Distributed Trace For Video Systems

Michael Bevilacqua-Linn
Comcast

Abstract

Modern video delivery systems, especially those invoved in IP video, are large scale distributed systems. They are composed of many collaborating subsystems, written by different teams, often in different technologies. Comcast's IP VOD system, for instance, comprises dozens of different subsystems involved with manifest generation, licensing, encryption, caching, advertisement and other alternate content insertion, user interfaces, etc.

These systems have no central coordinator or statefull session manager and are often located in geographically disparate areas. An error can occur in a system several hops behind the player, causing a video to fail play. Correlating that error with the error observed by a user in a highly distributed environment is very challenging, as is determining how much latency is induced by various parts of the system.

This paper describes how a method for distributed trace, based around a protocol and library named Money, an annotation-based distributed trace protocol with roots in Google's Dapper and Twitter's Zipkin. We describe the Money protocol and how we're using it inside of Comcast for our video delivery systems.

INTRODUCTION

Distributed Video Systems

Distributed systems are a fact of life in modern system design. There are several reasons this is so:

- Subsystems can be scaled independently of one another. If built appropriately, this can be done horizontally on commodity hardware.
- Subsystems can be built by small, focused development teams, without the need for much coordination overhead.
- Subsystems can be built using a variety of architectures and techniques, which makes it possible to build a new subsystem using state of the art methods, rather than being weighed down by years (or decades) of history.

Of course, nothing in life is free. One of the primary costs that come with large scale distributed systems is that understanding the system as a whole is extremely challenging.

Developers of an individual subsystem may understand their subsystem and its interfaces, and may even have some understanding of subsystems that call them, and that they in turn call. However, it's unlikely that they could explain the system as a whole.

The below sketch represents a fairly standard, though incomplete, architecture for an IP VOD system. Comcast's full production infrastructure contains many more subsystems, however this sketch is sufficiently demonstrative of the complexities involved.

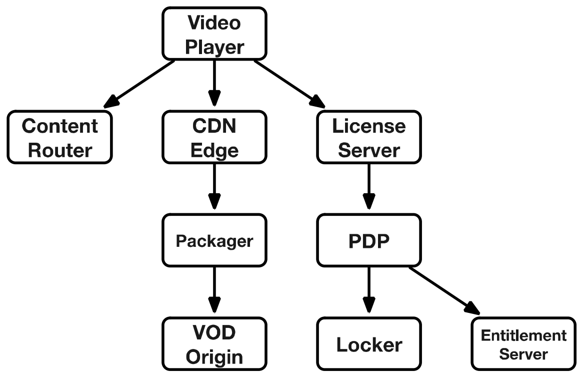

Figure 1 – Sample Architecture Sketch for IP VOD

The sample architecture contains several different classes of subsystems. First, there are players for every platform that video is delivered to, such as STBs, iOS, Android, etc. Second, there's a CDN comprised of a Content Router, Edge Nodes and Origin Nodes.

There's a Packager which produces ABR streams in various formats, and finally content protection infrastructure made up of a License Server, Policy Decision Point, which determines whether a given license request should be allows, a Locker which contains information on purchased assets, and an Entitlement Server which contains information on an account's entitlements.

Generally, no one team understands the system as a whole. Even when an architecture team is involved to provide oversight and higher level design, important, often gritty, details of system to system interaction are poorly understood.

In particular, two questions that are of great interest to engineering and operations teams are fiendishly difficult to answer in such a complicated system. The first is "Why did this video fail to play?", and the second "Why did this video take a long time to start?"

Let's examine failures first. Even assuming that every individual subsystem has a well defined set of error conditions that allow an operator of the subsystem to immediately identify and fix an issue, an assumption that already borders on fantasy, how is does an operator of the overall system determine that a particular failed playback was caused by a subsystem several network hops away from the player?

A common approach is to start by assuming that the player is at fault, forcing the team responsible for operating it to spend time figuring out which upstream system threw the error so that they can pass the bucket of water on. This process repeats itself, going deeper into the system, until the culprit is eventually found.

This is a slow, expensive process that generally involves ticketing systems, conference calls, and the occasional heated conversation.

Attempts to make troubleshooting easier generally follow a few paths. Sometimes, an attempt is made to pass detailed error information from the depths of the system to the front, which forces all subsystems in the path to understand deep implementation details of the subsystems they themselves rely on. This violates the encapsulation of those subsystems, and rarely ends well.

Another, more successful approach is to correlate metrics and log data from various systems using an existing identifier and a rough time period. For instance, it may be possible to pull together telemetry and log data for a given account, device combination over the course of a few seconds, and assume that they were all done on behalf of the same playback session.

The drawbacks to this approach are that they require subsystems to have access to the necessary identifiers, which may not be the case, not every system needs an account identifier to passed to it.

A more complete solution is to have the player create a unique session identifier for a video playback session and pass it through the call chain. Each subsystem then attaches the session identifier to all emited operational metrics, so that they can be correlated by data processing systems.

With appropriate data collection systems in place, this gives system operators a complete view across all the system as a whole, and makes it possible to determine where in the call chain an error occurred. However, a single session identifier makes it difficult to understand anything about individual service to service interactions within the session, they must be represented in an ad-hoc manner in emitted metrics, or known a-priori by operators.

This becomes more import when we're trying to understand where latency is induced in video is induced in video startup. While it's easy to record an overall timing from the player, it's difficult to understand which subsystems interactions are responsible for inducing latency, and how much, even if the individual subsystems are performance tested and their latency characteristics are well understood.

In real world situations, . In order to do so, timing metrics must be collected from every client-server interaction, from both the client and server sides. A single session identifier is insufficient for that level of granularity. In the next session, we'll examine a mechanism for tracing requests through a distributed system that does provided the needed granularity.

Tracing a Distributed System

The sequence of events in any complex request through a distributed system can be modeled as a graph of interactions between the individual subystems. Using the reference architecture in Figure 1, starting a session would looke something like Figure 2.

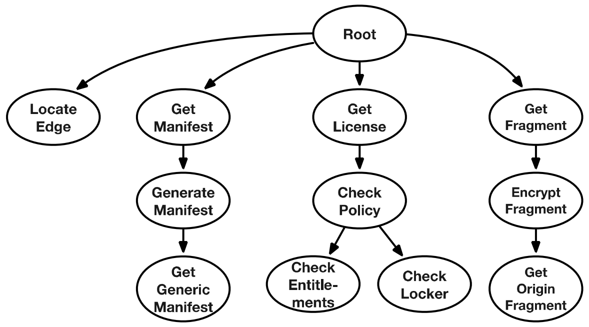

Figure 2 - Partial Call Graph For IP VOD

Here, every node in the graph represents an interesting interation between two subsystems, such as the player locating an appropriate CDN edge node from the content router, or the license server checking a Policy Decision Point to decide whether or not to allow a license request.

The goal of a distributed trace applied to IP video is to reconstruct the graph of calls that takes place to satisfy a particular video session so that useful telemetry can be attached to the individual nodes in the trace. In figure three, we add timing information in to the .

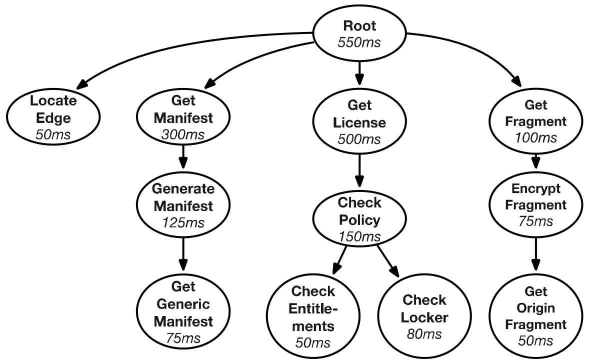

Figure 3 - Distributed Trace With Timings

SHOW ME THE MONEY

Traces and Spans

The main abstractions in a Money distributed trace are Traces and Spans.

A trace is generally scoped to all operations and interactions that take place to satisfy some use interaction, in our case, starting a video and the ensuing series of fragment requests that take place to keep it going. The call graph in Figure 3 represents an entire trace.

A span represents some interesting interaction in a trace, generally between two subystems in the distributed system, and associated metadata. In Figure 3, spans represent calls between systems which collaborate to start VOD playback.

A Minimal Implementation

The core of the Money protocol is small.

Trace identifiers are strings, which are expected to be GUIDs such as those specified in RFC-4122. Span identifiers are 64 bit signed integers, which are expected to be psuedorandomly generated at the creation of a new span.

Trace and Span identifers are passed from system to system on an HTTP header, `X-MoneyTrace`, which takes the following form, linebreaks are inserted for clarity:

```
X-MoneyTrace:
trace-id=$TRACE-ID;
parent-id=$PARENT-ID;
span-id=$SPAN-ID
```

Here, the `trace-id` field contains the GUID which identifies the overall trace, `trace-id` contains the 64 bit integer which identifies the current span, and `parent-id` identifies the parent of current span. It must be included so that a casual relationship between the two spans can be maintained throughout the trace.

The subsystem at which the trace is rooted, which will always be the player in our case, creates a Trace identifier. It then creates Span identifiers and corresponding an `X-MoneyTrace` headers for every request made to second level systems. For spans at the root, the `parent-id` and `span-id` are set to the same value, to indicate that the span is its own parent.

As the request progresses through the system, each subsystem must parse the `X-MoneyTrace` header and make its identifiers available as context for the subservice. When one subsystem makes a request to another, it creates a new Span identifier to encompass the new span, and creates a new `X-MoneyTrace` header with the appropriate Span and Trace identifiers.

On the return path, the `X-MoneyTrace` header is passed back as a response header. This makes it easy for systems which cache HTTP responses to keep track of the Trace that originally generated the response.

This small protocol is all that's required to participate in a Money distributed trace.

Data Collection and Processing

Once distributed trace data is being passed through the system, it can be used to tag existing log messages, or existing application level telemetry. We have found it useful to collect the following base set of telemetry across applications.

Metric	**Description**
span-id	ID for the current span, a Long
trace-id	ID for the current trance, a GUID of some sort

parent-id	ID for the parent, a Long
span-name	Operator readable name for the span
start-time	Start time for the span, logged as GMT
http-response	If the span encompasses an HTTP request/response, the response code
span-duration	The duration of the span, in microseconds
response-duration	If the span encompasses an HTTP request/response, the response duration, from the point of view of the application doing the logging
error-code	When the span did not complete successfully, an error code must be populated here
span-success	Whether or not the span completed successfully, the string “true” or “false”

A good starting point for metrics collection is to use existing application logs and emit a single log line containing span metadata as a set of key value pairs.

CONCLUSION

Modern IP video delivery systems tend to be highly distributed and complex. Operating such a system is fundamentally different from operating many traditional video delivery systems in that there is no centralized orchestrator that has a full view of the system.

We can get this overall view of the system using a simple distributed trace protocol to pass trace context through the system, which can then be used to tag existing telemetry and log data.

REFERENCES

Google Dapper - http://research.google.com/pubs/pub36356.html