component in a larger distributed system has visibility into that component.

When an incident occurs, those engineering and operations teams are often brought together in a conference bridge, chat room, or email chain of doom. Using parochial tools and data collection, the individuals responsible for the troubleshooting effort attempt to communicate as best they can about complex issues, often without the ability to effectively share data.

The data that's collected for these parochial toolsets often comes in the form of unstructured logs, which are under the control of an individual engineering team. They may change the format of those logs at any time to better meet their needs. As this is the often only data collection source for a given component, point-to-point integrations between log collection systems and other systems which produce business value, such as business intelligence clusters, customer care tools, and so on.

As the primary motivation for the initial data collection is parochial, and the logs are not considered part of the contract that the component has with the outside world, this often causes much strife. A development team may add to, modify the semantics of, or remove elements from the log file at will.

Additionally, the point-to-point integrations between systems are expensive, and involve time consuming, and error prone, ETL from one system to another.

These issues can be overcome with the addition of a stream data platform, which is intented to summplement existing parochial data collection systems. The stream data platform has several main goals.

First and foremost, it serves as an extremely high volume, resilient, real time data bus for data which needs to leave the parochial confines of an individual component. When a stream goes into the stream data platform once, it can be used by many other systems which provide business value. For instance, a stream of video starts from an IP player may be used for business intelligence reporting, operational tooling, customer care tools, reccomendations, and so on.

Second, it provides a means to associate a schema with streams of data, and a means to evolve the schema for that data without breaking existing consumers of it. A schema is essential to aid consumers of the data in their data integration projects, however, in recent years, schemas have fallen somewhat out of style in the big data ecosystem.

As the ecosystem matures, this is changing. However, the schema technology needs to be more flexibile than traditional RDBMS schemas. In particular, a single stream may be consumed by many consumers, and not all of them will be willing or able to update their schema when a new version of it is created. Through a combination of technology choices and process, we enable schemas to be evolved by a data producer independent of any data consumer.

Third, the stream data platform provides ETL tools which can take raw streams of data, and transform them into clean streams of data once at ingest time. While we prefer, and recommend, that a source system directly emit clean data conforming to a schema, it's often impossible or impracticle to modify an existing system to do so. Doing this ETL once at ingest into the platform is still a win over doing it in an ad-hoc, case by case bases for point-to-point integration projects.

Fourth, the stream data platform will provide tools for data governance, security and discovery. Through the platform we must be able to secure an individual stream of data, through authorization before the stream can be accessed, as well as data security methods such as encryption of the data while it's at rest in the platform, and tokenization of individual fields. This security must be tied in with reasonable data governance policies, which specify what types of data need which security policies.

For example, a data stream which has elements in it which, when combined with other data sets, may constitue PII may need to both be encrypted while at rest, and have the the sensitive fields tokenized, to limit the risk of a data breach and ensure regulatory compliance.

Subject to data governance and security concerns, any user of the system should be able to discover the data sets, associated schemas and other metadata. This will ensure that users can find existing data streams and have a holistic view of the data available.

As of the writing of this paper, we mainly address the first three concerns of the stream data platform, and have done experimental work on the fourth.

THE ELEMENTS OF A STREAM DATA PLATFORM

The stream data platform has several elements. We have chosen to build the platform on top of open source software, primarily from the Apache foundation, along with some home grown components.

First, a system for high capacity data transfer. For this, we have chosen Apache's Kafka, a distributed messaging system which is highly optimized to ensure that all messages can be persisted to disk. This is essential to ensure that many consumers, all of which may not be consuming at the same time, can access

the datt streams. We have been scaled Kafka to a point where an individual cluster can handle over one million messages per second, without much tuning or effort, and are confident we will be able to scale it to meet our throughput targets as we grow the system.

A key element in Kafka is the topic, which represenents an individual data stream. We have tooling to create topics conforming to a naming convention, so that we can separate out different types of data streams

Our Kafkas are organized into ingest clusters, which ingest some subset of data from across our entire footprint, and larger aggregate clusters, which aggregate data from ingest clusters. Producers produce data to an ingest cluster, and consumers consume from an aggregate cluster. For resillency purpouses, we provide both ingest and aggregates in multiple, geographically isolated datacenters

Second, we have chosen Apache's Avro as our schema language. Avro allows us to create a schema for a given event, and also provides a set of libraries which allow us to serialize and deserialize data conforming to that schema in a variety of languages.

We have built a home-grown schema mananger that allows us to associate a schema with a topic, and evolve that schema. In the next section, we will describe the evolution process in more detail.

Third, we have chosen Apache's NiFi for basic data cleansing and ETL. While there will always be a need for advanced ETL, both at the batch and stream level, outside of this system, we decided that the capability to provide simple ETL was essential.

We use NiFi only to take streams of data that are in some non-standard format, such as raw logs, ad-hoc JSON, XML, etc, and put them into a standard format with a schema. NiFi was the right tool for this job because the vast majority of these transformations can be done by stringing together and configuring existing NiFi processors, without the need to write custom code. This makes it possible for analysts to do this ETL without needing engineering assistance.

Fourth, we provide a home grown HTTP ingest system. This is stood up alongside the ingest Kafka clusters, and allows us to work with systems which would be difficult or impractical to ingest a Kafka client into. This includes CPE and COAM devices.

The HTTP ingest system allows for two rough modes of operation. First, we allow data that comes into it in a clean format. This data is placed directly into a clean Kafka topic.

Second, we allow data that comes into the system in a raw format. This could be some legacy CSV, XML, JSON, etc. This data is placed into a raw, staging topic. From there, we use our data cleansing tool, NiFi, to clean and wrap a schema around the data. This allows us to avoid the need to duplicate data cleansing functionality in the HTTP ingest layer, as well as NiFi.

The final important platform we will provide in the stream data platform are tools for data security, discover, and governance. We are still early on in the selection of this toolset, but we have a good idea of some of the properties that we'll need.

Our data security tooling will need to allow us to enforce and configure two types of security policies, both of which we want to manage centrally.

First, we'll need to be able to configure access and authorization policies. This will allow access to topics on a per topic basis. Second, we'll need to configure data security policies. This is much more difficult, but at a

high level, these policies will allow us to encrypt streams of data, both at rest and in flight, and allow us to tokenize sensitive fields in the data.

Note that encryption at rest is not sufficient data protection. While at rest encryption will protect the data from filesystem access, assuming that the key management system has not been compromised, doing any analysis of large amounts of data will require it to be decrypted.

This means that many analysts, and analysis systems, will be authorized to access the data. If any of those systems, or people, are compromised, they will be able to decrypt and acquire large amounts of sensitive data.

The solution to this problem is to tokenize the most sensitive pieces of data, such as account identifiers, and other pieces of data which, when combined, could constitue PII. This must be done in a consistent way, so that analysts can still do joins, between datasets.

When done consistently, this level of tokenization will protect from de-ananomyization attacks associated with large scale data breaches.

Hand in hand with these security tools are our data governance and discovery tools. These tools will allow us to view the schemas for different data streams, view sample data for data streams (subject to security policy).

Basic, lightweight data governance, such as ensuring that clean data streams have a schema that conforms to basic guidelines, and so on, are essential to the ensuring that the stream data platform serves as a landing ground for high quality data, and not just a dumping ground of data streams.

SCHEMA DEFINITION AND EVOLUTION.

Our schema definition and evolution processes form the backbone of our data governance policies. We'll cover both in detail here, starting with schema definition.

We define certain core schemas, including distributed trace metadata, a common header format, common data types, such as account and device identifiers, and so on.

Schemas for a specific data stream are composed of these core schemas, in addition to data fields which are specific to that stream. Our data governance policy is centered around ensuring that we're making appropriate use of these schemas.

Wherever possible, we will automate data governance checks. For instance, we'll automatically validate schemas, check for required fields, such as docstrings, and so on. All of this will be done in a self-service manner, and be backed by a lightweight manual check.

Once a schema is created, it will be placed in the runtime schema management system we described in the last section. From there, a single producer system will write data into the stream data platform, conforming to the schema. Many consumers may then read and work with the data.

Producers must have the ability to evolve their schema independent of any consumer, as we cannot force new shcemas on highly distributed set of consumers. In order to do so, we take advantage of Avro's schema compatability feature. Avro allows for a reader schema, which is the schema used by the data consumer, and a writer schema, the schema used by the data producer.

Avro strictly specifies, and Avro libraries enforce, what it means for a reader schema to be compatible with a writer schema. The exact semantics can be found in the Avro

specification, but as a motivating example: a writer schema may have a field added to it, and remain compatible with an existing reader schema. However, a field can be removed if and only if a default was provided for it in the original schema.

While Avro's schema compatability feature strictly specifies what it means for a reader and writer schema to be compatibile, it does not have semantics to deal with changing a schema over time.

For instance, as mentioned earlier, it's "safe" to remove a field with a default from a writer schema. In this case, existing consumers will see the default value for the field.

However, it's possible to make a series of safe modifications that result in incompatible schemas. For instance, a field with a given name could be removed in one revision, then added back in in a subsequent revision, with a different type.

Consumers with the original schema would no longer be able to parse data produced by the new schema, as they would see data produced with a different type than they were expecting. At the time of this writing, we rely only on Avro's schema compatability, but we expect to extend this through our schema management system to enforce compatability through multiple schema versions.

SUMMARY

Comcast's stream data platform allows for the collection of telemetry data, at scale, and for a wide variety of purpouses. To do so, we provide a high capacity data bus, using Apache Kafka, as well as schema management and data ingest systems. Furture work will center around advanced schema management, data governance and security