TECHNICAL PAPER

Rock Solid Live Streaming

Building a fault-tolerant streaming deployment with HDS and HLS

By Glenn Eguchi

Senior Computer Scientist

February 2013

SUMMARY

In the real world of live streaming, accidents happen at the worst times. During the final seconds of the big game, a server will crash. Just before announcing the winners of the grand prize, an intern will unplug the network cable. When it counts, your organization must be able to rely on your live streaming system to tolerate and gracefully handle failures.

The goal of this whitepaper is to teach you how to deploy a live HTTP Dynamic Streaming (HDS) or HTTP Live Streaming (HLS) setup for maximum fault tolerance. It begins with a simple, demo-quality deployment. From this basic setup, you will learn how to progressively add redundancy and fault tolerance measures, until finally, you end up with a rock solid live streaming deployment.

TABLE OF CONTENTS

Adobe Systems Incorporated
345 Park Avenue, San Jose, CA 95110-2704 USA
www.adobe.com

IS THIS PAPER FOR YOU?

Before you get started, take a minute to make sure this is the right paper for you.

This is an intermediate to advanced level tutorial.

You should already understand how to set up a basic HDS live or DVR deployment end-to-end. Specifically, you should have a working knowledge of Adobe Media Server (AMS), the associated HDS and HLS Apache modules, and the Open Source Media Framework (OSMF). For more information see the Adobe Media Server Developer's Guide.

This paper only covers the live and DVR use cases.

This whitepaper does not cover the details of a Video on Demand (VOD) system.

This paper assumes that you will employ a Content Delivery Network (CDN).

If you aren't planning to use a CDN, you should strongly consider doing so, since CDNs specialize in efficient delivery of resources to end users. In the proposed architecture, the CDN will serve as an HTTP cache layer. It will cache responses from origin servers and serve those responses to clients to take maximum load off of the origin servers. The main focus of this whitepaper is how to create a robust, CDN-compatible origin deployment.

Are you still there? **Great! Let's get started.**

Download the sample files for this whitepaper (ZIP, 5 Kb)

BACKGROUND

It will help to begin with some background of how HDS works *under the hood*. By peeking into the HDS black box, you'll gain a deeper understanding of the elements involved in constructing an optimal deployment.

Bootstrap

Many areas of this whitepaper discuss a mysterious file called the ***bootstrap.***

The main job of the bootstrap is to list the available fragments for a given asset. The bootstrap contains two data structures that help it perform this duty: the ***fragment run table*** and the ***segment run table.***

The fragment run table lists the fragment numbers of all the available fragments. It also lists the time intervals that each of those fragments cover.

The segment run table lists the segment numbers for every available fragment.

By combining the information in the fragment run table and the segment run table, an HDS client can map time values to fragment URLs. This mapping is central to the operation of HDS, as you'll see below.

Live Playback Workflow

Stepping through a simple live HDS playback workflow will help you see how the bootstrap fits into your system.

To initiate playback:

1. OSMF fetches the manifest file or files.
2. OSMF fetches the bootstrap file.
3. OSMF searches the fragment run table for the fragment with the most recent timestamp.*
4. OSMF uses the fragment number, the segment run table, and the manifest to determine the URL for the fragment.
5. OSMF fetches the fragment and then buffers it for playback.
6. OSMF inspects the bootstrap to determine the next available fragment. OSMF fetches and buffers that fragment. OSMF repeats the process.
7. Once there is enough buffered data, OSMF begins playback.

During playback:

8. OSMF periodically refreshes the bootstrap so that it is aware of new fragments as they become available.

9. OSMF fetches and buffers fragments as needed to maintain a sufficient buffer.

**Actually, OSMF begins playback with a fragment that is a few seconds older than the most recent timestamp. The rationale for this behavior is explained later in this whitepaper in the First Steps – Misallaneous Configuration Tasks section.*

FIRST STEPS

This section covers some initial configuration steps that should go into a simple HDS origin deployment. Even if you've been exposed to these concepts before, it's a good idea to review this section to ensure that you haven't missed anything.

If you haven't already done so, set up an HDS origin deployment consisting of a single encoder and a single packager (see Figure 1).

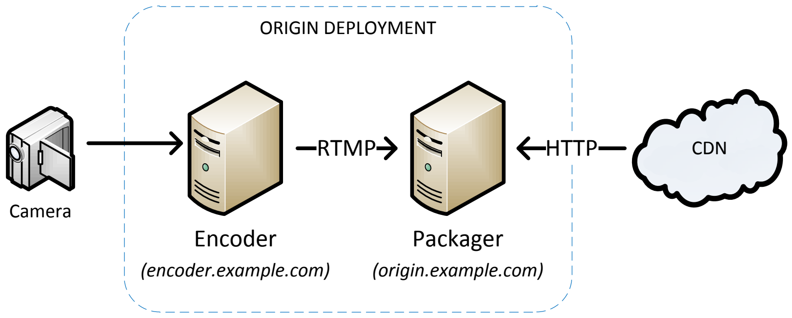

Figure 1. A simple origin deployment.

In this deployment, the encoder (hosted at encoder.exaple.com) will receive the camera feed and then push an RTMP content stream to the packager (hosted at origin.example.com). The packager will package the content for HDS distribution via the livepkgr application, and then make it available to the corresponding Apache modules. The CDN will service client requests for HDS assets, requesting non-cached content directly from the packager as needed. Incoming requests will be serviced by the packager's origin module.

If you do not know how to set up this deployment, refer to "Getting started streaming media / Stream live media (HTTP)" in the Adobe Media Server Developer's Guide.

Configure HTTP Cache-Control

Correct utilization of HTTP caches is an essential ingredient in creating a scalable HDS origin deployment. If you configure your origin caching behavior correctly, most client requests will be serviced by intermediate HTTP caches (such as your CDN), and your deployment will scale better. If you configure your caching behavior incorrectly, most requests will be proxied to your origins, resulting in poor scalability.

In HDS, the primarily mechanisms for controlling caching behavior are the HttpStreamingF4MMaxAge, HttpStreamingBootstrapMaxAge, and HttpStreamingFragMaxAge configuration parameters in httpd.conf. These parameters modify the HTTP Cache-Control: max-age and Expires headers that are emitted from the packager for the manifest, bootstrap, and fragment requests respectively. Intermediate HTTP caches use these headers to determine how long they can cache assets.

The goal is to tune max-age to be just right; max-age should be long enough so that intermediate caches don't serve stale data, but short enough so they will still absorb most of the load.

Each of the various HDS file types requires a different max-age to ensure the proper behavior:

- Fragment requests may be cached for a long period of time. Once a fragment has been generated, it will remain the same forever.
- Bootstrap requests should only be cached for a short period of time since the bootstrap file is updated every time a new fragment is available. Setting HttpStreamingFragMaxAge to half of a fragment duration is recommended. This will ensure that the intermediate caches periodically refetch the bootstrap, and clients are made aware of new fragments.
- Manifest requests may be cached for a long time. The manifest typically remains unchanged for the duration of the event.

  NOTE: Cache lifetimes for objects may change in future releases.

Edit your httpd.conf file to set the appropriate values, assuming a fragment duration of four seconds (4s):

```
<Location /hds-live>
    …

    # Manifest: 1 day
    HttpStreamingF4MMaxAge 86400

    # Bootstrap: 1/2 a fragment duration = 2s
    HttpStreamingBootstrapMaxAge 2

    # Fragment: 1 day
    # We have chosen to specify an explicit value instead of the default -1.
    # -1 omits the Cache-Control headers, which is less preferable than an
    # explicit timeline.
    HttpStreamingFragMaxAge 86400

</Location>
```

Now, verify that your packager is emitting the expected values:

```
$ wget -S http://packager.example.com/hds-live/livepkgr/_definst_/liveevent.f4m
  …
  Cache-Control: max-age=86400
  Expires: Thu, 14 Jun 2012 22:45:45 GMT
  Content-Length: 285
  Last-Modified: Wed, 13 Jun 2012 22:45:45 GMT
  …

$ wget -S http://packager.example.com/hds-
live/streams/livepkgr/streams/_definst_/livestream1/livestream1.bootstrap
  …
  Cache-Control: max-age=2
  Expires: Wed, 13 Jun 2012 22:50:58 GMT
  Content-Length: 4473
  Last-Modified: Wed, 13 Jun 2012 22:50:56 GMT
  …

$ wget -S http://packager.example.com/hds-
live/streams/livepkgr/streams/_definst_/livestream1/livestream1Seg209-Frag209
  …
  Cache-Control: max-age=86400
```

```
Expires: Thu, 14 Jun 2012 22:51:46 GMT
Content-Length: 414810
Last-Modified: Tue, 12 Jun 2012 00:23:40 GMT
…
```

(Tip: An easy way to get a valid fragment URL is to play the stream and inspect the logs.)

Enable Disk Management

Disk management is a feature designed to solve a few problems that commonly occur with long-running events.

Suppose you start a live event, leave it running, and then come back to check on your origin several days later. Without disk management, you might encounter a few problems:

- The packager might have run out of disk space because it has saved multiple days' worth of video fragments.
- The bootstrap file may be very large. Since the bootstrap is effectively a list of fragment names, storing entries for thousands of fragments takes up a fair amount of space. A large bootstrap file could incur additional latency and bandwidth usage on the client side.

The fix for both of these issues is simple: enable disk management. You can enable disk management by adding the following snippet to your livepkgr's Application.xml:

```
<Application>
    <HDS>
        <Recording>
            <DiskManagementDuration>4</DiskManagementDuration>
        </Recording>
    </HDS>
</Application>
```

This configuration tells your packager to set the disk management duration to four hours for all live events running in this application. AMS will delete fragments that are more than four hours old*, and it will delete entries from the bootstrap corresponding to old fragments.

Pay particular attention when sizing your disk management window. There are a few tradeoffs to consider:

- If the disk management window is too long, the bootstrap may grow to a large size and fragments may end up occupying more disk space than is optimal.
- If the disk management window is too short, this can affect fault tolerance. A good rule of thumb is that your disk management window should be reasonably longer than your DVR window (approximately one hour is sufficient).

For additional details on Disk Management, refer to "Getting started streaming media / Configure HTTP Dynamic Streaming and HTTP Live Streaming / Disk management" in the Adobe Media Server Developer's Guide .

Actually, AMS will ensure that each stream has at most four hours of content. This is a subtle difference that doesn't apply to most users. If you instead require AMS to always delete content that is more than four hours old, consider using the DiskManagementWithGap configuration parameter.

Use Absolute Time

HDS requires the usage of ***absolute time*** on both your packager and your encoder.

Absolute time is a timestamp scheme in which media timestamps are assumed to be relative to an absolute clock, as opposed to the time of connection. Under HDS, absolute time enables the system to correctly handle anomalies that can occur during encoder restarts and other special scenarios that occur in multi-packager deployments.

In practice, you don't usually need to worry about the details of what exactly absolute time means, and can treat absolute time as a black box. As long as you ensure that absolute time is enabled on both your packager and your encoder, HDS will work.

To configure your packager to use absolute time, ensure that the following configuration is present in your livepkgr's Application.xml:

```
<Application>
    <StreamManager>
        <Live>
            <AssumeAbsoluteTime>true</AssumeAbsoluteTime>
        </Live>
    </StreamManager>
</Application
```

To configure your encoder to use absolute time, consult your encoder documentation or contact your encoder vendor.

Miscellaneous Configuration Tasks

Finally, there are also a few miscellaneous configuration tasks that you'll need to perform for a production grade HDS system.

Configure your fragment duration to be an exact multiple of the key frame interval.

The bootstrap binary format is optimized to enable fragments of constant duration to be stored efficiently. If your encoder produces key frames at a relatively constant rate, you can take advantage of this optimization to help keep the size of your bootstrap down. You can configure the fragment duration in the Application/HDS/Recording/FragmentDuration element of your livepkgr's Application.xml. Remember to re-adjust other settings (like MaxAge) accordingly if you alter your fragment duration.

Make sure to use a live offset.

Having clients play too close to the live point in HDS can result in poor behavior on both the client side and the server side. Thus, it's a good idea to assign a live offset, which instructs the player to begin playback a specified number of seconds behind the live point. In OSMF, the live offset is configured by default to a reasonable value, but can be customized via the hdsDVRLiveOffset and hdsPureLiveOffset variables in org.osmf.utils.OSMFSettings.

MULTIPLE PACKAGERS

In this section, you'll learn how to add redundancy to your system by introducing two new elements. The first is an additional packager, which will improve load distribution and fault tolerance. The second is a reverse proxy between the packagers and the CDN (for reasons described later). The following figure illustrates the new architecture:

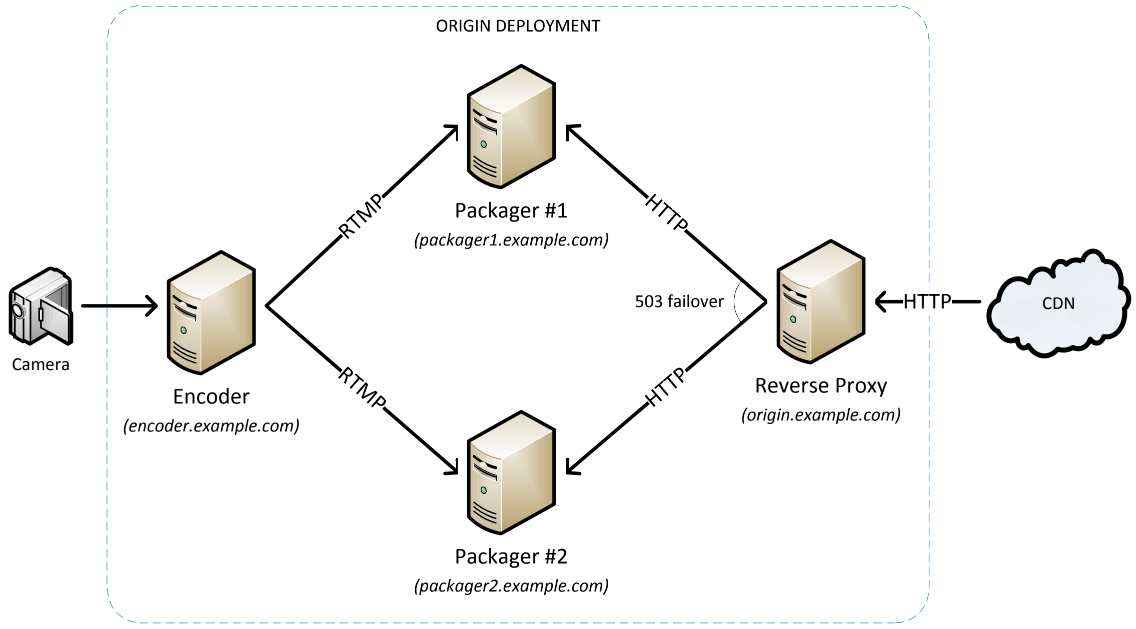

Figure 2. An example redundant packager deployment. Two packagers are hosted at packager1.example.com and packager2.example.com respectively. The encoder sends an identical RTMP feed to both packagers. A reverse proxy now acts on behalf of origin.example.com, sourcing its HTTP content from one of the two packagers.

Don't worry if you don't understand every part of this diagram yet. It will be covered in greater detail later.

Deploy an additional packager

First, provision a new machine for your packager and install and configure AMS on it. Instruct your encoder to publish the same event to both your original packager and your new packager. The event should have the same name on both packagers. If everything is configured properly, the new packager will produce fragments identical to those on your original packager.

Select a reverse proxy

The next component you will add to your system is a ***reverse proxy.***

A reverse proxy is a server situated between a pool of back-end servers and the CDN or end user. The reverse proxy acts as if it were the desired server, receiving HTTP requests and proxying those requests from a pool of back-end servers as needed. Reverse proxies often serve multiple roles in a system, acting as a cache, a load balancer, and a fault tolerance mechanism.

In this deployment, the reverse proxy will act on behalf of origin.example.com. It will be the public face of the origin deployment, residing between the pool of packagers and the CDN.

There are many different reverse proxy products to choose from, each with its own advantages and drawbacks. Ideally, the reverse proxy you select should support the following functionality:

- 503 Failover (described later)
- Response caching
- Rewriting an error response to a cacheable error
- Back-end selection based on URL (for HLS)

Subsequent code examples in this whitepaper make use of the Varnish proxy. Varnish was selected because it supports all of the functionality above. Varnish is only one of many possible proxies you can choose to deploy.

Once you have selected a reverse proxy, provision a new machine and install the reverse proxy on the new machine.

Configure the reverse proxy

The first step is to configure your proxy to forward requests to your back-end packagers.

In Varnish, this can be accomplished by adding the following code to your default.vcl:

```
backend b1 {
        .host = "packager1.example.com";
}
backend b2 {
        .host = "packager2.example.com";
}

director hds_round_robin round-robin {
        {.backend = b1;}
        {.backend = b2;}
}

sub vcl_recv {
        set req.backend = hds_round_robin;
}
```

The above script instructs the proxy to round-robin among between two packagers. If one packager fails, it will try the other one. This is a first step towards fault tolerance.

Problem: Packager race

There are a few subtle failure cases that can arise in this new system that will require special treatment. The first of these failure cases is a problem called the ***packager race***.

The packager race is best understood by example.

Consider a situation in which you have deployed your current two packager setup with a four-second fragment interval, when the following series of events happens:

1. At 1:00:00 PM:

   The encoder sends out a fragment's worth of video representing the four seconds starting at 12:59:56 PM.

2. At 1:00:01 PM:

   Packager 1 finishes receiving the video. In less than a second, it writes out a new fragment, and then updates the bootstrap to advertise the presence of the new fragment. Suppose the number of the new fragment is 100.

   Packager 2 experiences a short period of increased latency, which delays its receipt of the video. The latency will last for five seconds, meaning Packager 2 will not write out its new fragment or update its bootstrap until 1:00:06 PM.

3. At 1:00:02 PM:

   An OSMF client requests a bootstrap from the proxy (via the CDN). The proxy selects Packager 1 to service the bootstrap request. The client receives a bootstrap that advertises fragment 100 as available.

4. At 1:00:03 PM:

   The same OSMF client requests fragment 100 from the proxy. By chance, the proxy selects **Packager 2** to service the fragment request. Since fragment 100 is not available yet on Packager 2, the packager returns a 503 HTTP error to the proxy. The proxy then routes this error to the player.

5. The client, upon receiving an error, halts playback.

Solution: 503 failover

503 failover is a proxy configuration technique that provides a simple solution to the packager race problem, as well as several other similar problems that can occur. The technique works as follows:

1. When the proxy receives a 503 HTTP error from a packager, the proxy will attempt the request with the next packager in the pool.
2. Once any packager returns a valid response, the proxy will send that fragment to the player.
3. If all packagers return 503, the proxy should return a cacheable error (such as a 404). Set the max-age of this error to a short value (for example, half a fragment interval).

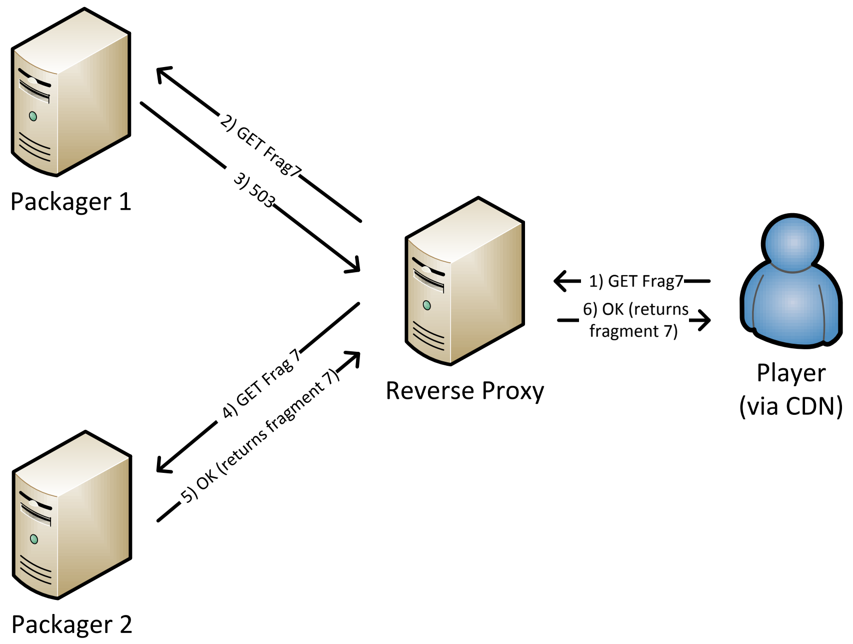

Figure 3. An example of 503 failover. (1) the player requests fragment 7. (2)The request is proxied to packager 1 (3), which returns a 503 error since it does not have the fragment. (4) The request is then proxied to packager 2 (5), which does have the fragment, and returns it accordingly. (6) The fragment is returned to the player.

503 failover endows your system with a useful property:

> **If the client requests a fragment, the client will receive that fragment as long as ANY back-end packager has it.**

Keep this property in mind; it will be exploited again later, when you adopt a technique called Best Effort Fetch.

In Varnish, you can configure your proxy for 503 failover with the following default.vcl code:

```
sub vcl_fetch {
    if (beresp.status == 503) {
        # we received a 503 error
        if(req.restarts < 1) {
            # try the next server in our pool
            return (restart);
        } else {
            # all servers failed, generate a cacheable 404 error.
            set beresp.status = 404;
            set beresp.http.Cache-Control = "maxage=2";
            set beresp.response = "Not found.";
        }
    }
}
```

Enable caching on the reverse proxy

It is often a good idea to perform caching in the reverse proxy layer.

Depending on your CDN, you may experience a nontrivial number of requests on your origin deployment. Adding a cache to your proxy layer may reduce the impact of this load on your system. Since the HDS Apache modules do not cache requests, and packagers are typically more resource constrained than proxies, transferring the burden of serving requests from the packagers to the proxies (via a cache) usually results in better overall system scalability.

If you are using Varnish as your proxy, caching is enabled on the command line and does not require a specific configuration.

Consider alternative load distribution schemes

In this example the load distribution scheme was round-robin, but be aware that this is not the only possible scheme. Under certain circumstances you may benefit from an alternate scheme. You may have a very large number of live streams, all of which will be packaged on all the servers. Or, you may have a very large number of proxies in your reverse proxy layer. In these situations, you could benefit from alternative schemes such as dividing your reverse proxies and packagers into multiple pools, or distributing load based on a URL hash and adding a secondary caching layer between your proxies and your packagers. A simple way to achieve this is to enable mod_cache on your Apache server.

BEST EFFORT FETCH

OSMF 2.0 introduces a feature called OSMF Best Effort Fetch (BEF), which is explicitly designed to improve the fault tolerance of a multi-packager HDS system. It specifically addresses two classes of failure that can occur: ***liveness*** and ***dropout***.

Enabling OSMF Best Effort Fetch

To enable BEF, add the bestEffortFetchInfo element to your manifest.xml:

```
<manifest xmlns="http://ns.adobe.com/f4m/1.0">
  …
  <!-- alter these values to specify your segment and fragment durations -->
  <bestEffortFetchInfo
       fragmentDuration="4"
       segmentDuration="4"/>
   …
 </manifest>
```

Upon seeing this element, the OSMF client will perform BEF behavior whenever necessary.

The remainder of this section covers what goes on under the hood when you enable BEF.

The Liveness and Dropout Problems

BEF addresses two specific classes of problems that can occur in a multi-packager deployment: ***liveness*** and ***dropout***. The best way to understand liveness and dropout is by example.

Liveness

Consider a simple multi-packager setup with two packagers and a reverse proxy configured for 503 failover. The fragment interval is one minute. For simplicity, this setup does not have a CDN and client requests arrive directly at the reverse proxy. Assume that this setup is serving a live stream that started a long time ago and the time interval 12:59 PM - 1:00 PM is represented by fragment number 100.

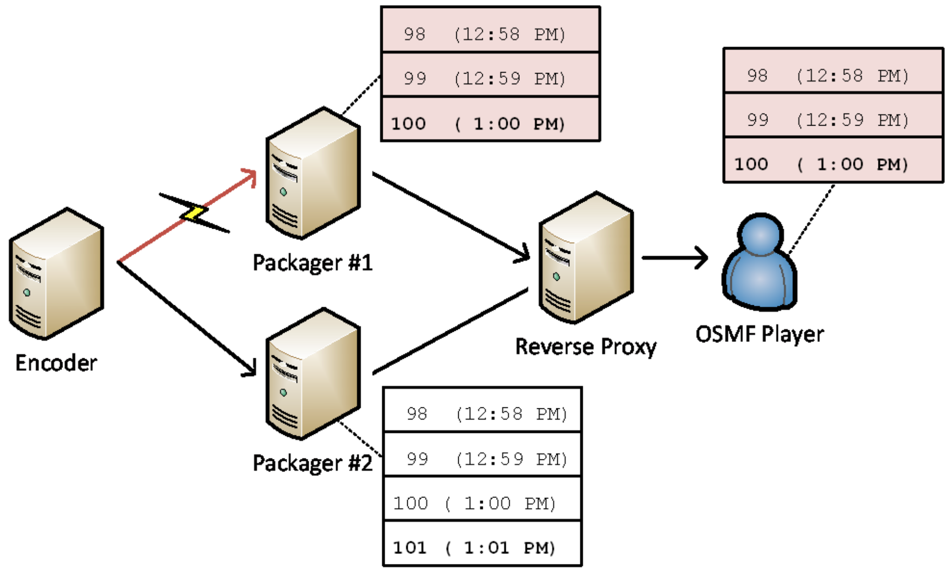

Figure 4. An example of liveness.

The following sequence of events then occurs:

1. Just before 1:00 PM, both packagers have a bootstrap that advertises the availability of fragments 100 and below.
2. At 1:00 PM, a network cable in the data center is unplugged. This results in a partial network disruption under which:
    - Packager 1 loses its connection to the encoder, but can still be reached by the proxy.
    - Packager 2 is not affected by the disruption. It continues to have healthy connections to both the proxy and encoder.
3. At 1:01 PM, a customer decides to play back the stream. The customer's OSMF player connects and fetches the bootstrap. At this time, the bootstraps on Packager 1 and Packager 2 differ.
    - Since Packager 1's encoder connection is still down, its bootstrap continues to advertise the older fragments it knows about, 100 and below.
    - Since Packager 2's encoder connection is healthy, it has received and packaged a new fragment. Its bootstrap now advertises fragments 101 and below.

By bad luck, the reverse proxy selects Packager 1 to serve the bootstrap. OSMF receives a bootstrap that only advertises fragments 100 and below.

4. OSMF begins playback offset from live, by starting at fragment 98. OSMF fetches and plays fragments 98, 99, and 100 from the server.

5. At 1:04 PM, OSMF approaches the live point (fragment 100) and re-requests the bootstrap. Again, the bootstraps on Packager 1 and Packager 2 differ.
   - Packager 1's bootstrap continues to advertise 100 and below.
   - Packager 2's bootstrap advertises fragments 104 and below.

   By bad luck, the reverse proxy again selects Packager 1 to serve the bootstrap. OSMF again receives a bootstrap that only advertises fragments 100 and below.
6. Since OSMF finds that there are no new fragments available, **live playback stalls**. The customer tolerates the stall for a few seconds, but eventually gives up and moves on to a different website.

The playback failed because the customer received a stale bootstrap due to an ongoing back-end failure. This class of failure is known as ***liveness***.

Liveness is a serious problem. In a real deployment that includes a CDN, liveness means that a single malfunctioning packager can spoil playback for all players. Liveness is not limited to partial network outages, but can occur in response to other failures such as process crashes.

The logical next step in the above example would be to fix the failed connection and enable Packager 1 to resume processing. Unfortunately, this results in a related failure called dropout.

Dropout

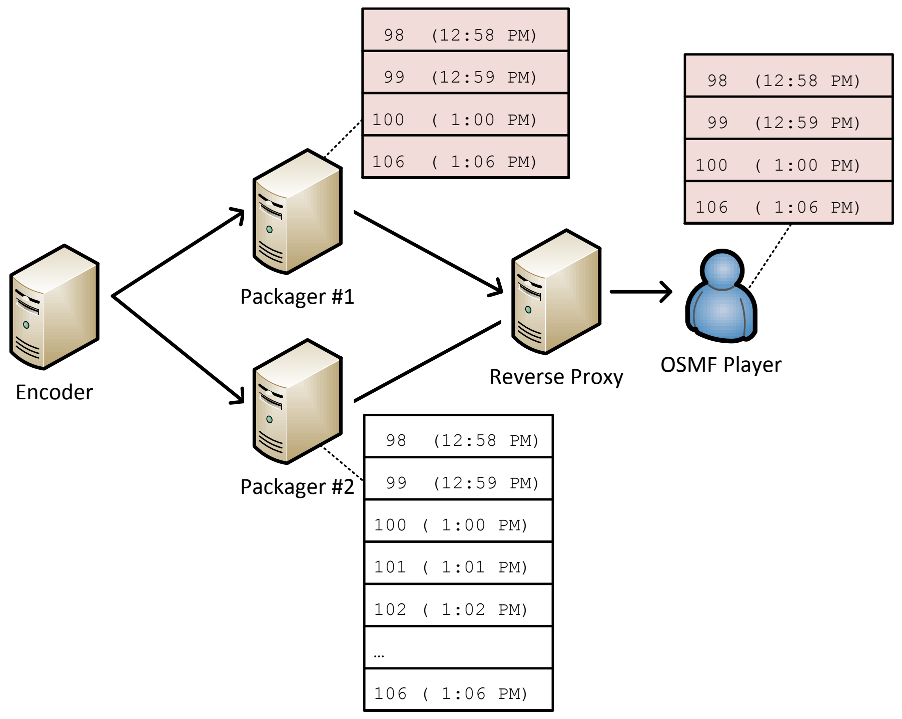

Figure 5. An example of dropout.

Continuing from the previous situation:

1. At 1:05 PM, after the outage of five minutes, the network disruption is fixed. Packager 1's network connection is restored.
2. At 1:06 PM, a new customer receives a call from a friend saying "You've got to see what happened at 1:00pm!" She starts OSMF and seeks to 12:58 PM within the DVR stream.
3. OSMF requests the bootstrap from the reverse proxy as usual. At this time, the bootstraps on Packager 1 and Packager 2 differ.
    - Packager 1's bootstrap advertises fragment 106, and fragments 100 and below. **The bootstrap does not advertise fragments 101-105.**
    - Packager 2's bootstrap advertises all fragments 106 and below.

   By chance, the proxy chooses Packager 1 as the source of the bootstrap.
4. OSMF begins fetching and playing fragments starting with fragment 98 (12:58 PM). OSMF fetches and plays fragments 98-100 normally. Content prior to 1:00 PM plays normally.
5. OSMF fetches and plays back the next fragment listed in the bootstrap, fragment 106. Since fragments 101-105 are missing from the bootstrap, they will not be requested.

6. **The new customer experiences a skip in video playback.** Because the video skips immediately from 1:00 PM to 1:06 PM, she misses exactly the portion of content she was most interested in.

This second class of failure, ***dropout***, is like its cousin liveness in that a single malfunctioning packager can spoil playback for all players.

Enter Best Effort Fetch

Best Effort Fetch provides a client-side solution to liveness and dropout. When best effort fetch is enabled, OSMF will perform additional requests for fragments that could be present on the back end, but have not been advertised by the bootstrap. Because of 503 failover, if the fragment is present anywhere on the back end, the fragment will be received on the client side.

While it may sound confusing now, Best Effort Fetch is actually a simple idea.

Best Effort Fetch with Liveness

Here is what happens when the liveness scenario is repeated with Best Effort Fetch enabled in OSMF:

1. At 1:01 PM, the customer requests the bootstrap. He receives a stale bootstrap that advertises fragments 100 and earlier, but is missing entries for fragments 101 and later.
2. OSMF begins playback offset from live. The player fetches and plays fragments 98, 99, and 100.
3. At 1:04 PM, OSMF completes playback of fragment 100. At this point, a normal player would have experienced a stall in playback.
4. **Instead of stalling, OSMF with Best Effort Fetch requests fragments beyond those specified in the bootstrap.** OSMF requests fragment 101 from the proxy.
5. Upon receiving the request, the proxy begins 503 failover:
    a. By bad luck, the reverse proxy first attempts to fetch fragment 101 from Packager 1.
    b. Packager 1 does not have fragment 101 so it returns a 503.
    c. The proxy then asks Packager 2 for the missing fragment.
    d. Packager 2 does indeed have fragment 101, and the fragment is returned to the player via the proxy.
6. Upon receiving fragment 101, OSMF plays back the content as usual.
7. OSMF repeats the process for fragments 102, 103, and so on. Best Effort Fetch enables the content to play back seamlessly despite the liveness failure.

   The customer is happy.

Best Effort Fetch with Dropout

Here is what happens when the dropout situation is repeated with Best Effort Fetch enabled:

1. At 1:06 PM, the second customer wants to play the time corresponding to fragments 98 through 105. Again, she unluckily receives a bootstrap that has a dropout between fragments 100 and 105.
2. OSMF begins playback from fragment 98, and successfully fetches and plays fragments 98 through 100. Upon completing playback of fragment 100, a normal player would have experienced a skip to fragment 106.
3. Instead of skipping, OSMF notices the discontinuity and attempts a best effort request into the dropout. OSMF requests fragment 101.
4. Fragment 101 is returned from Packager 2 via 503 failover.
5. The same process is repeated for fragments 102 through 105. Best Effort Fetch enables playback to continue smoothly in the face of problems with bootstrap.

   The second customer is happy as well.

Best Effort Fetch also works with seek, in a slightly more complex manner.

In some cases, there may be legitimate gaps in the content and the design of BEF has factored this case in. For example, suppose there are 100 fragments missing because the encoder was turned off during this period. In these cases, OSMF will not make requests for all 100 fragments. BEF is designed to fall back to normal fetch behavior after a configurable number of sequential failures occur. The default number of sequential failures is two.

THE CONTROL PLANE

Best Effort Fetch works well for addressing minor disruptions on the server side. When deployed properly, temporary liveness and dropout issues on the server side will be invisible to players. However, BEF is not a silver bullet. In the event of major disruptions, such as a packager going down for a very long time, even BEF will begin to exhibit discrepancies in playback.

To address these cases, AMS provides the control plane module, a set of administrative interfaces that give you additional control and monitoring of your streams, so you can detect malfunctioning streams and disable them.

Enable the control plane module

To enable the control plane, add the following lines to your httpd.conf on each of your packagers:

```
LoadModule ctrlplane_module modules/mod_ctrlplane.so
<IfModule ctrlplane_module>
<Location /ctrlplane>
                SetHandler ctrlplane
                HdsHttpStreamingLiveEventPath "../applications"
                HlsHttpStreamingLiveEventPath "../applications"
                MaxBootstrapAge 300
                Options -Indexes FollowSymLinks
</Location>
</IfModule>
```

*SECURITY NOTE: For your production deployment, you should **strongly consider** adding access controls to protect access to the /ctrlplane URL to prevent unauthorized changes to your events.*

Write your admin service

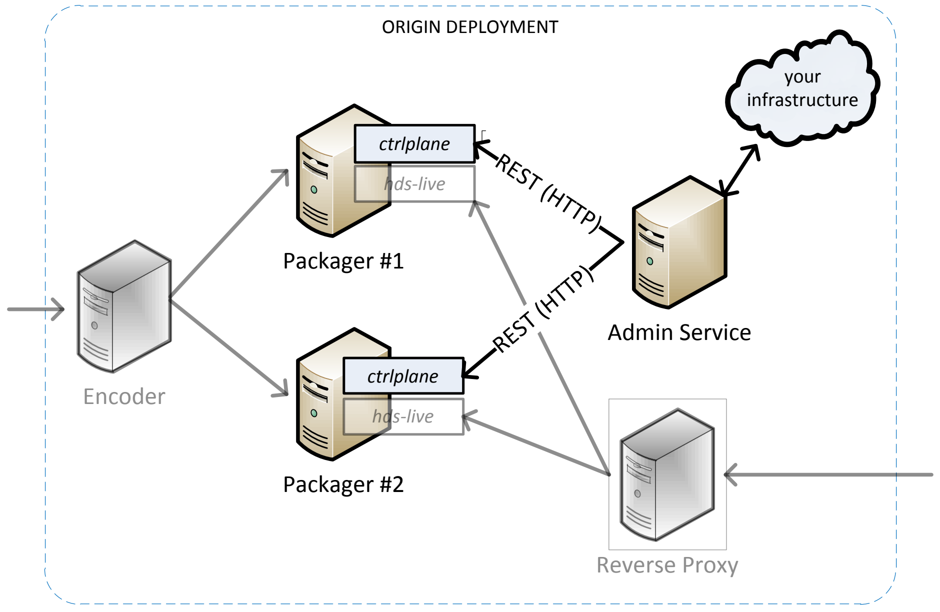

Figure 6. Admin service architecture.

The control plane module is designed to work in conjunction with an ***admin service.*** An admin service is a web service that you create and integrate with your existing infrastructure. An example admin service is included with the sample files for this whitepaper.

The admin service will query status and issue commands to the control planes on each packager in your system. The control plane exposes a simple RESTful interface, enabling you to code your admin service in the language and framework of your choice. Depending on your needs, you may choose to code your admin services as full blown PHP or Ruby on Rails web services, or you may choose to code your admin service as a simple cron job.

Regardless of the language you choose for your admin service, the admin service must fulfill certain responsibilities to maximize the fault tolerance of your HDS system.

Responsibility #1: Monitor event status

The first responsibility of your admin service is to monitor the health of your HDS events. To accomplish this, your admin service should periodically poll the control plane on each packager to get an updated status.

To query the health of an event, your admin service should issue an HTTP GET request to each control plane. The following is an example using curl:

```
# to query the status of the event with the url:
# http://packager1.example.com/hds-live/livepkgr/_definst_/liveevent.f4m
# we will issue a GET to
# http://packager1.example.com/ctrlplane/livepkgr/_definst_/liveevent.f4m/status

$ curl
http://packager1.example.com/ctrlplane/livepkgr/_definst_/liveevent.f4m/status

<?xml version="1.0" encoding="UTF-8"?>
<controlplane-event-status
xmlns="http://ns.adobe.com/hds/controlplane/status/1.0">
        <manifest-file>
                /livepkgr/_definst_/liveevent.f4m
        </manifest-file>
        <status>
                enabled
        </status>
        <done>
                true
        </done>
        <up>
                true
        </up>
        <bootstrap
                 name="livestream1"
        >
                <up>
                        true
                </up>
                <age>
                        9
                </age>
                <status>
                        enabled
                </status>
                <done>
                        true
                </done>
        </bootstrap>
</controlplane-event-status>
```

There are a few key pieces of information in the response:

<up> is the simplest way to determine whether an event is running normally. <up> is a simple yes or no answer to the question, "Is the stream working?" True means the stream is running normally.

<bootstrap> elements provide details about the individual streams of your event.

<age> indicates the number of seconds since the last update of the enclosing stream. If it is appropriate to your implementation, you may choose to only examine age and ignore the value of <up>. The control plane bases its value for <up> upon <age>.

Responsibility #2: Disable malfunctioning events

In your system, you must make sure that packagers do not continue serving malfunctioning events for a long period time. As indicated earlier, if a packager serves a malfunctioning event for a long period of time, this may result in playback discrepancies. It is the admin service's responsibility to disable malfunctioning events.

The control plane module provides an API that disables an individual event on a packager, while leaving other events on that packager intact. A disabled event will always return an HTTP 503 error to the proxy, which effectively skips over the packager in presence of 503 failover.

To disable a malfunctioning event, your admin service can issue an HTTP POST to the control plane of the packager on which the malfunctioning event resides; for example:

```
# to disable an event with the url:
# http://packager1.example.com/hds-live/livepkgr/_definst_/liveevent.f4m
# we will issue a POST to
# http://packager1.example.com/ctrlplane/livepkgr/_definst_/liveevent.f4m/disable

$ curl –d “”
http://packager1.example.com/ctrlplane/livepkgr/_definst_/liveevent.f4m/disable

<?xml version="1.0" encoding="UTF-8"?>
<controlplane-response xmlns="http://ns.adobe.com/hds/controlplane/status/1.0">
        <manifest-file>
                /livepkgr/_definst_/liveevent.f4m
        </manifest-file>
        <stream
                 name="livestream"
        >
                <status-code>
                        1
                </status-code>
                <status-desc>
                        Command executed successfully.
                </status-desc>
        </stream>
</controlplane-response>

# to re-enable we could issue a POST to
# http://packager1.example.com/ctrlplane/livepkgr/_definst_/liveevent.f4m/enable
```

Understand the implications of malfunctioning events

It is your responsibility to decide when to disable an event. When making this decision, you should factor in the consequences of leaving a malfunctioning event in circulation.

When a packager suffering from liveness is left in rotation, players could begin live playback starting in the past. Live playback always begins with the most recent fragment listed in the bootstrap. Therefore, if a player receives a stale bootstrap, it will initiate playback starting at the outdated live point advertised in the bootstrap. For example, if your packager lost its encoder connection an hour ago, a player could begin its live playback with content that is an hour old. Such a player will continue to play content, but that content will always be old content from an hour ago. In other words, the player will be stuck “back in time.”

When a packager suffering from a very long dropout is left in rotation, the DVR window may behave erratically on players (it may grow or shrink abruptly). A dropout can be considered to be “very long” if its duration is longer than the difference between the DVR window and the disk management window. In other words, if you’ve configured your disk management window properly, a very long dropout is usually on the scale of hours.

Possible implementations

While the best admin service implementation depends upon the specifics of your organization's requirements, there are a few typical implementations you may want to consider.

One typical admin service implementation automatically disables malfunctioning streams as soon as they are detected. The service will also notify system administrators of the problem. This implementation might be desirable if your system has sufficient redundancy and you expect to leave your system unattended for long periods of time. This workflow has the added benefit that it guarantees a minimal level of live latency.

Another possible admin service implementation notifies administrators after detecting a problem, but does not automatically react. This implementation gives the system administrators time to first examine the situation before deciding on the appropriate action to take. Administrators must decide whether the side effects of liveness and dropout are severe enough to warrant the added risk and effort involved in disabling a packager. This implementation might be desirable if you have limited redundancy and can tolerate some live latency.

Responsibility #3: Tell the control plane when your event is done

The final responsibility of the admin service is to notify the control plane when an event is completed. The packager does not have the ability to intrinsically distinguish between a stream going down because the event is complete and it going down due to an encoder failure. The control plane provides a way for you to tell the packager your event is actually completed. This enables you to prevent false positives in detection of downed streams, and also has the added benefit of preventing Best Effort Fetch from fetching unnecessary fragments beyond the end of a done live event.

To mark an event as completed, your admin service can issue an HTTP POST; for example:

```
# to mark an event with the following url as done:
# http://packager1.example.com/hds-live/livepkgr/_definst_/liveevent.f4m
# we will issue a POST to:
# http://packager1.example.com/ctrlplane/livepkgr/_definst_/liveevent.f4m/done

$ curl -d “”
http://packager1.example.com/ctrlplane/livepkgr/_definst_/liveevent.f4m/done

<?xml version="1.0" encoding="UTF-8"?>
<controlplane-response xmlns="http://ns.adobe.com/hds/controlplane/status/1.0">
        <manifest-file>
                /livepkgr/_definst_/liveevent.f4m
        </manifest-file>
        <stream
                 name="livestream"
        >
                <status-code>
                        1
                </status-code>
                <status-desc>
                        Command executed successfully.
                </status-desc>
        </stream>
```

```
</controlplane-response>

# to mark the event as in progress (not done) we could issue a POST to
# http://packager1.example.com/.../liveevent.f4m/inProgress
```

Best practices for restarting a downed packager

In the event of a major failure, you may need to take a packager completely offline, and then bring it back online several hours later. This section outlines a few best practices to consider when performing this task.

If the streams are live only (that is, no DVR is involved):

1. Start the packager with the offending streams disabled. The streams will be producing content, but will not be accessible by the proxy.
2. Wait several seconds. You should wait for a time equal to the live offset plus the client buffer duration. This enables the packager to have enough content available to allow a client to fill its buffer without stalling.
3. Re-enable the streams on the packager.

If the streams are DVR-enabled, the process is more complex:

1. First determine if the outage occurred at the start of your event or if the outage was longer than the disk management window minus the DVR window.

   In either of these situations, you could experience playback issues once the dropout overlaps the start of the DVR window. To avoid playback discrepancies, copy the contents of a good packager to the malfunctioning packager. Note that certain files (such as .stream files) contain machine-specific path names. If paths differ between your packagers, be certain to adjust these files accordingly.
2. Start the packager with the offending streams disabled. The streams will be producing content, but will not be accessible by the proxy.
3. Wait for some time before re-enabling the offending stream. The minimum amount of time you should wait is the live offset plus the client buffer duration. The safest amount of time to wait is an entire DVR window. The amount of time you wait directly corresponds to the amount of DVR content that is available to clients in a worst case scenario.

MULTIPLE REVERSE PROXIES

In the process of adding redundancy to a packaging layer, the proxy has introduced a potential single point of failure. To safeguard against a proxy failure, you should consider adding a second proxy to your system.

A typical architecture fully connects all proxies to all packagers and exposes each proxy as a different origin to your CDN:

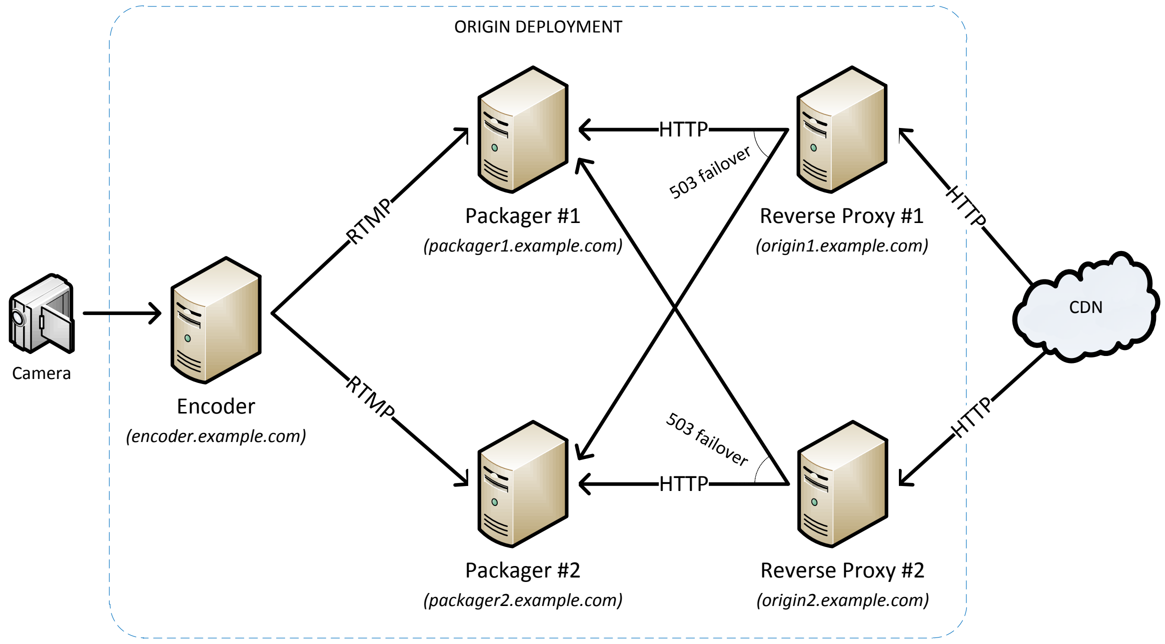

Figure 7. A typical multi-proxy deployment.

The best architecture for your deployment depends on the specifics of your CDN. As a result, it's a good idea to check with your CDN for additional guidance before settling on a particular proxy configuration.

MULTIPLE ENCODERS

Now that you have added redundancy at the proxy and packager layers, you can focus on adding redundancy at the encoder layer.

When deploying multiple encoders, there are two main approaches: a ***unique-event architecture*** and a ***redundant-event architecture***. The unique-event architecture is the preferred choice between the two, but is also comes with additional content and encoder requirements.

Unique-event architecture

In a unique-event architecture, each video feed corresponds to a single HDS event, regardless of the number of encoders present.

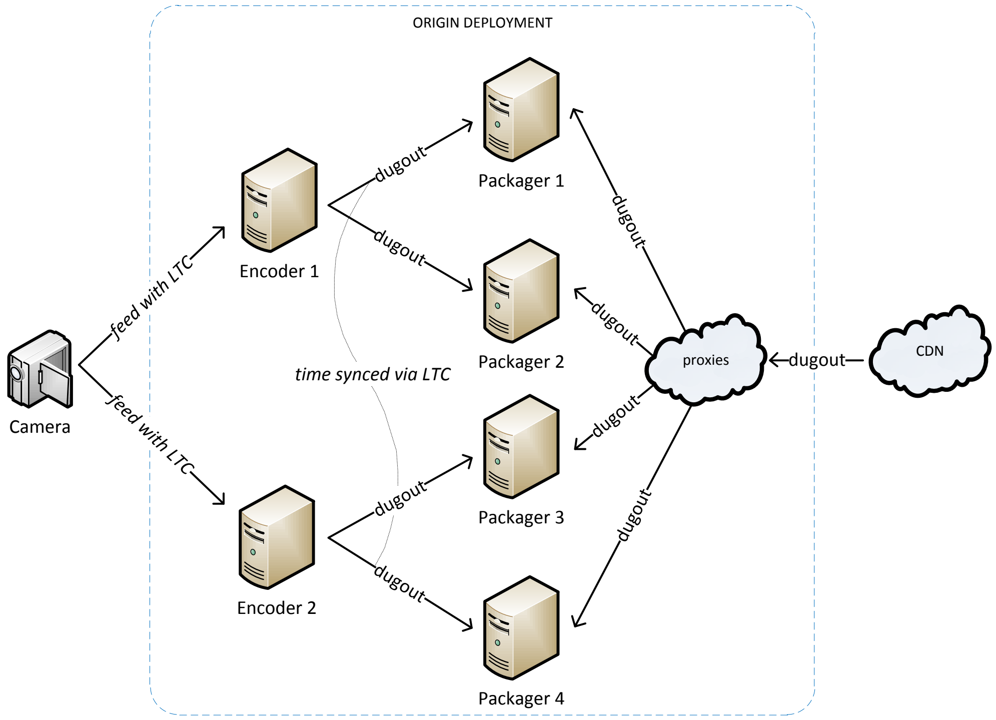

Figure 8. An example unique-event deployment.

For example, suppose you are streaming a baseball game with dual encoders, and one of your cameras is always focused on the dugout. In a unique-event deployment, you will have a single event that corresponds to the dugout video feed. All of the encoders in your system will encode the dugout video feed and then publish the stream to multiple packagers. They will always publish against the same event name: “dugout”. As a result, when the client wants to view the dugout camera angle, it will always play the “dugout” event.

NOTE: In the pictured deployment, there are four packagers. This configuration ensures that streaming will succeed even if an encoder and a packager fail simultaneously. If your risk tolerance is higher, it's also possible to deploy 2 packagers in total by pairing each packager to an encoder.

As mentioned earlier, the unique-event architecture is recommended over the redundant-event architecture, but it may not work for everyone. The unique-event architecture will only work if your content and encoders meet certain requirements.

Requirement #1: Encoders must produce identical output

All encoders that process a piece of content must encode that content in exactly the same way. Identical output guarantees that all the fragments for a given stream are identical, regardless of the original encoder that processed them.

When put into practice, this (almost always) means that:

> **All encoders must be the same model and use identical settings for key frame interval, bitrate, and other relevant settings.**

Requirement #2: Encoders must produce identical timestamps

All encoders must also produce consistently identical timestamps, such that the timestamp for a given frame of video or audio is always exactly the same value, regardless of which encoder processed the content.

When put into practice, this (almost always) means two things:

> **Content must have linear time codes (LTC) associated with it.**
>
> **All encoders must source their RTMP timestamps from the LTC.**

Since the details vary between encoders, make sure to consult your encoder documentation to verify that the ability to source RTMP timestamps from LTC is available.

Redundant-event architecture

Sometimes your content might not contain embedded LTC, or you may not be able to match encoder models. In these cases, you can still achieve encoder redundancy by using the redundant-event architecture.

In the redundant-event architecture, each video feed corresponds to multiple HDS events, one for each encoder present in the system.

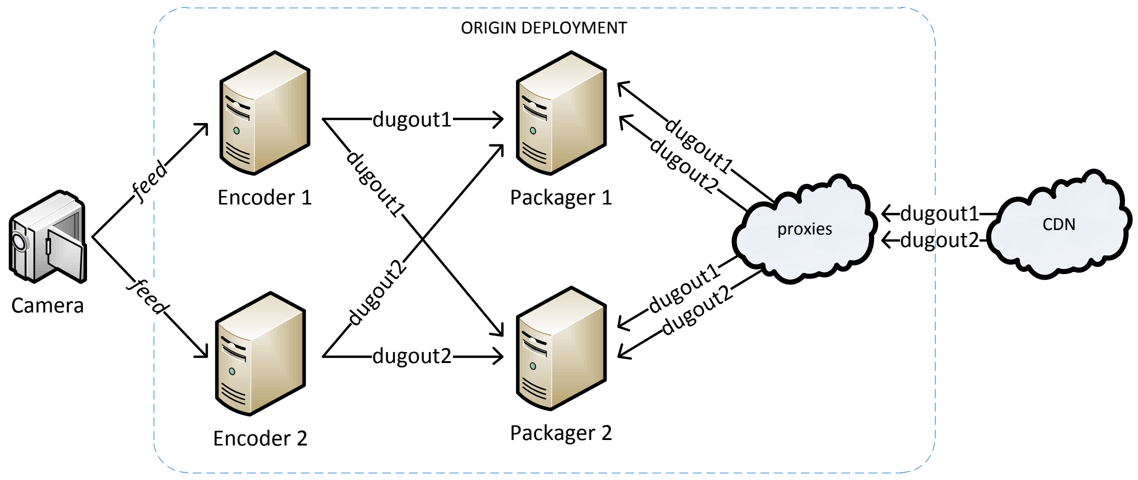

Figure 9. An example redundant-event deployment.

For example, suppose you are again streaming a baseball game with dual encoders. In a redundant-event deployment, you will have two events corresponding to the dugout video feed. Encoder 1 will receive the dugout feed and publish it to both packagers as "dugout1". Encoder 2 will receive the same dugout feed and publish it to both packagers as "dugout2". As a result, when the client wants to view the dugout camera angle, it must choose between using either "dugout1" or "dugout2".

To choose between the two events, the client must have some knowledge of which encoders are "up." One way to achieve this is to have the player call into a custom web service that will respond with the health of the two encoders, and switch based upon the result.

Another possibility is to make use of the OSMF PlayEvent.LIVE_STALL and PlayEvent.LIVE_RESUME events. PlayEvent.LIVE_STALL fires when stalls due to liveness are detected on the client side. When playback resumes ,PlayEvent.LIVE_RESUME fires. In response to these events, your client can switch to a different event. Further details can be found in the OSMF wiki.

Comparing encoder architectures

The following table compares the two architectures:

	Unique-event	**Redundant-event**
How are encoder failures handled?	Encoder failure is an instance of liveness. When an encoder fails, the situation is identical to the one in which some packagers are suffering from liveness, but others remain healthy. Since your multi-packager deployment already includes 503 failover and Best Effort Fetch, you are already well equipped to deal with liveness and dropout and you do not need anything new in order to handle encoder failure.	Encoder failure is detected and handled on the client side. Clients must detect whether a stream is "up" or not, and add custom client-side logic to switch to a backup stream.
Does the client need to know about the server-side architecture?	No, clients are abstracted from the server-side architecture. Clients do not know or care how many encoders are present on the server side. In production, you may add and remove redundant encoders as necessary without client-side changes.	Yes, clients must implicitly be aware of the server-side architecture by knowing how many redundant events are available for a given feed.
What are the content requirements?	Content must have embedded LTC.	Content does not require LTC.
What are the encoder requirements?	Content timestamps must be sourced from embedded LTC. Redundant encoders must be identical.	Encoders may be different. Sourcing timestamps from LTC is not required, but will make seeks and failover more accurate.

IMPROVING THE HLS EXPERIENCE

The majority of the discussion thus far has covered HDS failover. This section focuses on the other side of streaming, HLS.

The good news is that once you have a fault-tolerant HDS deployment, enabling a fault-tolerant HLS experience is easy. HLS will benefit from the same multi-proxy, multi-packager, and multi-encoder enhancements you made for HDS. There only a few specific areas you'll need to revisit.

Configure Cache-Control

As with HDS, you must always configure the HTTP Cache-Control: max-age to gain maximum scalability. These parameters can be tuned via the HLSM3U8MaxAge and HLSTSSegmentMaxAge parameters, which control the M3U8 and TS file lifetimes respectively.

- TS files are the HLS analog of fragments. They do not change frequently, so may be configured with a long max-age.
- M3U8 files are the HLS analog of the bootstrap, and should be configured with a short max-age. A good value is typically half of a fragment interval.

You can achieve this by altering httpd.conf as follows:

```
<Location /hls-live>
    HLSM3U8MaxAge 2
    HLSTSSegmentMaxAge 86400
</Location>
```

Enable URL hashing for .m3u8 files

Although the M3U8 is the HLS equivalent of the bootstrap, HLS requires that the m3u8 file must only grow in size. An M3U8 can only have entries appended to its tail in subsequent refreshes. When a reverse proxy is present in front of multiple packagers, natural latency between the packagers can cause minor fluctuations between subsequent m3u8 requests. As a result, it is highly recommended that you enable the URL hashing scheme to redirect m3u8 requests. URL hashing assigns a URL an affinity for a particular packager. This causes the m3u8 to consistently be served by the same packager, ensuring that the m3u8 will only grow between subsequent refreshes.

If you are using Varnish as your reverse proxy, you can achieve this by altering your default.vcl:

```
director hls_hash hash {
        {.backend = b1; .weight = 1;}
        {.backend = b2; .weight = 1;}
}

sub vcl_recv {
        if(req.url ~ "(?i)^/hls-live/.*\.m3u8$") {
                # for HLS, use hashing due to M3U8 append requirement
                set req.backend = hls_hash;
        } else {
                # for most requests we will perform round robin
                set req.backend = hds_round_robin;
        }
}
```

Liveness and Dropout

Best Effort Fetch is a great solution for HDS client side improvements, but it is not a good option for addressing liveness and dropout issues when using HLS. HLS differs from HDS in that a client-side solution isn't viable. To work around this limitation, AMS provides a server-side solution to HLS liveness and dropout problems.

The idea behind the HLS solution is simple. The HLS Apache module has been modified to detect periods of dropout and generate TS entries during those periods, even if those TS entries do not correspond to the fragments on the current packager. In a similar manner, entries to accommodate for liveness are generated based upon the age of the existing bootstrap file. The end result of these modifications are that the player will make a series of fetches for content that may not be present on the packager that generated the M3U8, but may be present elsewhere in the system. In practice, this causes the HLS player to perform a series of requests that are analogous to those performed with HDS Best Effort Fetch.

Enabling HLS liveness and dropout protection is straightforward. When the HLS module detects the presence of the bestEffortFetchInfo tag in the manifest.xml for a stream, HLS "best effort fetch" is automatically enabled. In other words, once BEF is enabled for OSMF, it's also enabled for HLS.

CONCLUSION

If you've followed all the steps outlined in this whitepaper, you've been on quite a journey.

You started working with a fragile and failure-prone deployment, one that had everyone hoping that nothing would go wrong during crucial moments.

You transformed this deployment, making use of advanced features such as Best Effort Fetch and the control plane and adding redundancy at all levels.

What is the reward for all this hard work? The answer is simple: You can be confident in your system. You know that it will scale and you know that it will withstand failures. You no longer need to merely hope it will work during crucial moments, because you have built a rock-solid deployment that instills confidence.

ONLINE RESOURCES

HTTP Dynamic Streaming and HTTP Live Streaming with Adobe Media Server:

http://help.adobe.com/en_US/adobemediaserver/devguide/index.html

Open Source Media Framework:

http://www.osmf.org/

Tutorial on the Varnish proxy:

https://www.varnish-software.com/static/book/

Internet Caching Tutorial:

http://www.mnot.net/cache_docs/

ABOUT THE AUTHOR

Glenn Eguchi is a Senior Computer Scientist in the Adobe Video Solutions group. When he is not thinking about failover, Glenn likes to spend his time playing guitar or cooking for his wife.