CloudFront Design Patterns And Best Practices

Abhishek Tiwari

Citation: *A. Tiwari*, "CloudFront Design Patterns And Best Practices", Abhishek Tiwari, 2013. doi:10.59350/bksfp-rkm49

Published on: January 31, 2013

CloudFront is a content delivery service offered by Amazon web services(AWS). CloudFront serves static contents (images, audio, video etc) using a global network of more than 28+ edge locations. Using these edge locations, CloudFront accelerates delivery of content by serving the cached copies of the content objects from a nearest edge location. Service is highly reliable and designed to be used with Amazon S3 or any other custom origin server.

CloudFront can also serve the dynamic or interactive content but current support is quite limited compared to similar services such as Akamai. Given that CloudFront is more cost effective than Akamai [1] and service is evolving very rapidly in terms of features, I would highly recommend the CloudFront.

There are some interesting uses patterns and best practices developed around the CloudFront. Before we deep dive into CloudFront design patterns and best practices it will be nice to have some familiarity with the basic CloudFront concepts.

Cloudfront Distribution

A CloudFront (CF) distribution basically identifies, distributes or caches the content objects to edge locations. A CloudFront distribution consists a collection of origin servers and rules to define the origin or cache behavior. An origin server is a original source of content, it can be a static file store like Amazon S3 or a dynamic content server. Currently CloudFront supports only 10 cache behaviors for each CloudFront distribution. Note that each cache behavior is unique and it can be only associated with only one origin server. Ability to associate one cache behavior to multiple origin server is currently not available. Off course you can associate one origin server to multiple cache behaviors. CloudFront distribution also supports the versioning of the content objects using query strings.

[1] Truth be told, Amazon CloudFront has a more open pricing model compared to Akamai.

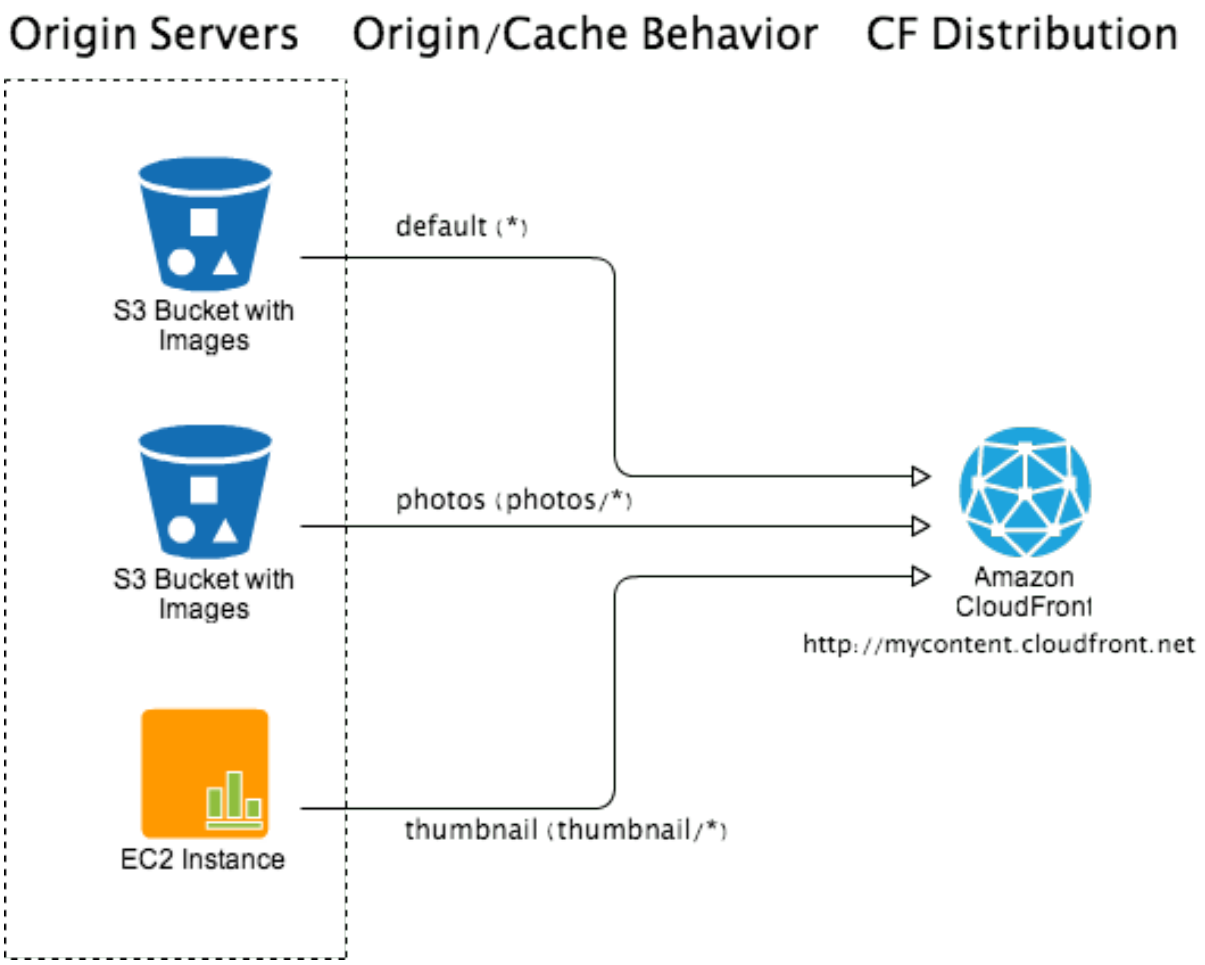

Figure 1: CloudFront-Origin and Cache Behavior

Depending on content, CloudFront distribution can be download or streaming distribution. Streaming distribution is normally used for on-demand video/audio streaming. Download distribution is used to serve other forms of non-streaming contents like images, CSS, JavaScripts, HTML pages etc. Video/Audio content can also be served by download distribution assuming a media player plugin is active in the browser. Both download and streaming distributions support HTTP/HTTPS protocol. Depending on media server streaming distribution may support additional protocols.

How Cloudfront Works?

When a file is requested to a CloudFront distribution,

1. CloudFront first determines nearest CloudFront edge location.

2. Then CloudFront checks cache of the nearest edge location for the requested file. If the file is in the cache, edge location will serve it.
3. If the files is not in the cache, CloudFront will match the request pattern with origin or cache behavior and obtain the file from corresponding origin server and distribute it to edge locations.

Pattern-1: Domain Sharding

All web browsers are capable of downloading resources such as CSS, JavaScripts, images etc in parallel. Traditionally, web browsers limit the number of simultaneous connections a browser can make to one domain [2][3]. For instance, Internet Explorer 7 only allows two parallel downloads per server.

Most of the modern browsers have removed the restriction of just two parallel downloads per server. In fact, Firefox desktop can make eight connections per server, and for Google Chrome and Internet Explorer 8/9 the limit is now six connections per server.

Still there is lot of opportunity to improve. For instance, a normal e-commerce web page downloads more than 80+ resources during a single page view - eventually slowing down the page load time. This is where domain sharding technique comes very handy.

Domain sharding uses two or more hostnames or CNAME aliases to serve the content from a single server. Browser treat them as different servers and hence we see increase in limit on parallel downloads. For instance, sharding across 2 CNAME aliases will double the parallel download capabilities. On average, using two CNAME aliases is best. More than four CNAME aliases will degrade browser performance resulting in high CPU and memory uses[4].

Using CluouFront CNAME alias feature you can map multiple CNAMEs on a given CloudFront distribution (upto 10 CNAMEs). For CNAME aliasing we can use Amazon Route 53. On application side, you have to enable domain sharding logic which basically requires filtering all your static/media URL references in your view/template and randomly mapping to one of the CNAME alias. For instance, in Django you can write a Django middleware or template processor which will replace any `MEDIA_URL` or `STATIC_URL` element in template to one of the CNAME alias.

[2] Why Domain Sharding is Bad News for Mobile Performance and Users
[3] Sharding Dominant Domains
[4] Performance Research, Part 4: Maximizing Parallel Downloads in the Carpool Lane

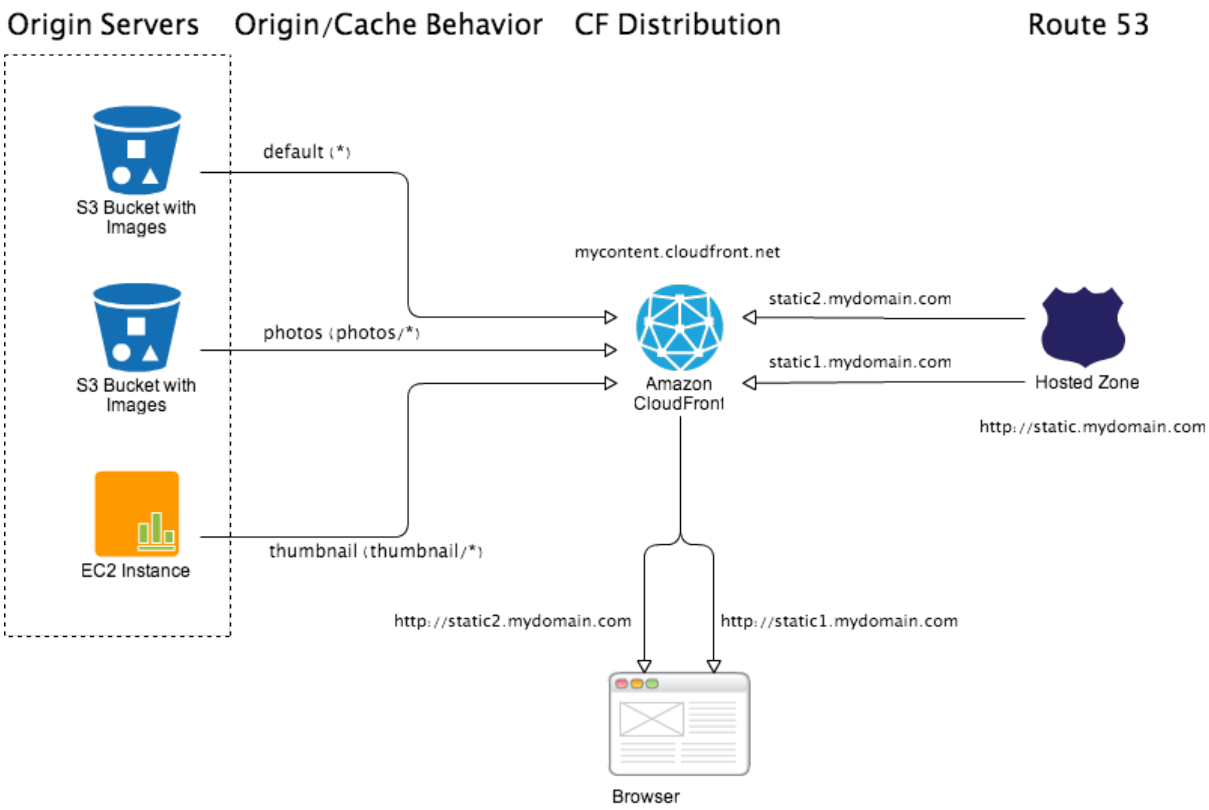

Figure 2: Domain Sharding with CloudFront

Please note, currently domain sharding is not possible for HTTPS requests due to SSL certificate mismatch issues. To bypass this limitation you can use two different CloudFront distributions with same origin servers and rules.

Pattern-2: Versioning

In CloudFront cache invalidation is a costly operation with various restrictions. First of all, you can run only 3 invalidation requests at any given time. Second, in each validation request you can included maximum of 1000 files. Third, invalidation takes time propagate across all edge locations (5~10 minutes). The change in CloudFront distribution will eventually be consistent but not immediately. In terms of cost, in a given month invalidation of 1000 files is free after that you have to pay per file for each file listed in your invalidation requests.

Content versioning is one of the best way to avoid the invalidation related issues. CloudFront supports versioning using query strings. To enable query string based versioning, you have to turn on “Forward Query Strings” for a given cache behavior. After that CloudFront will pass the full object path (including the query string) to the origin server. CloudFront will use full object path to uniquely identify the

content object in cache. So for invalidation of versioned object you have to provide full object path like `/static/profile.png?versionID=123`. On origin server side, server will interpret the query string and serve the definitive version of the content object.

It is suggested that when "Forward Query Strings" is turned off CloudFront caching performance is better. If query based versioning don't work for you then you can use version based file naming without compromising the CloudFront caching performance. For versioning you can use one or combination of following approaches,

1. File name plus query string with version `/static/profile.png?versionID=123`.
2. File name with version prefix/suffix `/static/profile_v123.png`.
3. File name as unique key based on file content `712vds57tr18929812312enb.png`.

File Name Plus Query String With Version This works quite well for all content types including CSS, JavaScript, images, videos etc. For application content types such as CSS/JavaScript served from a custom origin such as application server you can use hash of Git/Hg `HEAD` of application code as version-id. For media content types such as images/videos stored in Amazon S3 you can use `x-amz-version-id` header as version-id.

For example in following illustration, all application servers are running same version of application code. `468df6b` is first 7 chars of hash of Git `HEAD` of application code. Off course your view/template will need to extract this version-id from Git `HEAD` and use when referencing to a CloudFront content URL. Also, When deploying new version of application code ensure that application server acting as custom origin for CloudFront is always updated first [5]. Moreover when storing reference to CloudFront objects, for versioned objects you have to store the full object path with version string in your application database.

[5] If application server acting as custom origin is not updated first, it can not serve file with appropriate version and hence chances are it will serve and old file.

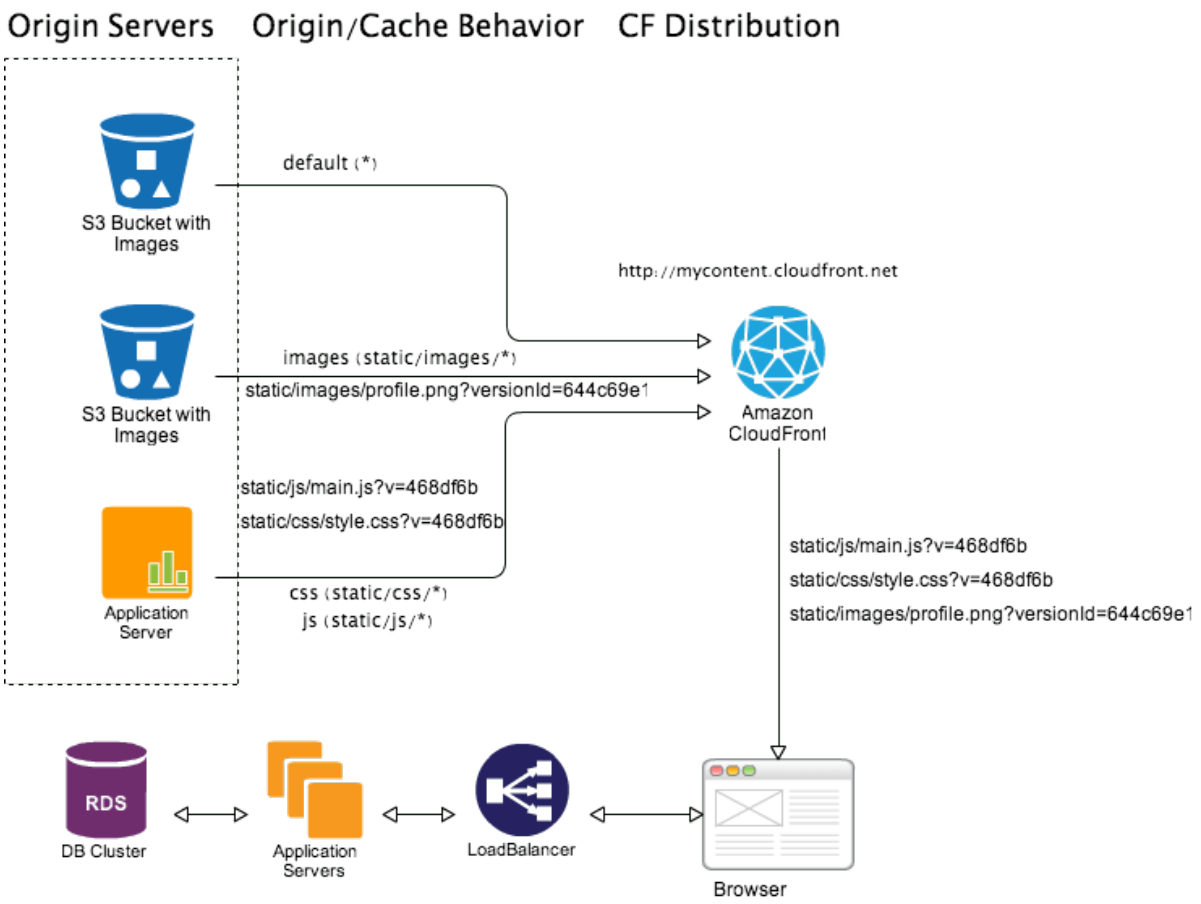

Figure 3: Object versioning with CloudFront using query string

File Name With Version Prefix/Suffix This works pretty much same as first approach but without query string and with improved caching. This is more suitable for the application content types(CSS/JavaScript). New object paths looks like,

1. `static/js/main_v468df6b.js`
2. `static/css/style_v468df6b.css`

File Name As Unique Key Based On File Content In this approach file name is an unique hash key generated from file content. This is suitable for both application and media content types.

For images/videos content, before uploading to Amazon S3 you can generate MD5 hash for a given file and use it as file name for uploading the content. Using this approach you can also avoid the content duplication as it is possible to compare the contents using MD5 hashes.

For application content, you can use a pipeline which will minify and combine CSS/JavaScript files and then generate a single file for CSS/JavaScript with file name generated using MD5 hash of file content.

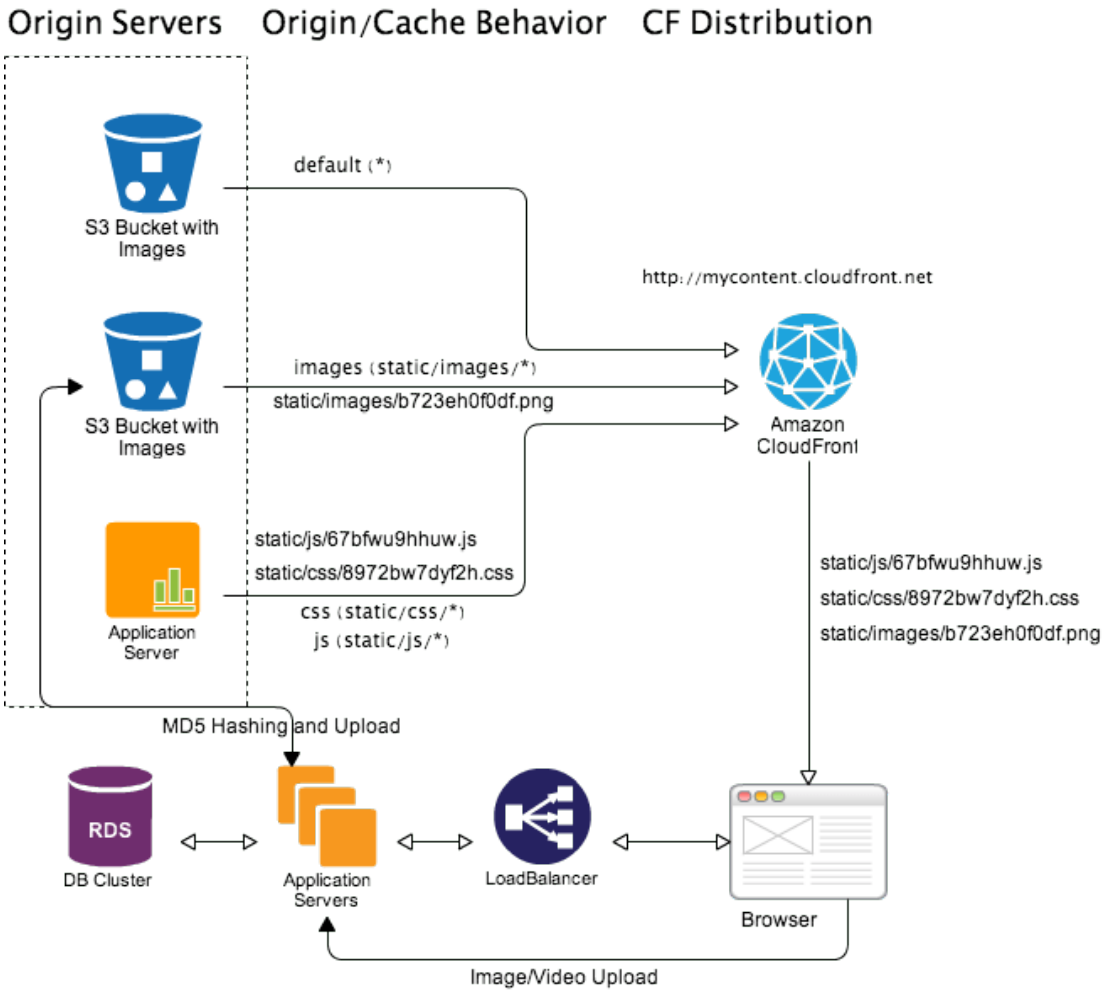

Figure 4: Content based hashing and versioning for CloudFront

Pattern-3: Dynamic Thumbnailing And Encoding

Using custom origin it is possible to do dynamic image thumbnailing and video encoding. An example workflow depicted in following diagram.

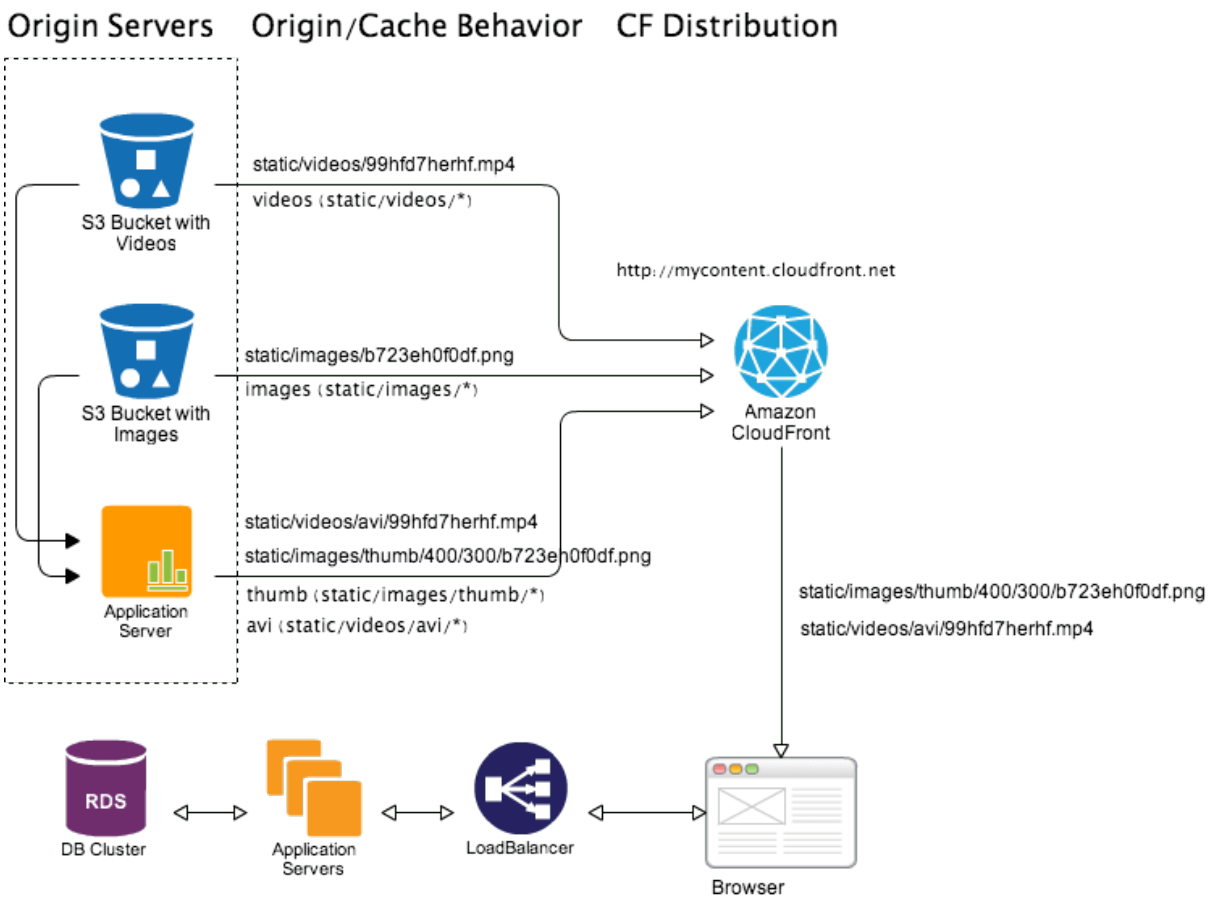

Figure 5: Dynamic Thumbnailing and Encoding for CloudFront

Let say user's browser requested file `static/images/thumb/400/300/b723eh0f0df.png` which is basically thumbnail of size `400x300` for image `static/images/b723eh0f0df.png` stored in a S3 bucket. Now if this image is already in CloudFront cache then CloudFront will serve cached thumbnail. Otherwise CloudFront will pass the request to origin server with matching cache behavior (`static/images/thumb/*`). In this case application server origin will take the call, download the original image `static/images/b723eh0f0df.png` from S3 bucket and generate thumbnail of requested size and return it to CloudFront distribution to serve. Now depending on size of original image and quality of thumbnail this whole process can take 50-500ms. Once generated we can set longer expiration time for the thumbnail image so that in near-future CloudFront don't request again. As the original image is versioned the thumbnail is also version controlled.

One may argue to pre-generate thumbnails in bulk for pre-defined sizes and store in Amazon S3 but given that now a days content is consumed by devices with variable screen size and resolution, practically it will be impossible to generate thumbnails in bulk. Not to mention a small change in thumbnail size will require bulk re-generation of thumbnails. Dynamic thumbnailing can generate thumbnails

optimized for the device on the fly and cache it on CloudFront.

Same kind of workflow can be used for dynamic video/audio encoding. Only difference will be the required time to complete the round trip which will be longer than image thumbnailing. Unfortunately dynamic encoding may be practical for very small sized files only. Using Amazon Elastic Transcoder and Amazon SQS with a custom origin server we may improve overall performance but still it is not as efficient as batch encoding.

Pattern-4: Compression

Amazon CloudFront can serve both compressed and uncompressed version of files depending on viewer or browser request. To receive compressed content browser must include `Accept-Encoding: gzip` in the request header. If compressed content is available then it will be served otherwise CloudFront will serve the uncompressed version of the file.

Compressed content is served faster and uses less bandwidth. Unfortunately for compression CloudFront relies on the origin servers. Amazon S3 does not perform the compression as well. Although Amazon S3 can store both gzip and non-gzip versions of the file in the same bucket. When uploading gzip file to Amazon S3 we need to set `Content-Encoding` to `gzip`. It is recommended to use a custom origin server to compress the content on-the-fly. Most of servers like Nginx, Apache etc support on-the-fly gzip compression.

If we are using custom origin server to serve the versioned content (HTML, CSS, and JavaScript) it can perform compression at same time. Generally media files (imgaes/videos) are already compressed so there is not much scope for additional compression. For media files additional compression can be achieved by reducing the quality of content during thumbnailing or encoding.

Pattern-5: Audio/Video Streaming

On-demand streaming of audio/video content is quite easy using CloudFront. Using CloudFront signed URLs we can also support paid on-demand streaming.

Audio/Video streaming using CloudFront requires two CloudFront distributions - one regular download distribution (HTTP protocol) for the media player and one streaming distribution (RMTP protocol)[6].

> When you configure CloudFront to distribute media files, CloudFront uses Adobe Flash Media

[6] Working with Streaming Distributions

> Server 3.5 as the streaming server and streams your media files using Adobe's Real-Time Messaging Protocol (RTMP). CloudFront accepts RTMP requests over port 1935 and port 80.

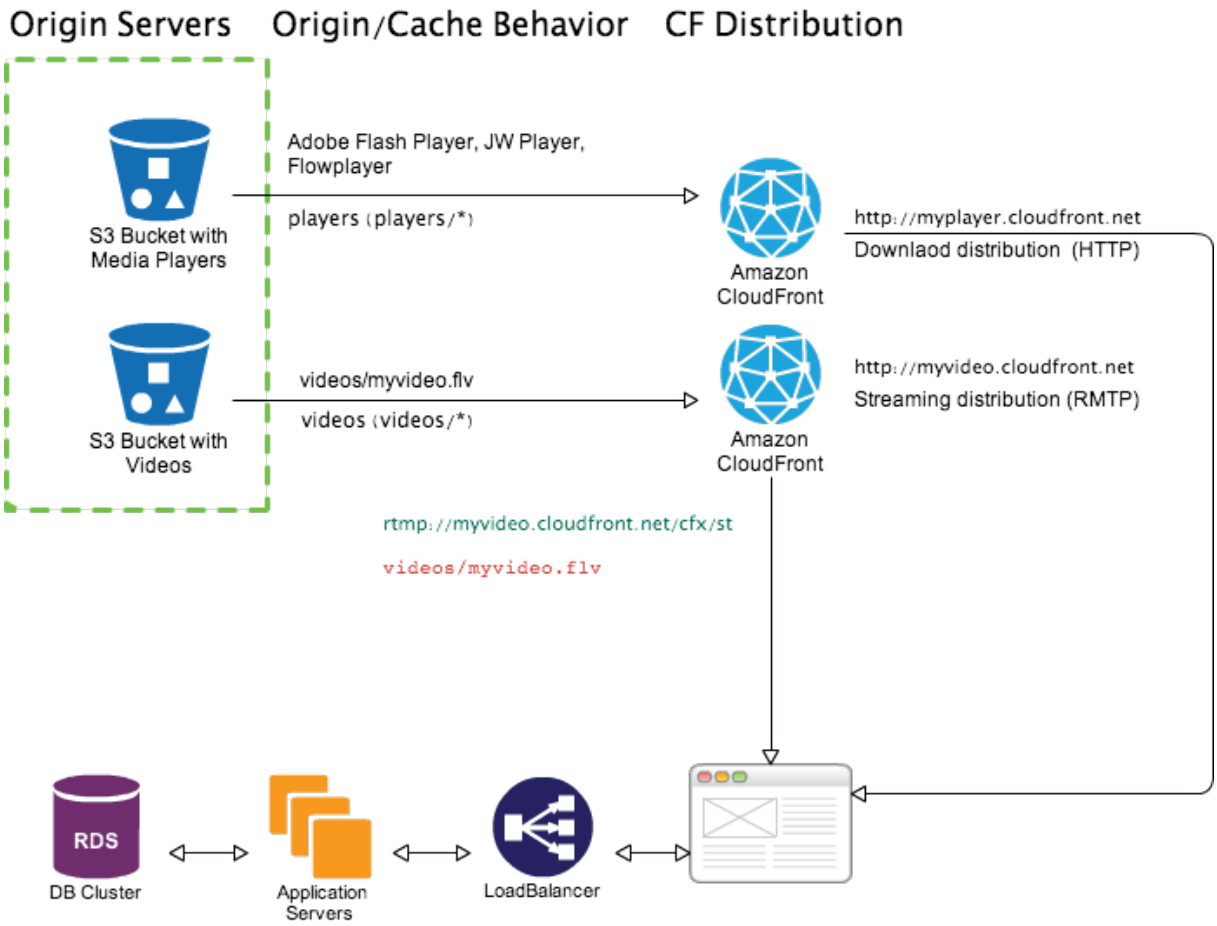

Figure 6: On-demand audio/video streaming from CloudFront

To configure the media player with correct path to the file we have to provide `streamer` URL plus `file` path. Streamer URL is base CloudFront URL plus `cfx/st`. For `rmtp://myvideo.cloudfront.net/cfx/st/videos/myvideo.flv`

- streamer : `rmtp://myvideo.cloudfront.net/cfx/st`
- file : `videos/myvideo.flv`
- stream URL : `rmtp://myvideo.cloudfront.net/cfx/st/videos/myvideo.flv`

Please note for publicly available streaming content we are using Amazon S3 URLs and not CloudFront signed URLs.

Pattern-7: Private Content

CloudFront can restrict the access to private content provided by both download and streaming distributions. To access the private content end user requires special signed CloudFront URLs which can be generated either manually or automatically.

Signed URL[7] A signed CloudFront URL restrict access to CloudFront objects. It controls access based on several parameter such as expiration data and time, valid after data and time, IP address etc included in signed URL.

When generating signed CloudFron URL for a download distribution content object, you must use the full CloudFront URL as the resource. However for streaming distribution content object we only use the object path of the media file, other details come from the streamer URL.

A complete singed URL contains a base URL, policy statement, signature and CloudFront key-pair id and optional parameters such as expiration date and time. When CloudFront receives the request for a content object it matches the signed URL pattern. If signed URL request is valid then it gives end-user access to content object. For download distribution object CloudFront also compares expiration date and time with date and time of the HTTP request. For streaming distribution object it compares expiration date and time with date and time of the play event.

Signed URLs can be created either using a custom policy[8] or a canned policy[9]. A Signed URL created using Canned policy can provide access to only one file. A Signed URL created using custom policy enables access to one or more files using patterns matching or wild cards.

[7] Overview of Signed URLs
[8] Creating a Signed URL Using a Custom Policy
[9] Creating a Signed URL Using a Canned Policy

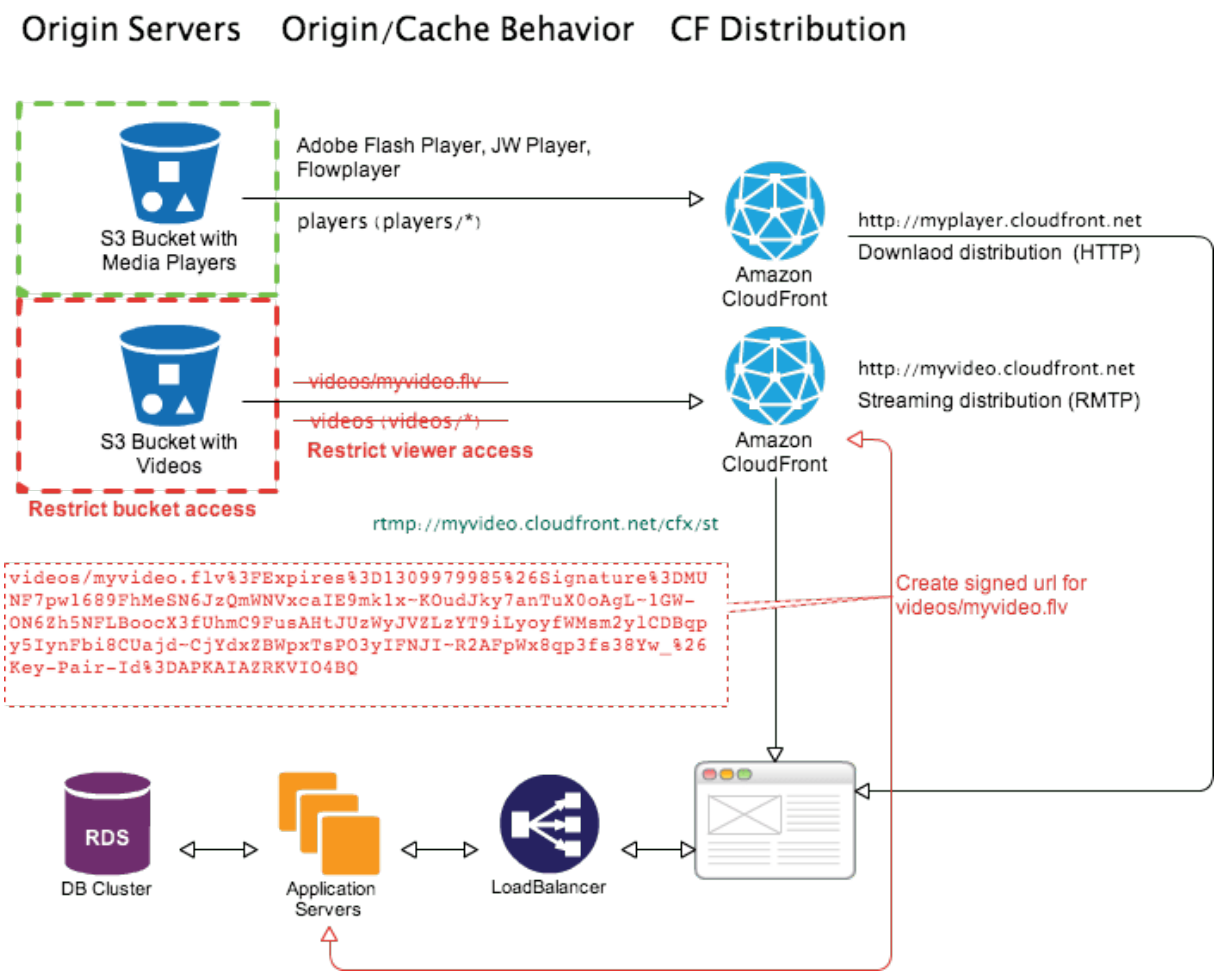

Figure 7: On-demand audio/video streaming of private content from CloudFront

Use Case Let say you are streaming a movie (illustrated above). To access the content for 24 hrs end-user requires to pay for streaming. To protect your video content you have restricted the bucket access and viewer access. This prevents anyone from bypassing CloudFront signed URLs. Once end-user has completed payment steps your application server creates CloudFront signed URL for the video content and redirects end-user to download and load the media player in the browser which is pre-configured to stream video content using signed URL. Now this video content is accessible for 24 hrs. After 24 hrs signed URL will be invalid and end-user can not play the video. Off course in 24 hrs period they can play the content as many times he or she want.

A similar workflow can be created for paid content download as described below.

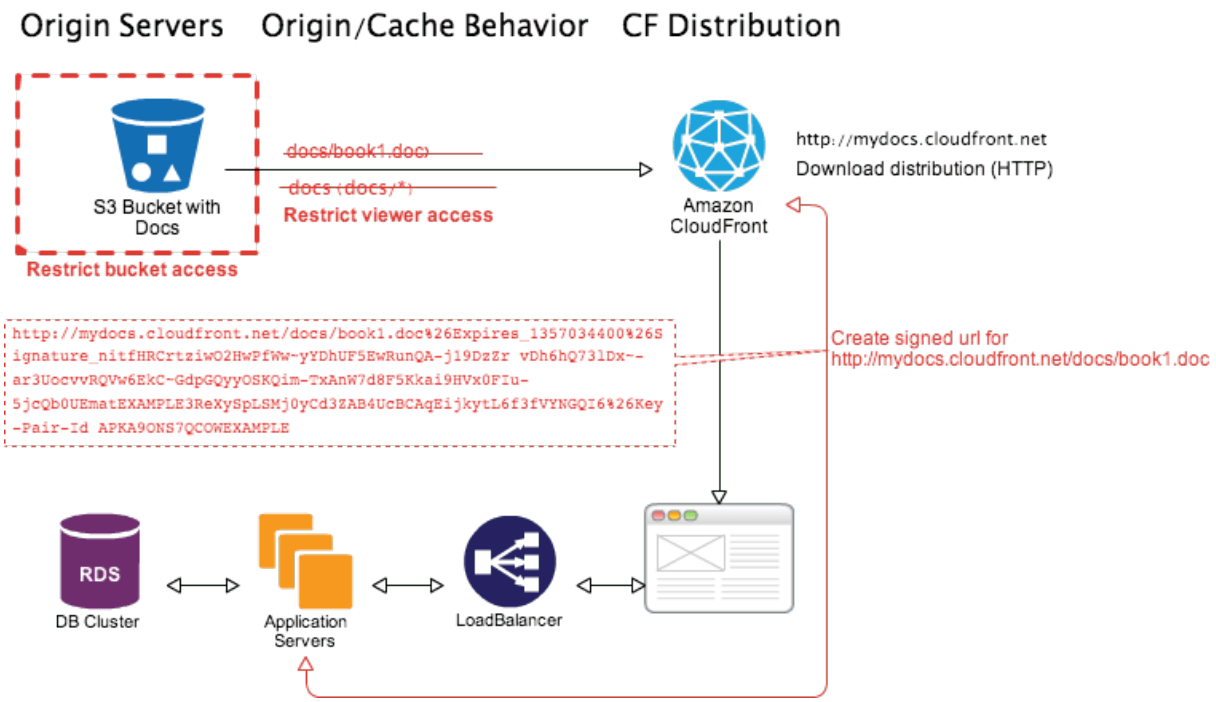

Figure 8: Download of private content from CloudFront

Pattern-8: Live Streaming

Currently CloudFront supports live streaming with Adobe's Flash Media Server 4.5[10] and IIS Media Services (Smooth Streaming)[11]. Both solutions live stream media over HTTP/HTTPS to Apple iOS devices by streaming in Apple's HTTP Live Streaming (HLS) format. In addition Adobe's Flash Media Server 4.5 can stream on demand content to Adobe Flash clients with Adobe's Real-Time Messaging Protocol (RTMP). CloudFront accepts RTMP requests over port 1935 and port 80. CloudFront supports the following variants of the RTMP protocol:

- RTMP— Real-Time Message Protocol
- RTMPT—RMTP Tunneled (over HTTP/HTTPS)
- RTMPE—RMTP Encrypted
- RTMPTE—RMTP Tunneled Encrypted

Similarly Smooth Streaming - an extension of IIS Media Services can live stream to Microsoft Silverlight clients over HTTP/HTTPS.

[10] Live HTTP Streaming Using CloudFront and Adobe Flash Media Server 4.5
[11] Live Smooth Streaming Using Amazon CloudFront and IIS Media Services 4.1

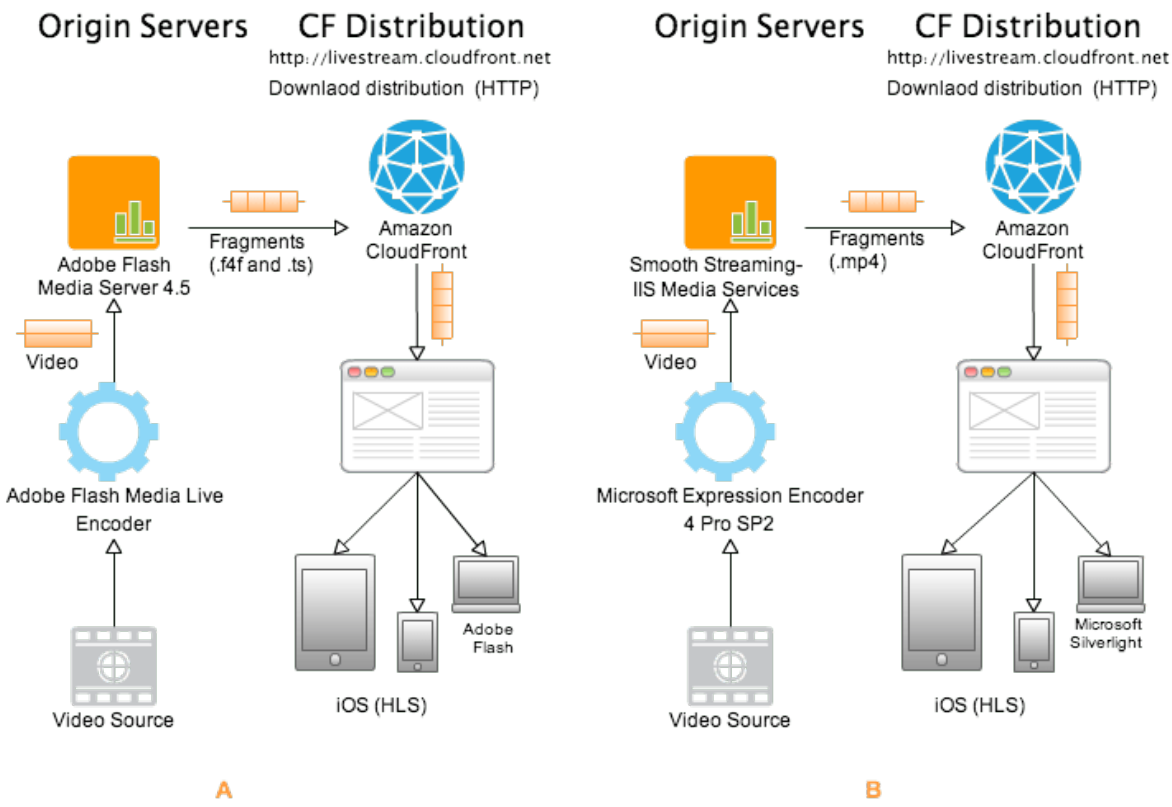

Figure 9: Live Streaming and CloudFront

How it works? Each solution relies on an encoder and a media service or server as described in above image (A: Adobe's Flash Media Server 4.5[12] and B: IIS Media Services). Encoder takes live video as input and convert video into right format. Then video is pushed to origin server (media service or server). Origin server then breaks the video into a series of smaller files (called segments or fragments) that are cached in the CloudFront network. It all happens in real time. Now each fragment can be encoded on different bit rate and depending on client's network conditions an appropriate fragments steam with optimal bit rate is served from CloudFront.

[12] Live HTTP Streaming Using CloudFront and Adobe Flash Media Server 4.5