

Network Working Group
Internet-Draft
Updates: 8006, 8008 (if approved)
Intended status: Standards Track
Expires: 8 September 2022

G. Goldstein
W. Power
Lumen Technologies
G. Bichot
Broadpeak
A. Sioniz
Telefonica
7 March 2022

CDNI Metadata Model Extensions
draft-goldstein-cdni-metadata-model-extensions-02

Abstract

The Content Delivery Network Interconnection (CDNI) Metadata interface enables interconnected Content Delivery Networks (CDNs) to exchange content distribution metadata in order to enable content acquisition and delivery. To facilitate a wider set of use cases such as Open Caching, this document describes extensions to the CDNI Metadata object model and its associated Capabilities model as documented in "CDNI Metadata" RFC8006 and "CDNI Request Routing: Footprint and Capabilities Semantics" RFC8008 .

This document is a reflection of the content in the Streaming Video Alliance specification titled "SVA Configuration Interface: Part 2 Extensions to CDNI Metadata Object Model".

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction and Scope	4
1.1. Terminology	5
1.2. Requirements Language	6
2. CDNI Metadata Model Extensions	6
2.1. Cache Control Metadata	8
2.1.1. MI.CachePolicy	9
2.1.2. MI.NegativeCachePolicy	11
2.1.3. MI.StaleContentCachePolicy	12
2.1.4. MI.CacheBypassPolicy	14
2.1.5. MI.ComputedCacheKey	16
2.2. Origin Access Metadata	17
2.2.1. MI.SourceMetadataExtended	18
2.2.1.1. MI.SourceExtended	19
2.2.1.2. MI.LoadBalanceMetadata	22
2.2.2. MI.Auth	24
2.2.2.1. MI.HeaderAuth	24
2.2.2.2. MI.AWSSv4Auth	26
2.3. Edge Control Metadata	28
2.3.1. MI.CrossOriginPolicy	28
2.3.1.1. MI.AccessControlAllowOrigin	31
2.3.2. MI.AllowCompress	33
2.3.3. MI.TrafficType	35
2.3.4. MI.OcnSelection	36
2.3.4.1. MI.OcnDelivery	37
2.4. Processing Stage Metadata	38
2.4.1. MI.ProcessingStages	40
2.4.2. MI.StageRules	43
2.4.3. MI.ExpressionMatch	45
2.4.4. MI.StageMetadata	46
2.4.5. MI.RequestTransform	48
2.4.6. MI.ResponseTransform	49
2.4.7. MI.SyntheticResponse	50

2.4.8.	MI.HeaderTransform	52
2.4.9.	MI.HTTPHeader	54
2.5.	General Metadata	55
2.5.1.	MI.ServiceIDs	55
2.5.2.	MI.PrivateFeatureList	57
2.5.2.1.	MI.PrivateFeature	58
2.5.3.	MI.RequestRouting	59
3.	Metadata Expression Language	60
3.1.	Expression Variables	62
3.2.	Expression Operators and keywords	63
3.3.	Expression Built-in Functions	64
3.3.1.	Basic Functions: Type Conversions	64
3.3.2.	Basic Functions: String Conversions	65
3.3.3.	Convenience Functions	65
3.4.	Error Handling	66
3.4.1.	Compile Time Errors	66
3.4.2.	Runtime Errors	67
3.5.	Expression Examples	67
3.5.1.	ComputedCacheKey	67
3.5.2.	ExpressionMatch	67
3.5.3.	ResponseTransform	68
3.5.4.	MI.ServiceIDs	68
4.	CDNI Capabilities Extensions	69
4.1.	FCI Metadata Object	69
4.2.	FCI Model Extensions	70
4.2.1.	FCI.AuthTypes	70
4.2.2.	FCI.ProcessingStages	72
4.2.3.	FCI.SourceMetadataExtended	73
4.2.4.	FCI.RequestRouting	74
4.2.5.	FCI.PrivateFeatures	75
4.2.5.1.	FCI.PrivateFeature	76
4.2.6.	FCI.OcnSelection	76
5.	IANA Considerations	77
5.1.	CDNI Payload Types	77
6.	Security Considerations	79
7.	Conclusion	79
8.	References	79
8.1.	Normative References	79
8.2.	Informative References	80
	Authors' Addresses	81

1. Introduction and Scope

The Content Delivery Network Interconnection (CDNI) Metadata interface enables interconnected Content Delivery Networks (CDNs) to exchange content distribution metadata in order to enable content acquisition and delivery. To facilitate a wider set of use cases encountered in the commercial CDN and Open Caching ecosystems, this document describes extensions to the CDNI Metadata object model and its associated Capabilities model.

The objectives of this document are:

1. Identify the requirements for extending [RFC8006] and [RFC8008] and specify a set of extensions that realize these requirements.
2. Maintain backward compatibility with [RFC8006] and [RFC8008] by not altering the definitions or semantics of the original object model. All extensions are defined as new GenericMetadata Objects.
3. Define the metadata object model independently of the APIs used to publish and retrieve metadata.

Scope this document ADDRESSES:

1. Define and register CDNI GenericMetadata objects, as defined in section 4 of [RFC8006].
2. Define and register CDNI Payload Types, as defined in section 7.1 of [RFC8006].
3. Define Capabilities Objects that facilitate advertisement of a dCDN's support of these new metadata features, extending definitions in section 5 of [RFC8008].
4. Specification of a Metadata Expression Language Section 3 used within the metadata object model extensions.
5. Provide JSON examples illustrating real-world CDN and Open Caching use cases.

Scope this document DOES NOT ADDRESS:

1. Metadata object model definitions already specified in [RFC8006].
2. Interface API definitions for publishing and retrieving configuration metadata. The Metadata Interface (MI) as defined in [RFC8006] can be used to retrieve metadata. To enable more

sophisticated metadata configuration publishing workflows, the Streaming Video Alliance (SVA) Open Caching API [OC-CI], as documented in the SVA Configuration Interface Part 3 (Simple API) and Part 4 (Advanced API) specifications can be used.

1.1. Terminology

For consistency with other CDNI documents this document follows the CDNI convention of uCDN (upstream CDN) and dCDN (downstream CDN). It should be noted, however, that uCDN and dCDN are roles that can be played by a variety of entities in the distribution ecosystem. A Content Provider, for example, can play the roles of a uCDN, while a commercial CDN or Open Caching system can play either the roles of a uCDN or dCDN. Additionally, this document reuses the terminology defined in [RFC6707], [RFC7336], [RFC8006], [RFC8007] and [RFC8804].

The following terms are used throughout this document:

- * API - Application Programming Interface
- * AWS - Amazon Web Services
- * CDN - Content Delivery Network
- * CDNi - CDN Interconnect
- * CORS - Cross-Origin Resource Sharing
- * CP - Content Provider
- * dCDN - Downstream CDN
- * DNS - Domain Name System
- * FCI - Footprint and Capabilities Advertising Interface
- * HREF - Hypertext Reference (link)
- * HTTP - Hypertext Transfer Protocol
- * IETF - Internet Engineering Task Force
- * ISP - Internet Service Provider
- * JSON - JavaScript Object Notation
- * MEL - Metadata Expression Language

- * Object - A collection of properties.
- * OC - Open Caching
- * OCN - Open Caching Node
- * PatternMatch - An object which matches a string pattern
- * UA - User Agent
- * uCDN - Upstream CDN
- * URI - Uniform Resource Identifier
- * URN - Uniform Resource Name
- * VOD - Video-on-Demand
- * W3C - World Wide Web Consortium

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. CDNI Metadata Model Extensions

This section details extensions to the CDNI Metadata model as defined in Section 4 of [RFC8006], expressed as a set of new GenericMetadata objects. To preserve backward compatibility with [RFC8006], no changes are proposed to the original set of GenericMetadata.

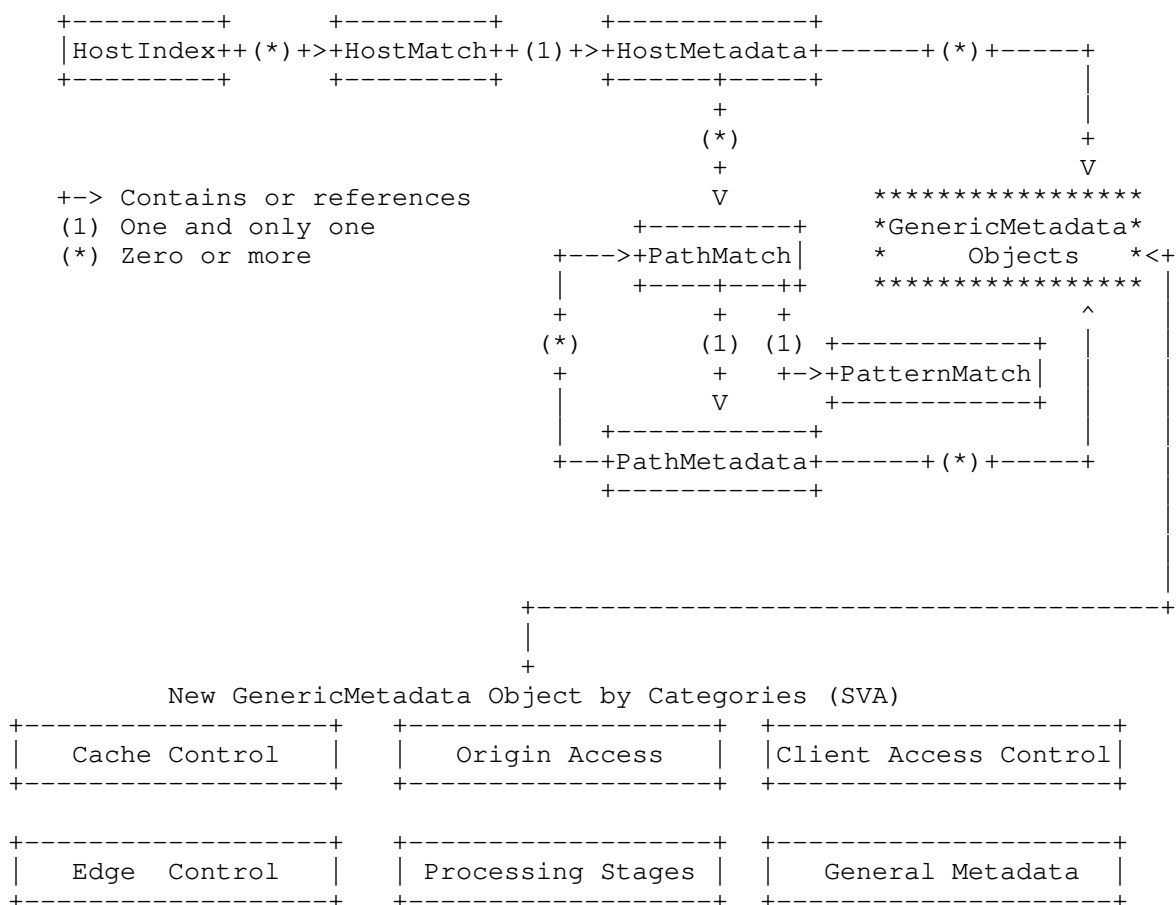


Figure 1: CDNI Metadata Model with Extensions

The remainder of this section presents the extended set of GenericMetadata objects organized by the categories in the above diagram.

Note: In the following sections, the term "mandatory-to-specify" is used to convey which properties MUST be included when serializing a given capability object. When mandatory-to-specify is defined as "Yes" for an individual property, it means that if the object containing that property is included in a message, then the mandatory-to-specify property MUST also be included.

2.1. Cache Control Metadata

In addition to the cache control policies currently specified by CDNI metadata, content providers often need more fine-grained control over CDN caching, including scenarios where it is desirable to override or adjust cache-control headers from the origin.

The following additional capabilities are needed for general CDN and open caching use cases:

1. Positive Cache Control - Allows the uCDN to specify internal caching policies for the dCDN and external caching policies advertised to clients of the dCDN, overriding any cache control policy set in the response from the uCDN.
2. Negative Cache Control - Allows the specification of caching policies based on error response codes received from the origin, allowing for fine-grained control of the downstream caching of error responses. For example, it may be desirable to cache error responses at the dCDN for a short period of time to prevent an overwhelmed origin service or uCDN from being flooded with requests.
3. Cache Bypass Control - Allows content providers to bypass CDN caching when needed (typically for testing or performance benchmarking purposes).
4. Stale Content Policies - Allows control over how the dCDN should process requests for stale content. For example, this policy allows the content provider to specify that stale content be served from cache for a specified time period while refreshes from the origin occur asynchronously.
5. Dynamically Constructed Cache Keys - While the properties provided by the standard CDNI metadata Cache object provide some simple control over the construction of the cache key, it is typical in advanced CDN configurations to generate cache keys that are dynamically constructed via lightweight processing of various properties of the HTTP request and/or response. As an example, an origin may specify a cache key as a value returned in a specific HTTP response header. A rich expression language is provided to allow for such advanced cache key construction.

2.1.1.1. MI.CachePolicy

CachePolicy is a new GenericMetadata object that allows for the uCDN to specify internal caching policies for the dCDN, as well as external caching policies advertised to clients of the dCDN (overriding any cache control policy set in the response from the uCDN).

Property: internal

- Description: Specifies the internal cache control policy to be used by the dCDN.
- Type: Number in seconds encoded as string (e.g. 5 is a five second cache) and/or a list of Enumeration [as-is|no-cache|no-store]
- Mandatory-to-Specify: No. The default is to use the cache control policy specified in the response from the uCDN.

Property: external

- Description: Specifies the external cache control policy to be used by clients of this dCDN.
- Type: Number in seconds encoded as string (e.g. 5 is a five second cache) and/or a list of Enumeration [as-is|no-cache|no-store]
- Mandatory-to-Specify: No. The default is to use the cache control policy specified in the response from the uCDN.

Property: force

- Description: If set to True, the metadata interface cache policy defined in the MI.CachePolicy will override any cache control policy set in the response from the uCDN. If set to False, the MI.CachePolicy is only used if there is no cache control policy provided in the response from the uCDN.
- Type: Boolean
- Mandatory-to-Specify: No. The default is "False", which will apply the MI.CachePolicy only if no policy is provided in the response from the uCDN.

Example 1: An MI.CachePolicy that sets the internal cache control policy to five seconds. The external cache policy is set to 'no-cache':

```
{
  "generic-metadata-type": "MI.CachePolicy",
  "generic-metadata-value": {
    "internal": "5",
    "external": "no-cache",
    "force": "true"
  }
}
```

Example 2: An MI.CachePolicy that sets the internal cache control policy to "as-is" (keep the policy set in the response from the uCDN). The external cache policy is set to 'no-cache':

```
{
  "generic-metadata-type": "MI.CachePolicy",
  "generic-metadata-value": {
    "internal": "as-is",
    "external": "no-cache",
    "force": "true"
  }
}
```

Example 3: An MI.CachePolicy in the context of the processing stages model that sets a caching policy only if the HTTP status code received from the origin is a 200. In this example, the internal cache control policy is set to five seconds. The external cache policy is set to 'no-cache'. Force is set to 'False', indicating that the MI.CachePolicy only applies if there is no cache policy in the response from the uCDN.

```

{
  "generic-metadata-type": "MI.ProcessingStages",
  "generic-metadata-value": {
    "origin-response": [
      {
        "match": {
          "expression": "resp.status == 200"
        },
        "stage-metadata": {
          "generic-metadata": [
            {
              "generic-metadata-type": "MI.CachePolicy",
              "generic-metadata-value": {
                "internal": "5",
                "external": "no-cache",
                "force": "false"
              }
            }
          ]
        }
      }
    ]
  }
}

```

2.1.2. MI.NegativeCachePolicy

NegativeCachePolicy is a new GenericMetadata object that allows for the specification of caching policies based on error response codes received from the origin.

Property: error-codes

- Description: Array of HTTP response error status codes (See Sections 6.5 and 6.6 of [RFC7231] , that if returned from the uCDN, will be cached using the cache policy defined by the cache-policy property.
- Type: Array of HTTP response error status codes
- Values: ["404", "503", "504"]
- Mandatory-to-Specify: No. The default is to revert to [RFC8006] behavior.

Property: cache-policy

- Description: MI.CachePolicy to apply to the HTTP response error status codes returned by the uCDN.
- Mandatory-to-Specify: Yes

Example: A MI.NegativeCachePolicy that applies to HTTP error codes: "404", "503", "504" and sets the internal cache control policy to five seconds and external to 'no-cache'.

```
{
  "generic-metadata-type": "MI.NegativeCachePolicy",
  "generic-metadata-value": {
    "error-codes": [ "404", "503", "504" ],
    "cache-policy": {
      "internal": "5",
      "external": "no-cache",
      "force": "true"
    }
  }
}
```

2.1.3. MI.StaleContentCachePolicy

MI.StaleContentCachePolicy is a new GenericMetadata object that allows the uCDN to specify the policy to use by a dCDN when responding with stale content. For example, this policy allows the content provider to specify that stale content be served from cache for a specified time period while refreshes from the origin occur asynchronously.

Property: stale-while-revalidating

- Description: Instructs the dCDN to serve a stale version of a resource while refreshing the resource with the uCDN. When set to "True", the dCDN will return a previously cached version of a resource while the resource is refreshed with the uCDN in the background.
- Type: Boolean
- Mandatory-to-Specify: No. The default is False, which waits for the uCDN to refresh a resource before responding to the client.

Property: stale-if-error

- **Description:** Instructs the dCDN to serve a stale version of a resource if an error was received when trying to refresh the resource with the uCDN. When set, the dCDN will return a previously cached version of a resource instead of caching the error response. Per Section 4 of [RFC5861], an error is any situation that would result in a 500, 502, 503, or 504 HTTP response status code being returned
- **Type:** Array of HTTP response error status codes. Example: ["503", "504"]
- **Mandatory-to-Specify:** No. The default is to cache the error response received from the uCDN.

Property: failed-refresh-ttl

- **Description:** Instructs the dCDN to serve a stale version of a resource for the number of seconds specified in failed-refresh-ttl before trying to revalidate the resource with the uCDN. Use of failed-refresh-ttl allows the load to be reduced on the uCDN during times of system stress.
- **Type:** Integer
- **Mandatory-to-Specify:** No

Example 1: A MI.StaleContentCachePolicy where stale-while-revalidating is true, instructing the dCDN to respond with a stale cached version of the resource while it refreshes the resource with the uCDN in the background:

```
{
  "generic-metadata-type": "MI.StaleContentCachePolicy",
  "generic-metadata-value": {
    "stale-while-revalidating": true
  }
}
```

Example 2: A MI.StaleContentCachePolicy where stale-if-error instructs the dCDN to use the stale cached resource if it receives an error of type 503 or 504 when trying to refresh the resource with the uCDN.

failed-refresh-ttl instructs the dCDN to use a five second cache TTL on the resource that receives an error when refreshing from the uCDN. That is, after five seconds, the dCDN will attempt to refresh the resource with the uCDN.

```
{
  "generic-metadata-type": "MI.StaleContentCachePolicy",
  "generic-metadata-value": {
    "stale-if-error": [ "503", "504" ],
    "failed-refresh-ttl": "5"
  }
}
```

Example 3: A `MI.StaleContentCachePolicy` where `stale-while-revalidating` is `true`, instructing the dCDN to respond with a stale cached version of the resource while it refreshes the resource with the uCDN in the background.

`stale-if-error` instructs the dCDN to use the stale cached resource if it receives an error of type 404, 503, or 504 when trying to refresh the resource with the uCDN.

`failed-refresh-ttl` instructs the dCDN to use a five second cache TTL on the resource that receives an error when refreshing from the uCDN. That is, after five seconds, the dCDN will attempt to refresh the resource with the uCDN.

```
{
  "generic-metadata-type": "MI.StaleContentCachePolicy",
  "generic-metadata-value": {
    "stale-while-revalidating": "true",
    "stale-if-error": [ "404", "503", "504" ],
    "failed-refresh-ttl": "5"
  }
}
```

2.1.4. `MI.CacheBypassPolicy`

`CacheBypassPolicy` is a new `GenericMetadata` object that allows a client request to be set as non-cacheable. It is expected that this feature will be used to allow clients to bypass cache when testing the uCDN fill path. Note: `CacheBypassPolicy` is typically used in conjunction with a path match or match expression on a header value or query parameter. Any content previously cached (by client requests that do not set `CacheBypassPolicy`) is not evicted.

Property: `bypass-cache`

- Description: A Boolean value that can activate the feature for a given client request. It is expected that this feature will be used within `ProcessingStages` to allow a client request to be marked to bypass cache.

- Type: Boolean
- Mandatory-to-Specify: No. The default is False.

Example 1: A MI.CacheBypassPolicy with the client HTTP header of:
CDN-BYPASS: "True":

```
{
  "generic-metadata-type": "MI.ProcessingStages",
  "generic-metadata-value": {
    "client-request": [
      {
        "match": {
          "expression": "req.h.cdn-bypass == 'true'"
        },
        "stage-metadata": {
          "generic-metadata": [
            {
              "generic-metadata-type": "MI.CacheBypassPolicy",
              "generic-metadata-value": {
                "bypass-cache": "true"
              }
            }
          ]
        }
      }
    ]
  }
}
```

Example 2: A MI.CacheBypassPolicy that applies to all requests where the host header is bypass.example.com:

```

{
  "generic-metadata-type": "MI.ProcessingStages",
  "generic-metadata-value": {
    "client-request": [
      {
        "match": {
          "expression": "req.h.host == 'bypass.example.com'"
        },
        "stage-metadata": {
          "generic-metadata": [
            {
              "generic-metadata-type": "MI.CacheBypassPolicy",
              "generic-metadata-value": {
                "bypass-cache": "true"
              }
            }
          ]
        }
      }
    ]
  }
}

```

2.1.1.5. MI.ComputedCacheKey

While the properties provided by the standard CDNI metadata Cache object (See Section 4.2.6 of [RFC8006]) provide some simple control over the construction of the cache key, it is typical in advanced CDN configurations to generate cache keys that are dynamically constructed via lightweight processing of various properties of the HTTP request and/or response. As an example, an origin may specify a cache key as a value returned in a specific HTTP response header.

ComputedCacheKey is a new GenericMetadata object that allows for the specification of a cache key using the metadata expression language. Typical use cases would involve the construction of a cache key from one or more elements of the HTTP request. In cases where both the ComputedCacheKey and the Cache object are applied, the ComputedCacheKey will take precedence.

Property: expression

- Description: The expression that specifies how the cache key shall be constructed.
- Type: String. An expression using [CDNI-MEL] to dynamically construct the cache key from elements of the HTTP request and/or response.

- Mandatory-to-Specify: Yes

Example, using a custom request header as the cache key instead of the URI path:

```
{
  "generic-metadata-type": "MI.ComputedCacheKey",
  "generic-metadata-value": {
    "expression": "req.h.X-Cache-Key"
  }
}
```

2.2. Origin Access Metadata

The CDNI metadata definitions for sources (also known as origins in the CDN industry), are extended to provide the following capabilities required:

1. Designation as to whether a source requires HTTPS access.
2. Specification of the source's TCP port number.
3. Web root path specification for the source.
4. Indication as to whether redirects should be followed.
5. Support for additional forms of origin authentication.
6. Multi-origin failover - The ability to specify a list of origins that can act as fallbacks to the primary origin. Failure rules can specify types of errors and timeout values that trigger failover.
7. Multi-origin load balancing - The ability to specify a list of origins that can be selected by one of several balancing rules (round robin, content hash, IP hash).
8. Specification of SNI configurations required for origin access.
9. Specification of connection control parameters for origin access.

2.2.1. MI.SourceMetadataExtended

SourceMetadataExtended is an alternative to the CDNI standard SourceMetadata object, which adds a property to specify load balancing across multiple sources, as well as a SourceExtended sub-object with additional attributes to the CDNI standard Source object. While both SourceMetadataExtended and SourceMetadata can be provided for backward compatibility, a dCDN that advertises capability for SourceMetadataExtended will ignore SourceMetadata if both are provided for a given host or path match.

Property: sources

- Description: Sources from which the dCDN can acquire content, listed in order of preference.
- Type: Array of SourceExtended objects
- Mandatory-to-Specify: No. Default is to use static configuration, out-of-band from the CDNI metadata interface.

Property: load-balance

- Description: Specifies load balancing rules for the set of sources.
- Type: LoadBalanceMetadata object
- Mandatory-to-Specify: No

Example of a SourceMetadataExtended object with the associated LoadBalanceMetadata configuration object:

```
{
  "generic-metadata-type": "MI.SourceMetadataExtended",
  "generic-metadata-value": {
    "sources": [
      {
        "endpoints": [
          "a.service123.ucdn.example",
          "b.service123.ucdn.example"
        ],
        "protocol": "http/1.1"
      },
      {
        "endpoints": [
          "origin.service123.example"
        ],
        "protocol": "http/1.1"
      }
    ],
    "load-balance": {
      "balance-algorithm": "content-hash",
      "balance-path-pattern": "^/prod/(.*)/.*\\.ts$"
    }
  }
}
```

2.2.1.1. MI.SourceExtended

SourceExtended is an alternative to the CDNI standard Source object with additional metadata. It contains all the attributes of the [RFC8006] Source object (acquisition-auth, endpoints, and protocol), with additions specified below.

Property: acquisition-auth

- Description: Authentication method to use when requesting content from this source. Same as [RFC8006].
- Type: Auth (see [RFC8006] Section 4.2.7 and the new MI.Auth types in this specification)
- Mandatory-to-Specify: No. Default is no authentication required.

Property: endpoints

- Description: Origins from which the dCDN can acquire content. If multiple endpoints are specified, they are all equal, i.e., the list is not ordered by preference. Same as [RFC8006].

- Type: Array of Endpoint objects (see [RFC8006] Section 4.3.3)
- Mandatory-to-Specify: Yes..

Property: protocol

- Description: Network retrieval protocol to use when requesting content from this source. Same as [RFC8006].
- Type: Protocol (see [RFC8006] Section 4.3.2)
- Mandatory-to-Specify: Yes..

Property: origin-host

- Description: HTTP host header to pass to the endpoints when retrieving content from a uCDN. The host MUST conform to the Domain Name System (DNS) syntax defined in [RFC1034] and [RFC1123]
- Type: String
- Mandatory-to-Specify: No. The default is to use the host name passed by the dCDN.

Property: webroot

- Description: The path element that is prepended to a resource's URI before retrieving content from a uCDN.
- Type: String
- Mandatory-to-Specify: No. The default is to use the original URI.

Property: follow-redirects

- Description: If the follow-redirects property is set to "True", HTTP redirect responses returned from a uCDN will be followed when retrieving content. Otherwise, the HTTP redirect response is returned to the client.
- Type: Boolean
- Mandatory-to-Specify: No. The default is "True" (i.e., follow redirect responses from the uCDN).

Property: timeout-ms

- Description: A timeout (in milliseconds) to apply when connecting to a uCDN. If the connection is not established within timeout-ms, this source is abandoned and the next source in the MI.SourceMetadataExtended sources array is tried. Once a connection is established, timeout-ms is used on subsequent reads of data from the uCDN.
- Type: Integer
- Mandatory-to-Specify: No. The default is to revert to [RFC8006] behavior.

Property: failover-errors

- Description: Array of HTTP response error status codes (Section 6 of [RFC7231]), that if returned from the uCDN, will trigger a failover to the next source in the MI.SourceMetadataExtended sources array. If the uCDN returns an HTTP error code that is not in the failover-errors array, that error code is returned to the client of the dCDN.
- Type: Array of HTTP response error status codes.
- Mandatory-to-Specify: No. The default is to revert to [RFC8006] behavior.

Example of a SourceExtended object that describes a pair of endpoints (servers) that the dCDN can use to acquire content for the applicable host and/or URI path:

```

{
  "generic-metadata-type": "MI.SourceMetadataExtended",
  "generic-metadata-value": {
    "sources": [
      {
        "endpoints": [
          "a.service123.ucdn.example",
          "b.service123.ucdn.example:8443"
        ],
        "protocol": "https/1.1",
        "origin-host": "internal.example.com",
        "webroot": "/prod",
        "follow-redirects": false,
        "timeout-ms": 4000,
        "failover-errors": [ "502", "503", "504" ]
      },
      {
        "endpoints": [ "origin.service123.example" ],
        "protocol": "http/1.1",
        "webroot": "/prod",
        "follow-redirects": true,
        "timeout-ms": 8000
      }
    ]
  }
}

```

2.2.1.2. MI.LoadBalanceMetadata

The LoadBalanceMetadata object defines how content acquisition requests are distributed over the SourceExtended objects listed in the SourceMetadataExtended object.

Property: balance-algorithm

- Description: Specifies the algorithm to be used when distributing content acquisition requests over the sources in a SourceMetadataExtended object. The available algorithms are random, content-hash, and ip-hash.

o random: Requests are distributed over the sources in proportion to their associated weights.

o content-hash: Requests are distributed over the sources in a consistent fashion, based on the balance-path-pattern property.

o ip-hash: Requests are distributed over the sources in a consistent fashion based on the IP address of the client requestor.

1. Type: Enumeration [random|content-hash|ip-hash] encoded as a lowercase string.
2. Mandatory-to-Specify: No. The default is to use sources in preference order as defined in the SourceMetadataExtended object.

Property: balance-weights

- Description: This property specifies the relative frequency that a source is used when distributing requests. For example, if there are three SourceExtended objects in a SourceMetadataExtended object with balance-weights [1, 2, 1], source 1 will receive 1/4 of the requests; source 2 will receive 2/4 of the requests; source 3 will receive 1/4 of the requests.
- Type: Array of integers
- Mandatory-to-Specify: No. The default is to use sources in preference order as defined in the SourceMetadataExtended object.

Property: balance-path-pattern

- Description: This property specifies a regular expression pattern to apply to the URI when calculating the content hash used by the balance-algorithm. For example, "balance-path-pattern": "^/prod/(.*)/.*\\.ts\$" would distribute requests based on the URI but excluding the /prod prefix and the .ts segment file.
- Type: String (regular expression)
- Mandatory-to-Specify: No. The default is to use the original URI.

Example 1: The LoadBalanceMetadata object distributes content acquisition requests over sources using a content-hash algorithm:

```
{
  "generic-metadata-type": "MI.LoadBalanceMetadata",
  "generic-metadata-value": {
    "balance-algorithm": "content-hash",
    "balance-path-pattern": "^/prod/(.*)/.*\\.ts$"
  }
}
```

Example 2: The LoadBalanceMetadata object distributes content acquisition requests over sources using the random algorithm:

```
{
  "generic-metadata-type": "MI.LoadBalanceMetadata",
  "generic-metadata-value": {
    "balance-algorithm": "random",
    "balance-weights": [ 1, 2, 1 ]
  }
}
```

2.2.2. MI.Auth

To meet the typical industry requirements for authenticating CDNs to external origins, two new authentication types are defined.

auth-type: MI.HeaderAuth

- Description: Header based authentication is used to pass an HTTP header (secret-name) and value (secret-value) to a uCDN when requesting content. The header name and value are agreed upon between parties out of band.
- Note: We may want to add a way to encrypt or separately communicate the secret; this could be a general capability for CDNI.
- auth-value: MI.HeaderAuth object specifying the header name and value (secret name, secret key) required for authenticated access to an origin. For more information, refer to the MI.HeaderAuth section below.

auth-type: MI.AWSSv4Auth

- Description: Allows for the specification of a set of headers to be added to requests that are forwarded to an origin to enable Amazon Web Services (AWS) authentication, as documented by AWS (See Specifications & Standards References).
- auth-value: MI.AWSSv4Auth object specifying the access parameters. For more information, refer to the MI.AWSSv4Auth section below.

2.2.2.1. MI.HeaderAuth

The HeaderAuth metadata object is used in the auth-value property of the

Auth object, as defined in [RFC8006] section 4.2.7, and may be applied to any

source by including or referencing it under its authentication property. This method of authentication provides a simple capability for a mutually agreed upon header to be added by the CDN to all requests sent to a specific origin. Note that if a dynamically generated header value is required, the RequestTransform capabilities within StageProcessing can be used.

Property: header-name

- Description: Name of the authentication header.
- Type: String
- Mandatory-to-Specify: Yes

Property: header-value

- Description: Value of the authentication header (typically a pre-shared key). Note that this value SHOULD NOT be disclosed; it SHOULD be protected by some mechanism such as a secret-sharing API, which is outside the scope of this specification.
- Type: String
- Mandatory-to-Specify: Yes

Example Auth object for header authentication:

```

{
  "generic-metadata-type": "MI.SourceMetadataExtended",
  "generic-metadata-value": {
    "sources": [
      {
        "endpoints": [ "origin.example.com" ],
        "protocol": "http/1.1",
        "acquisition-auth": {
          "generic-metadata-type": "MI.Auth",
          "generic-metadata-value": {
            "auth-type": "MI.HeaderAuth",
            "auth-value": {
              "header-name": "X-Origin-Auth",
              "header-value": "SECRETKEYJKSDHFSIFUI4UFH78HW4NF7"
            }
          }
        }
      }
    ]
  }
}

```

2.2.2.2. MI.AWsv4Auth

The AWsv4Auth metadata object is used in the auth-value property of the Auth object as defined in [RFC8006] section 4.2.7, and may be applied to any source by including or referencing it under its authentication property.

AWsv4 authentication causes upstream requests to have a signature applied, following the method described in [AWsv4Method]. A hash-based signature is calculated over the request URI and specified headers, and provided in an Authorization: header on the upstream request. The signature is tied to a pre-shared secret key specific to an AWS service, region, and key ID.

1. We may want to add a way to encrypt or separately communicate the secret; this could be a general capability for CDNI.
2. We may want to add optional properties that allow overriding the default headers to sign.
3. We may want to add optional properties that allow the signature to be sent in a way other than with the Authorization: header (e.g., query strings are also supported).

Property: access-key-id

- Description: The preconfigured ID of the pre-shared authorization secret.
- Type: String
- Mandatory-to-Specify: Yes

Property: secret-access-key

- Description: The pre-shared authorization secret, which is the basis of building the signature. This is a secret key that SHOULD NOT be disclosed; it SHOULD be protected by some mechanism such as a secret-sharing API, which is outside the scope of this specification.
- Type: String
- Mandatory-to-Specify: Yes

Property: aws-region

- Description: The AWS region name that is hosting the service and shares the key ID and corresponding pre-shared secret.
- Type: String
- Mandatory-to-Specify: Yes

Property: aws-service

- Description: The AWS service name that is serving the upstream requests.
- Type: String
- Mandatory-to-Specify: No. It defaults to "s3" if not specified.

Property: host-name

- Description: The host name to use as part of the signature calculation.
- Type: String

- **Mandatory-to-Specify:** No. It defaults to using the value of the `Host` header of the upstream request. This property is available in case the application needs to override that behavior.

Example Auth object for AWSv4 authentication:

```
{
  "generic-metadata-type": "MI.SourceMetadataExtended",
  "generic-metadata-value": {
    "sources": [
      {
        "endpoints": [ "origin.example.com" ],
        "protocol": "http/1.1",
        "acquisition-auth": {
          "generic-metadata-type": "MI.Auth",
          "generic-metadata-value": {
            "auth-type": "MI.AWSv4Auth",
            "auth-value": {
              "access-key-id": "MYACCESSKEYID",
              "secret-access-key": "SECRETKEYJKSDHFSIUHKWRGHHF",
              "aws-region": "us-west-1"
            }
          }
        }
      }
    ]
  }
}
```

2.3. Edge Control Metadata

CDNs typically require a set of configuration metadata to inform processing of responses downstream (at the edge and in the user agent). This section specifies GenericMetadata objects to meet those requirements.

2.3.1. MI.CrossOriginPolicy

Delegation of traffic between a CDN over an open caching node based on HTTP redirection does change the domain name in the client requests. This represents a cross-origin request that must be managed appropriately using Cross-Origin Resource Sharing (CORS) headers in the responses.

The dynamic generation of CORS headers is typical in modern HTTP request processing and avoids CORS validation forwarded to the CDN origin servers, particularly with the preflight OPTIONS requests. The CDNI metadata model requires extensions to specify how a CDN or open caching node should generate and evaluate these headers.

Required capabilities:

1. Set a default value for CORS response headers independent of the origin request header value.
2. Match the origin request header with a list of valid values, including PatternMatch, to return or not return the CORS response headers.
3. Set a list of custom headers that can be exposed to the client (expose headers).
4. Support for preflight requests using the OPTIONS method, including custom header validation, expose headers, and methods.
5. Support for credentials validation within CORS.

Simple CORS requests are those where both HTTP method and headers in the request are included in the safe list defined by the World Wide Web Consortium [W3C]. The user agent (UA) request can include an origin header set to the URL domain of the webpage that the player runs. Depending on the metadata configuration, the logic to apply by the open caching node (OCN) is:

1. Validation of the origin header - Metadata can include a list of valid domains to validate the request origin header. If it does not match, the CORS header must not be included in the response.
2. Wildcard usage - Depending on the configuration, the resultant CORS header to include in the response will be the same as the request origin header, or a wildcard.
3. If no validation of request is included in the origin header, set a default value for CORS response headers independent of the origin request header value.

When a UA makes a request that includes a method or headers that are not included in the safe-list, the client will make a CORS preflight request using the OPTIONS method to the resource including the origin header. If CORS is enabled and the requests passes the origin validation, the OCN SHOULD respond with the set of headers that indicate what is permitted for that resource, including one or more of the following:

1. Allowed methods
2. Allowed credentials
3. Allowed request headers
4. Max age that the OPTIONS request is valid
5. Headers that can be exposed to the client

CrossoriginPolicy is a GenericMetadata object that allows for the specification of dynamically generated CORS headers.

Property: allow-origin

- Description: Validation of simple CORS requests.
- Type: Object MI.AccessControlAllowOrigin
- Values: One element for each of the following properties.
- Mandatory-to-Specify: Yes

Property: expose-headers

- Description: A list of values the OCN will include in the Access-Control-Expose-Headers response header to a preflight request.
- Type: Array of strings
- Mandatory-to-Specify: No

Property: allow-methods

- Description: A list of values the OCN will include in the Access-Control-Allow-Methods response header to a preflight request.
- Type: Array of strings

- Mandatory-to-Specify: No

Property: allow-headers

- Description: A list of values the OCN will include in the Access-Control-Allow-Headers response header to a preflight request.

- Type: Array of strings

- Mandatory-to-Specify: No

Property: allow-credentials

- Description: The value the OCN will include in the Access-Control-Allow-Credentials response header to a preflight request.

- Type: Boolean

- Mandatory-to-Specify: No

Property: max-age

- Description: The value the OCN will include in the Access-Control-Max-Age response header to a preflight request.

- Type: Integer

- Mandatory-to-Specify: No

2.3.1.1. MI.AccessControlAllowOrigin

The MI.AccessControlAllowOrigin object has the following properties:

Property: allow-list

- Description: List of valid URLs that will be used to match the request origin header. The Origin header is a HTTP extension. Its value is a version of the Referer header in some specific requests, and used for Cross Origin requests. . Permitted values are schema://hostname[:port]

- Type: Array of PatternMatch objects

- Mandatory-to-Specify: Yes

Property: wildcard-return

- Description: If "True", the OCN will include a wildcard (*) in the Access-Control-Allow-Origin response header. If "False", the OCN will reflect the request origin header in the Access-Control-Allow-Origin response header.
- Type: Boolean
- Mandatory-to-Specify: Yes

The examples below demonstrate how to configure response headers dynamically for CORS validation.

Example 1: A simple CORS validation configuration:

```
{
  "generic-metadata-type": "MI.CrossoriginPolicy",
  "generic-metadata-value": {
    "allow-origin": {
      "allow-list": [
        {
          "pattern": "*"
        }
      ],
      "wildcard-return": true
    }
  }
}
```

Example 2: Validation of a preflight request when some of the headers included in the subsequent object request are not included in the CORS specification safelist:


```
{
  "generic-metadata-type": "MI.CrossoriginPolicy",
  "generic-metadata-value": {
    "allow-origin": {
      "allow-list": [
        {
          "pattern": "*/://sourcepage.example.com"
        },
        "wildcard-return": false
      ],
      "allow-methods": [ "GET", "POST" ],
      "allow-credentials": true,
      "allow-headers": [ "X-PINGOTHER", "Content-Type" ],
      "expose-headers": [ "X-User", "Authorization" ],
      "max-age": 3600
    }
  }
}
```

2.3.2. MI.AllowCompress

Downstream CDNs often have the ability to compress HTTP response bodies in cases where the client has declared that it can accept compressed responses (via an Accept-Encoding header), but the source/origin has returned an uncompressed response.

The specific compression algorithm used by the dCDN is negotiated by the client's Accept-Encoding header according to [RFC7694] (including q= preferences) and the compression capabilities available on the dCDN.

In addition, HeaderTransform allows the uCDN to normalize, or modify, the Accept-Encoding header to allow for fine-grain control over the selection of the compression algorithm (e.g., gzip, compress, deflate, br, etc.).

AllowCompress is a new GenericMetadata object that allows the dCDN to compress content before sending to the client.

Property: allow-compress

- Description: If set to "True", then the dCDN will try to compress the response to the client based on the Accept-Encoding request header.
- Type: Boolean
- Values: True or False

- Mandatory-to-Specify: No. The default is "False".

Example 1: An MI.AllowCompress that allows manifests (*.m3u8) to be compressed by the dCDN:

```
{
  "match": {
    "expression": "req.h.uri *= '*.m3u8'"
  },
  "stage-metadata": {
    "generic-metadata": [
      {
        "generic-metadata-type": "MI.AllowCompress",
        "generic-metadata-value": {
          "allow-compress": "true"
        }
      }
    ]
  }
}
```

Example 2: An MI.AllowCompress that allows manifests (*.m3u8) to be compressed by the dCDN but normalizing the client's Accept-Encoding header:

```
{
  "match": {
    "expression": "req.h.accept-encoding *= '*gzip*'"
  },
  "stage-metadata": {
    "generic-metadata": [
      {
        "generic-metadata-type": "MI.AllowCompress",
        "generic-metadata-value": {
          "allow-compress": "true"
        }
      }
    ]
  }
}
```

2.3.3. MI.TrafficType

Content delivery networks often apply different infrastructure, network routes, and internal metadata for different types of traffic. Delivery of large static objects (such as software downloads), may, for example, use different network routes than video stream delivery. In an HTTP adaptive bitrate video service, every video title corresponds to a set of video files and descriptors according to different video protocols, and this is independent of the type of service (Video-on-Demand, Live, Catch-up, etc.).

The way the video service is consumed by the user agents can vary. For instance, a segment that belongs to a Video-on-Demand (VoD) title can be requested for every moment the content is available for the user agents to consume, while a segment of live content will be only requested as long as the time-shift duration is configured for that service. Knowing those differences, a CDN or OCN provider can implement specific strategies that will maximize performance and thereby provide more available capacity to the upstream provider. It should be noted that the dCDNs handling of the traffic types is implementation-specific and not prescribed here.

TrafficType metadata defines a set of descriptors that characterize either the type or usage of the traffic, enabling CDNs and OCNs to apply any internal configuration rules without exposing an unnecessary number of internal details. Note that the interpretation of these traffic types and application of rules such as rate limiting or delivery pacing are implementation specific.

Property: traffic-type

- Description: A literal that defines the traffic type. uCDN will use the literal that is most representative of the traffic being delegated.
- Type: Enumeration [vod, live, object-download] encoded as lowercase string
- Mandatory-to-Specify: Yes

Property: hints

- Description: Other traffic characteristics that the uCDN can indicate to the dCDN as suggestions for service optimization. Accepts free-form unconstrained values.
- Type: Array of strings

- Mandatory-to-Specify: No

A TrafficType definition example for HostMetadata:

```
{
  "generic-metadata-type": "MI.TrafficType",
  "generic-metadata-value": {
    "traffic-type": "vod",
    "hints": [ "low-latency", "catch-up" ]
  }
}
```

2.3.4. MI.OcnSelection

Configuration metadata is required to permit several levels of OCN selection policies. For example, in a mobile network, several physical locations are possible (i.e., candidates) for hosting the OCN that will take charge in the delegation for the uCDN. This is the case when the cache is virtualized and deployed dynamically. Depending on the OCN selection policy (which may be a cost driver), the dCDN may attempt to favor certain types of caches at the edge, for example. The default OCN selection policy might be "best-effort". Another one might be linked to the network characteristics like "Edge" or ("average latency < 10ms").

OcnSelection is a new GenericMetadata object that allows the uCDN to indicate to the dCDN a preference in terms of OCN selection.

Property: ocn-delivery

- Description: Instructs the dCDN to perform delegation operating a particular medium and/or a transport arrangement.
- Type: Object MI.OcnDelivery
- Mandatory-to-Specify: No. At least one of the two properties, ocn-type or ocn-delivery, must be present.

Property: ocn-type

- Description: Instructs the dCDN to perform delegation operating the type of open caching nodes.
- Type: A string corresponding to one of the open caching node types announced by the dCDN through the FCI interface.
- Mandatory-to-Specify: No. At least one of the two properties, ocn-type or ocn-delivery, must be present.

Property: ocn-selection

- Description: This property enforces the selection of OCNs, considering the ocn-type and/or the ocn-delivery properties. "False" means best-effort.
- Type: string. "attempt-or-failed" and "attempt-or-besteffort" mean that the delegation must be attempted considering the ocn-type and/or the ocn-delivery properties. If not possible, it is considered as an error and either fails (configuration failure) or the dCDN continues with a best-effort procedure. Last, "best effort" means the dCDN tries its best to fulfil the requested ocn-selection policy.
- Mandatory-to-Specify: No. Best-effort is the default OCN selection policy.

2.3.4.1. MI.OcnDelivery

An ocn-delivery object contains the following properties:

Property: ocn-medium

- Description: Instructs the dCDN to perform delegation operating a particular medium. The following values are specified: "SATELLITE".
- Type: String
- Mandatory-to-Specify: No. Either the ocn-medium property or the ocn-transport property must be present.

Property: ocn-transport

- Description: Instructs the dCDN to perform delegation operating a particular transport arrangement. The following values are specified: "MABR".
- Type: String
- Mandatory-to-Specify: No. At least one of the two properties (ocn-medium or ocn-transport) must be present.

2.4. Processing Stage Metadata

It is typical in CDN configurations to define matching rules and metadata that are to be applied at specific stages in the request processing pipeline. For example, it may be required to append a host header prior to forwarding a request to an origin, or modify the response returned from an origin prior to storing in the cache.



Figure 2: Processing stages

Processing stages:

1. `clientRequest` - Rules run on the client request prior to further processing.
2. `originRequest` - Rules run prior to making a request to the origin.
3. `originResponse` - Rules run after a response is received from the origin and before being placed in the cache.
4. `clientResponse` - Rules run prior to sending the response to the client. If the response is from the cache, rules are applied to the response retrieved from the cache prior to sending to the client.

Requirements:

1. **Header Matching** - While CDNI metadata defines some basic matching rules for host names and pattern matching on paths, CDN and open caching use cases often require matching on specific fields in Hypertext Transfer Protocol (HTTP) request and response headers to set metadata. A typical example may be matching on a user agent string to set access controls or matching on a mime-type header to set caching rules. A rich expression matching syntax that allows matching on any combination of host, path, and header values covers most typical use cases.

2. Expression Matching - Header matching alone is not always sufficient for identifying a set of requests or responses that require specific metadata. CDN and open caching systems often require a rich set of matching rules, with full regular expressions and Boolean combinations of matching parameters for host, path, and header elements of a request. In typical CDN implementations, this capability is provided by a rich expression language that can be embedded in the metadata configurations.
3. URI Modifications - In processing HTTP requests, modifications to the request Uniform Resource Identifier (URI) are often required for uses such as collapsing multiple paths to a common cache key or normalizing file extension naming conventions before making a request to the origin. In cases where the modified URI needs to be constructed dynamically, an expression language is provided that allows elements of requests and responses to be concatenated with string literals.
4. Header Modifications - In processing HTTP requests, it is often required to modify HTTP request or response headers at one of the processing stages, requiring CDNI metadata to have the capability to update any field in an HTTP request or response header. It should be noted that certain HTTP headers (such as Set-Cookie) have multiple occurrences in a request or response, thereby requiring that we allow for add and replace designations for header modification. In cases where a header value needs to be constructed dynamically, an expression language is provided that allows elements of requests and responses to be concatenated with string literals. All of the following capabilities are required at each processing stage:
 5. Add Request Header Field - Add a header name/value to the request, along with any headers of the same name that may already be present.
 6. Replace Request Header Field - Add a header name/value to the request, replacing any headers of the same name that may already be present.
 7. Delete Request Header Field - Delete all occurrences of the named header from the request.
 8. Add Response Header Field - Add a header name/value to the response, along with any headers of the same name that may already be present.

9. Replace Response Header Field - Add a header name/value to the response, replacing any headers of the same name that may already be present.
10. Delete Response Header Field - Delete all occurrences of the named header from the response.
11. Synthetic Responses - It is quite common in CDN configurations to specify a synthetic response be generated based on inspection of aspects of the original request or the origin response. The synthetic response capability allows for the specification of a set of response headers, a status code, and a response body. In cases where a header value or the synthetic response body needs to be constructed dynamically, an expression language is provided that allows elements of requests and responses to be concatenated with string literals.

2.4.1. MI.ProcessingStages

A ProcessingStages object is a new GenericMetadata which describes the matching rules, metadata, and transformations to be applied at specific stages in the request processing pipeline. The processing rules and transformations are defined as a child data model referenced within a ProcessingStages object, as defined below.

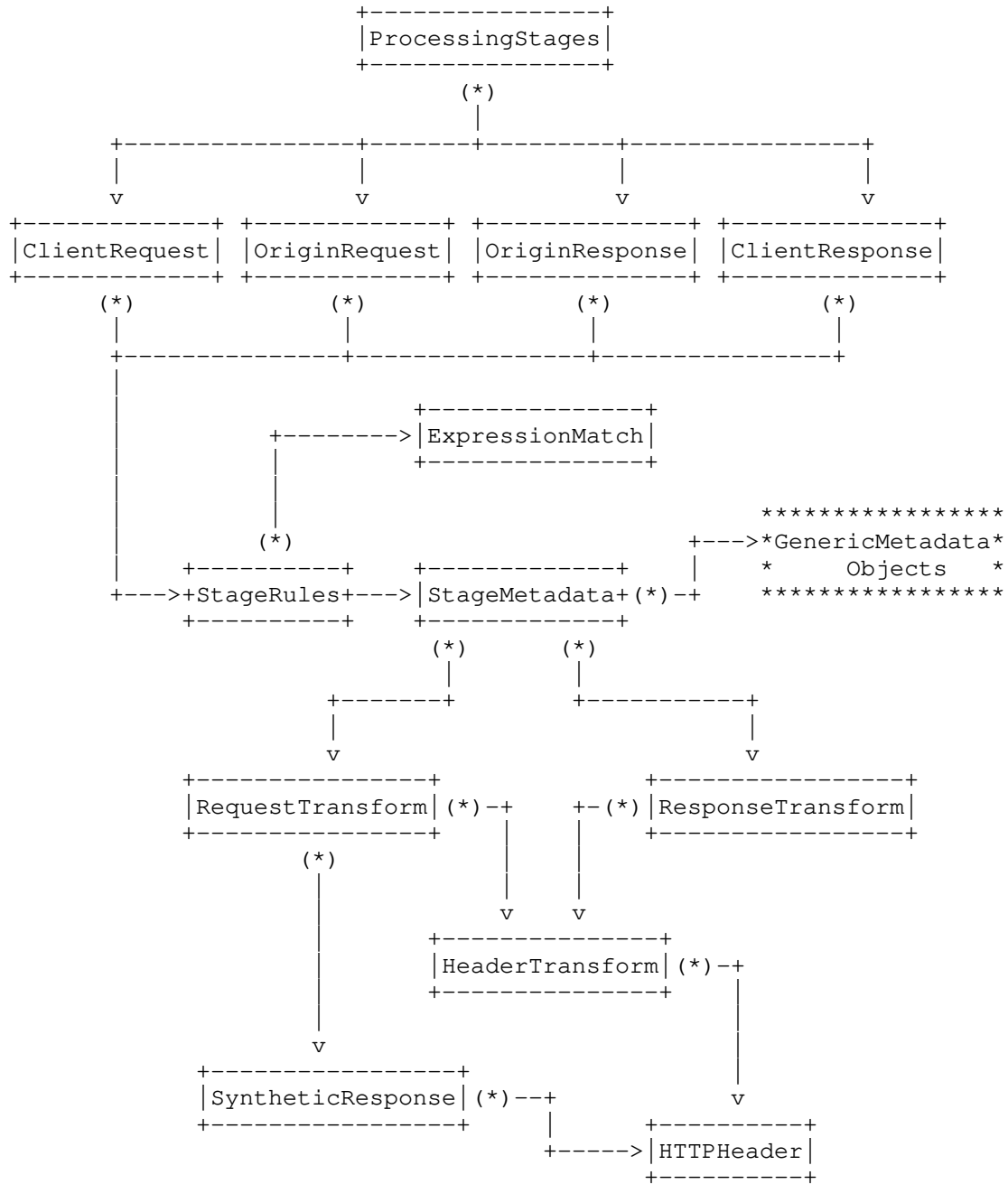


Figure 3: CDNI ProcessingStages metadata model with contained objects

Each of the four processing stages is represented by an array of StageRules objects, with each StageRules object defining criteria along with metadata that MUST be applied if the match applies to "True". It should be noted that the StageRules objects in the array are evaluated and processed in order. A possible future extension to this processing model could allow for an if-else structure, where processing for a stage is halted upon the first match of a StageRules expression.

Property: client-request

- Description: Allows for the specification of conditional metadata to be applied at the client request processing stages, as defined in the Rule Processing Stages section. The StageRules in the array are evaluated in order.
- Type: Array of StageRules objects
- Mandatory-to-Specify: No

Property: origin-request

- Description: Allows for the specification of conditional metadata to be applied at origin request processing stages, as defined in the Rule Processing Stages section. The StageRules in the array are evaluated in order.
- Type: Array of StageRules objects
- Mandatory-to-Specify: No

Property: origin-response

- Description: Allows for the specification of conditional metadata to be applied at origin response processing stages, as defined in the Rule Processing Stages section. The StageRules in the array are evaluated in order.
- Type: Array of StageRules objects
- Mandatory-to-Specify: No

Property: client-response

- Description: Allows for the specification of conditional metadata to be applied at client response processing stages, as defined in the Rule Processing Stages section. The StageRules in the array are evaluated in order.

- Type: Array of StageRules objects
- Mandatory-to-Specify: No

Example specifying all four processing stages. In this example, the client-request stage has two StageRules, applying one set of metadata if "ExpressionMatch1" evaluates to "True" and applying another set of metadata if "ExpressionMatch2" evaluates to "True".

```
{
  "generic-metadata-type": "MI.ProcessingStages",
  "generic-metadata-value": {
    "client-request" : [
      {
        "match" : < ExpressionMatch1 for conditional metadata >
        "stage-metadata" : <StageMetadata1 for clientRequest stage>,
      },
      {
        "match" : <Additional ExpressionMatch2 for conditional metadata >
        "stage-metadata" : <StageMetadata2 for clientRequest stage>,
      }
    ],
    "origin-request" : [{
      "match" : <Optional ExpressionMatch for conditional metadata >
      "stage-metadata" : <StageMetadata for originRequest stage>,
    }],
    "origin-response" : [{
      "match" : <Optional ExpressionMatch for conditional metadata >
      "stage-metadata" : <StageMetadata for originResponse stage>,
    }],
    "client-response" : [{
      "match" : <Optional ExpressionMatch for conditional metadata >
      "stage-metadata" : <StageMetadata for clientResponse stage>,
    }]
  }
}
```

2.4.2. MI.StageRules

A StageRules object is used within the context of ProcessingStages to define elements in a list of match rules and stage-specific metadata and transformations that MUST be applied conditionally on a rich expression match.

Property: match

- Description: An ExpressionMatch object encapsulating a rich expression using the CDNI Metadata Expression Language [CDNI-MEL] to evaluate aspects of the HTTP request and/or response. The stage-metadata rules are only applied if the match evaluates to "True" or if no match expression is provided
- Type: ExpressionMatch object
- Mandatory-to-Specify: No. The stage-metadata rules are always applied if no match expression is provided. This would be the case when stage-metadata should be applied unconditionally within the context of the higher-level host and path matches.

Property: stage-metadata

- Description: Specifies the set of StageMetadata to be applied at the processing stage if the match expression evaluates to "True" or is not present.
- Type: Array of StageMetadata objects, applied in order.
- Mandatory-to-Specify: Yes

An example of StageRules that are applied just after responses are received from the origin. In this example, receipt of a response status code of 304 from the origin indicates that CachePolicy metadata SHOULD be applied (as specified via an external HREF), and that response headers SHOULD be modified (X-custom-response-header added and ETag deleted).

```
{
  "match": {
    "expression": "resp.status == 304"
  },
  "stage-metadata": {
    "generic-metadata": [
      {
        "type": "MI.CachePolicy",
        "href": "https://metadata.ucdn.example/origin_response_cache"
      }
    ],
    "response-transform": {
      "headers": {
        "add": [
          {
            "name": "X-custom-response-header",
            "value": "header-value"
          }
        ],
        "delete": [ "ETag" ]
      }
    }
  }
}
```

2.4.3. MI.ExpressionMatch

The ExpressionMatch object contains the rich expression that must evaluate to "True" for the StageMetadata to be applied for the specific StageRules. Defining expressions as stand-alone objects allows for sets of reusable match expressions to be reused via metadata reference linking.

Property: expression

- Description: A rich expression using CDNI-MEL to evaluate aspects of the HTTP request and/or response. See documentation on the Metadata Expression Language for details on the expression of matching variables and syntax.
- Type: String, using CDNI-MEL syntax. See the METADATA EXPRESSION LANGUAGE (CDNI-MEL) section.
- Mandatory-to-Specify: Yes

Example of ExpressionMatch on the referrer and user agent request headers:

```
{
  "expression" : "req.h.user-agent != '*Safari*' and req.h.referrer == 'www.myhos
t.com'"
}
```

2.4.4. MI.StageMetadata

The StageMetadata object contains GenericMetadata and HTTP request/response transformations that **MUST** be applied for a StageRules match. The following table defines the processing stages where request and response transformations are possible:

Stage	request-transform	response-transform
clientRequest	yes	yes
originRequest	yes	yes
originResponse	no	yes
clientResponse	no	yes

Table 1: StageMetadata stages

Note that for the stages where both request and response transformations are allowed, it is possible to specify both. This may be the case if, for example, the request URI needs alteration for cache-key generation and the response headers need to be manipulated.

Property: generic-metadata

- **Description:** Specifies the set of GenericMetadata to be applied for a StageRules match. A typical use case would be the application of a CachePolicy or TimeWindowACL conditionally on matching HTTP headers. Support for this capability is optional and can be advertised via feature-flags in the FCI interface.
- **Type:** Array of GenericMetadata, applied in order. Note that not all GenericMetadata object types may be applicable at all processing stages.
- **Mandatory-to-Specify:** No. The generic-metadata property would not be needed when StageMetadata is used to only specify request or response transformations, such as modifications of HTTP headers.

Property: request-transform

- Description: Specifies a transformation to be applied to the HTTP request for a StageRules match. The transformation can be the modification of any request header and/or the modification of the URI. Modifications are applied such that downstream processing stages receive the modified HTTP request as their input. Support for this capability is optional and can be advertised via feature-flags in the FCI interface.
- Type: RequestTransform object
- Mandatory-to-Specify: No

Property: response-transform

- Description: Specifies a transformation to be applied to the HTTP response for a StageRules match. The transformation can be the modification of any response header, HTTP response status code, or the generation of a synthetic response. Modifications are applied such that downstream processing stages receive the modified HTTP response as their input. Support for this capability is optional and can be advertised via feature-flags in the FCI interface.
- Type: ResponseTransform object
- Mandatory-to-Specify: No

Example of a StageMetadata object:

```
{
  "generic-metadata" : [{
    < Optional list of generic metadata to apply at this stage >
  }],
  "request-transform" : {
    "headers" : { <list of request headers to add/replace/delete> },
    "uri" : < URI rewrite, either static or dynamically constructed >
  }
  "response-transform" : {
    "headers" : { <list of response headers to add/replace/delete> },
    "response-status" : <Status either static or dynamically constructed >
  }
}
```

2.4.5. MI.RequestTransform

The RequestTransform object contains metadata for transforming the HTTP request for a specific StageRules object. The transformation can be the modification of any request header and/or the modification of the URI. Modifications are applied such that downstream processing stages receive the modified HTTP request as their input.

Property: headers

- Description: A HeaderTransform object specifying HTTP request headers to add, replace, or delete.
- Type: HeaderTransform object
- Mandatory-to-Specify: No

Property: uri

- Description: Replacement value for the HTTP request.
- Type: String. Either a literal (static string) or an expression using CDNI-MEL to dynamically construct a URI value from elements of the HTTP request and/or response.
- Mandatory-to-Specify: No

Property: uri-is-expression

- Description: Flag to signal whether the URI is a static string literal or a CDNI-MEL expression that needs to be dynamically evaluated.
- Type: Boolean
- Mandatory-to-Specify: No. The default is "False", indicating that the URI is a string literal and does not need to be evaluated.

Example of a RequestTransform object illustrating a dynamically constructed URI rewrite:

```
{
  "request-transform" : {
    "headers" : { <Optional list of request headers to add/replace/delete> },
    "uri" : "req.uri.path",
    "uri-is-expression" : true
  }
}
```


2.4.6. MI.ResponseTransform

The ResponseTransform object contains metadata for transforming the HTTP response for a StageRules match. The transformation can be the modification of any response header, HTTP response status code, or the generation of a synthetic response. Modifications are applied such that downstream processing stages receive the modified HTTP response as their input.

Property: headers

- Description: A HeaderTransform object specifying HTTP response headers to add, replace, or delete.
- Type: HeaderTransform object
- Mandatory-to-Specify: No

Property: response-status

- Description: Replacement value for the HTTP response status code.
- Type: Integer. Either a static integer or an expression using CDNI-MEL that evaluates to an integer to dynamically generate an HTTP status code based on elements of the HTTP request and/or response. Expressions that do not evaluate to an integer shall be considered invalid and result in no override of origin-provided response status.
- Mandatory-to-Specify: No

Property: status-is-expression

- Description: Flag to signal whether the response-status is a static integer or a CDNI-MEL expression that needs to be dynamically evaluated to generate an HTTP response status code.
- Type: Boolean
- Mandatory-to-Specify: No. The default is "False", indicating that the response-status is a static integer and does not need to be evaluated.

Property: synthetic

- **Description:** Specification of a complete replacement of any HTTP response that may have been generated in an earlier processing stage with a synthetic response. Use of this property to specify a synthetic response would override any response transformations or status codes specified by other properties.
- **Type:** SyntheticResponse object
- **Mandatory-to-Specify:** No

Example of a ResponseTransform object, illustrating a dynamically constructed header value that uses the expression language to concatenate the user agent and host header, and forces a 403 HTTP response status code:

```
{
  "response-transform": {
    "headers": {
      "add": [
        {
          "name": "X-custom-response-header",
          "value": "req.h.user-agent . &#8216;-&#8216; . req.h.host",
          "value-is-expressions": true
        }
      ]
    },
    "response-status": "403"
  }
}
```

2.4.7. MI.SyntheticResponse

The SyntheticResponse object allows for the specification of a synthetic response to be generated in response to the HTTP request being processed. The synthetic response can contain a set of response headers, a status code, and a response body, and is a complete replacement for any HTTP response elements generated in an earlier processing stage.

A dynamically generated Content-Length HTTP response header is generated based on the length of the generated response body.

Property: headers

- **Description:** An array of HTTP header objects that specify the full set of headers to be applied to the synthetic response.

- Type: Array of HTTP header objects
- Mandatory-to-Specify: No, although it would be unusual to not specify minimal standard response headers, such as Content-Type.

Property: response-status

- Description: HTTP response status code.
- Type: Integer. Either a static integer or an expression using CDNI-MEL that evaluates to an integer to dynamically generate an HTTP status code based on elements of the upstream HTTP request and/or response. Expressions that do not evaluate to an integer shall be considered invalid and result in a 500 status for the synthetic response.
- Mandatory-to-Specify: Yes

Property: status-is-expression

- Description: Flag to signal whether the response-status is a static integer or a CDNI-MEL expression that needs to be dynamically evaluated to generate an HTTP response status code.
- Type: Boolean
- Mandatory-to-Specify: No. The default is "False", indicating that the response-status is a static integer and does not need to be evaluated.

Property: body

- Description: Body for the synthetic HTTP response. The response body can either be static or dynamically constructed from a rich expression.
- Type: String. Either a literal (static string) or an expression using CDNI-MEL to dynamically construct a response body from elements of the HTTP request and/or response.
- Mandatory-to-Specify: No. If absent, an empty HTTP response with a zero-value Content-Length header is generated.

Property: body-is-expression

- Description: Flag to signal whether the synthetic response body is a static string literal or a CDNI-MEL expression that needs to be dynamically evaluated.
- Type: Boolean
- Mandatory-to-Specify: No. The default is "False", indicating that the body is a string literal and does not need to be evaluated.

Example of a SyntheticResponse object illustrating a dynamically constructed response body that uses the expression language to combine the request URI with static text and forces a 405 HTTP response status code:

```
{
  "headers": [
    {
      "name": "content-type",
      "value": "text/plain"
    },
    {
      "name": "X-custom-response-header",
      "value": "some static value"
    }
  ],
  "response-status": "405",
  "response-body": "'Sorry, Access to resource '.req.uri.' not allowed'",
  "body-is-expression": false
}
```

2.4.8. MI.HeaderTransform

The HeaderTransform object specifies how HTTP headers MUST be added, replaced, or deleted from HTTP requests and responses.

Property: add

- Description: List of HTTP headers (name/value pairs) that MUST be added to the HTTP request or response. Note that any existing headers in the request or response with the same names of those added are not affected, resulting in multiple headers with the same name.
- Type: Array of HTTPHeader objects containing header name/value pairs
- Mandatory-to-Specify: No

Property: replace

- Description: List of HTTP headers (name/value pairs) that MUST be added to the HTTP request or response, replacing any previous headers with the same name.
- Type: Array of HTTPHeader objects containing header name/value pairs
- Mandatory-to-Specify: No

Property: delete

- Description: List of names of HTTP headers that MUST be deleted from the
- HTTP request or response. If a named header appears multiple times, all occurrences are deleted.
- Type: Array of strings, with each string naming an HTTP header to delete
- Mandatory-to-Specify: No

Example of a HeaderTransform object illustrating the addition of two customer headers, the replacement of any previously provided Accept-Encoding header, and the removal of any previously provided Authorization or Accept-Language headers:

```
{
  "add": [
    {
      "name": "X-custom-header1",
      "value": "header-value 1"
    },
    {
      "name": "X-custom-header2",
      "value": "header-value 2"
    }
  ],
  "replace": [
    {
      "name": "Accept-Encoding",
      "value": "gzip,deflate,br"
    }
  ],
  "delete": [
    "Authorization",
    "Accept-Language"
  ]
}
```

2.4.9. MI.HTTPHeader

The HTTPHeader object contains a name/value pair for an HTTP header to add or replace in a request or response. The CDNI-MEL expression language can be used to dynamically generate response values.

Property: name

- Description: Name of the HTTP header.
- Type: String
- Mandatory-to-Specify: Yes

Property: value

- Description: New value of the named HTTP header.
- Type: String. Either a static string or an expression using [CDNI-MEL] to dynamically construct a header value from elements of the HTTP request and/or response.
- Mandatory-to-Specify: Yes

Property: value-is-expression

- Description: Flag to signal whether the value is a static string literal or a [CDNI-MEL] expression that needs to be dynamically evaluated.
- Type: Boolean
- Mandatory-to-Specify: No. The default is "False", indicating that the value is a string literal and does not need to be evaluated.

Example of an HTTPHeader illustrating a dynamically constructed header value that equals the session parameter from the query string:

```
{  
  "name": "X-custom-response-header",  
  "value": "req.uri.query.session",  
  "value-is-expression": true  
}
```

2.5. General Metadata

This section documents a set of general purpose GenericMetadata objects whose use and interpretation may be specific to a CDN or Open Caching system's implementation, enabling extensibility and service differentiation for providers.

2.5.1. MI.ServiceIDs

CDN configurations typically have multiple tiers of identifiers that group configurations by customer account to facilitate logging, billing, and support operations. This structure supports two tiers of identifiers (a serviceID which typically identifies a high level customer's service, and a propertyID which typically represents a logical grouping of a set of hosts within a customers' service. It should be noted, however, that the interpretation of ServiceID and PropertyID are implementation-specific, and may not be used by all CDNs and Open Caching systems.

This metadata model extension allows for the association service identifier metadata to a host or path match and to allow for these IDs to be dynamically generated via an expression language. For example, it may be necessary to extract a portion of the Request URI path to derive a service identifier (e.g.: /news/* maps to one propertyID and /movies/* maps to a different propertyID). When processing the MI.ServiceIDs metadata for a request, implementations SHOULD override any previously assigned service identifiers with those specified by this metadata.

MI.ServiceIDs is a new GenericMetadata object that allows for the specification of the two tiers of CDN-specific service identifiers and service names.

Property: service-id

- Description: A provider-specific identifier for the service (typically a customer account identifier).
- Type: String. Either a literal (static string) or an expression using CDNI-MEL to dynamically construct the ID from elements of the HTTP request and/or response.
- Mandatory-to-Specify: No

Property: service-id-is-expression

- Description: Flag to signal whether the service-id is a static string literal or a CDNI-MEL expression that needs to be dynamically evaluated.
- Type: Boolean
- Mandatory-to-Specify: No. The default is "False", indicating that the service-id is a string literal and does not need to be evaluated.

Property: service-name

- Description: Human-readable name for the service-id.
- Type: String
- Mandatory-to-Specify: No

Property: property-id

- Description: A provider-specific identifier for the property (typically identifies a child configuration within the parent service-id).
- Type: String. Either a literal (static string) or an expression using CDNI-MEL to dynamically construct the ID from elements of the HTTP request and/or response.
- Mandatory-to-Specify: No

Property: property-id-is-expression

- Description: Flag to signal whether the property-id is a static string literal or a CDNI-MEL expression that needs to be dynamically evaluated.
- Type: Boolean
- Mandatory-to-Specify: No. The default is "False", indicating that the property-id is a string literal and does not need to be evaluated.

Property: property-name

- Description: Human-readable name for the property-id.
- Type: String
- Mandatory-to-Specify: No

Example illustrating the assignment of a literal service-id along with a dynamically computed property-id that is extracted from the root element of the request URI path.

```
{
  "generic-metadata-type": "MI.ServiceIDs",
  "generic-metadata-value": {
    "service-id": "12345",
    "service-name": "My Streaming Service",
    "property-id": "path_element(req.uri, 1)",
    "property-id-is-expression": true
  }
}
```

2.5.2. MI.PrivateFeatureList

The dCDN may gather a certain number of private features (i.e., not [yet] adopted by SVA or considered marginal) that it may want to expose to the content provider and/or the uCDN. Although private, the announcement, selection, and configuration of this private feature could be done through the OC API.

One example could be the support in OCNs of a new protocol that allows the ability to get additional insight about the user agent status (e.g., CTA Wave CMCD).

As another example, Broadpeak has developed a feature called S4Streaming, and would like to give the opportunity to control that feature to the uCDN.

PrivateFeatureList is a GenericMetadata configuration object as a base generic object that permits the control of private features.

Property: features

- Description: The list of feature configuration objects.
- Type: List (array) of MI.PrivateFeature objects .
- Mandatory-to-Specify: Yes

2.5.2.1. MI.PrivateFeature

MI.PrivateFeature object contains the following properties:

Property: feature-oid

- Description: The owner/organization that has specified that feature.
- Type: String
- Mandatory-to-Specify: Yes

Property: feature-type

- Description: Indicates the type/name of the private feature configuration object.
- Type: String
- Mandatory-to-Specify: Yes

Property: feature-value

- Description: Feature configuration object.
- Type: Format/type is defined by the value of the feature-type property above.
- Mandatory-to-Specify: Yes

Note that the private features exposed by the dCDN can be advertised through a dedicated FCI object.

Example, illustrating the Broadpeak S4 Streaming feature:

```
{
  "generic-metadata-type": "MI.PrivateFeatureList",
  "generic-metadata-value": {
    "feature": {
      "feature-oid": "Broadpeak",
      "feature-type": "S4Streaming",
      "feature-value": {
        "footprint": {
          "footprint-type": "ipv4cidr",
          "footprint-value": [
            "192.0.2.0/24",
            "198.51.100.0/24"
          ]
        },
        "activation": "ON",
        "mode": "transparent",
        "policy": "bandwidth-max"
      }
    }
  }
}
```

2.5.3. MI.RequestRouting

The uCDN requires the ability to indicate whether HTTP redirect, DNS redirect, and manifest rewrite are allowed, and indicate which is preferable. This is required in cases where the uCDN would like to delegate the traffic relying on the iterative method but knows the client will not support HTTP redirect. In that case, the uCDN needs a means to force the dCDN to perform request routing based on DNS redirect (or manifest rewrite).

This configuration possibility is useful only if the dCDN can advertise the mode of redirection it supports. There is an ongoing discussion in the IETF CDNI group to understand the semantics behind the redirection modes currently in Footprint & Capabilities Advertising Interface (I-DNS and I-HTTP). It is not clear whether this indicates that the dCDN supports one or both delegation modes (the request routing performed by the uCDN can only be based on DNS redirect or HTTP redirect or both), or whether it indicates that the dCDN supports, as its own request routing mode, DNS redirect and/or HTTP redirect. The latter is required for this new configuration object to be valid.

MI.RequestRouting is a new GenericMetadata object that allows the uCDN to force the dCDN request routing mode(s) to be applied when working in iterative redirection mode. The list of redirection modes supported by the dCDN is advertised through the FCI.RedirectionMode object. The list of request routing modes supported by the dCDN is advertised through the FCI.RequestRoutingMode object.

Property: request-routing-modes

- Description: Instructs the dCDN to perform request routing according to one or more preferred modes among those supported and advertised by the dCDN through the FCI.RequestRouting object. One must understand that forcing (instead of letting the dCDN request router select) one particular request routing mode may trigger some inefficiency in the request routing process.
- Type: List (array) of iterative request routing modes
- Values: "DNS", "HTTP", "MANIFEST_REWRITE"
- Mandatory-to-Specify: No. By default, all request routing modes supported by the dCDN can be used by the dCDN as part of its request routing process.

Example, illustrating the uCDN forcing the dCDN to use DNS or HTTP as the method for request routing in case the uCDN performs an iterative delegation (i.e., iterative redirection mode):

```
{
  "generic-metadata-type": "MI.RequestRouting",
  "generic-metadata-value": {
    "request-routing-modes": [ "DNS", "HTTP" ]
  }
}
```

3. Metadata Expression Language

The CDNI Metadata Expression Language provides a syntax with a rich set of variables, operators, and built-in functions to facilitate use cases within the extended CDNI metadata model.

Enables expression matching to dynamically determine if StageMetadata (Section 2.4) should be applied at a StageRules match.

Enables the dynamic construction of a value to be used in scenarios such as constructing a service identifier or cache key, setting an HTTP header, rewriting a request URI, setting a response status code, or dynamically generating a response body for a SyntheticResponse.

Expressions can evaluate to a Boolean, string, or integer, depending on the use case:

Usage	Description	Evaluation Results
ExpressionMatch.expression	Dynamically determines if StageMetadata should be applied at a specific StageRules.	Boolean. Expressions that do not evaluate to True or False shall be considered as False.
RequestTransform.uri	Rewrites request URI that will be presented to all downstream stages.	String
ResponseTransform.response-status	Dynamically sets a response status code to replace the status-code returned by the origin.	Integer (HTTP status code)
SyntheticResponse.response-status	Dynamically sets a response status code for a synthetically constructed response.	Integer (HTTP status code)
SyntheticResponse.body	Dynamically constructs a response body.	String
HTTPHeader.value	Dynamically	String

	constructs a header value.	
ComputedCacheKey.expression	Dynamically constructs a cache key.	String
ServiceIDs.property-id, ServiceIDs.service-id	Dynamically constructs service and property identifiers.	String

Table 2: CDNI MEL expressions

3.1. Expression Variables

Variable	Meaning
req.h.<name>	Request header <name>
req.uri	Request URI (includes query string and fragment identifier, if any)
req.uri.path	Request URI path
req.uri.pathquery	Request path and query string
req.uri.query	Request query string
req.uri.query.<key>	Request query string value associated with <key>
req.method	Request HTTP method (GET, POST, others)
resp.h.<name>	Response header <name>
resp.status	Response status code

Table 3: CDNI MEL variables

3.2. Expression Operators and keywords

Operator	Type	Result Type	Meaning
==	infix	Boolean	Equality test
!=	infix	Boolean	Inequality test
!	infix	Boolean	Logical NOT operator
>	infix	Boolean	Greater than test
<	infix	Boolean	Less than test
>=	infix	Boolean	Greater than or equal test
<=	infix	Boolean	Less than or equal
*=	infix	Boolean	Glob style match
~=	infix	Boolean	Regular expression match (see https://www.pcre.org/ for details on PCRE RegEx matching)
ipmatch	infix	Boolean	Match against IP address or CIDR (IPv4 and IPv6)
+	infix	Numeric	Addition
-	infix	Numeric	Subtraction
*	infix	Numeric	Multiplication
/	infix	Numeric	Division
%	infix	Unsigned or Integer	Modulus
.	infix	String	Concatenation
? :	ternary	*	Conditional operator: <e> ? <v1> : <v2> Evaluates <v1> if <e> is true, <v2> otherwise.
()	grouping		Used to override precedence and

			for function calls.	
--	--	--	---------------------	--

Table 4: CDNI MEL expression operators

Keyword	Meaning
and	Logical AND
or	Logical OR
not	Logical NOT (see also the ! operator)
nil	No value (distinct from empty value)
true	Boolean constant: true
false	Boolean constant: false

Table 5: CDNI MEL expression keywords

3.3. Expression Built-in Functions

3.3.1. Basic Functions: Type Conversions

Function	Action	Argument(s)	Returns
integer(e)	Converts expression to integer.	1	integer
real(e)	Converts expression to real.	1	real
string(e)	Converts expression to string.	1	string
boolean(e)	Converts expression to Boolean.	1	Boolean

Table 6: CDNI MEL type conversions

3.3.2. Basic Functions: String Conversions

Function	Action	Argument (s)	Returns
upper(e)	Converts a string to uppercase. Useful for case-insensitive comparisons.	1	string
lower(e)	Converts a string to lowercase. Useful for case-insensitive comparisons.	1	string

Table 7: CDNI MEL string conversions

3.3.3. Convenience Functions

Function	Action	Argument (s)	Returns
match(string Input, string Match)	Regular expression Match is applied to Input and the matching element (if any) is returned. Empty string is returned if there is no match. See https://www.pcre.org/ for details on PCRE RegEx matching.	2	string
match_replace(string Input, string Match, string Replace)	Regular expression Match is applied to Input arg and replaced with the Replace arg upon successful match. Returns updated (replaced) version of Input.	3	string
add_query(string Input, string q, string v)	Add query string element q with value v to the Input string. If v is nil, then just add the query string element q. The query element q and value v	2	string

--+	path_element(string	Return the path elements from position	3	string
g	Input, integer n,	n to m.		
	integer m)			
--+				

Table 8: CDNI MEL convenience functions

3.4. Error Handling

3.4.1. Compile Time Errors

To ensure reliable service, all CDNI Metadata configurations MUST be validated for syntax errors before they are ingested into a dCDN. That is, existing configurations should be kept as the live running configuration until the new configuration has passed validation. If errors are detected in a new configuration, the configuration MUST be rejected. A HTTP 500 Internal Server Error should be returned with a message that indicates the source of the error (line number, and configuration element that caused the error).

Examples of compile-time errors:

1. Configuration does not parse relative to the CDNI Metadata JSON schema
2. Unknown CDNI Metadata object referenced in the configuration
3. CDNI Metadata object parse error
 - a. Missing mandatory CDNI Metadata property
 - b. Unknown CDNI Metadata property
 - c. Incorrect type for a CDNI Metadata property value
4. CDNI-MEL
 - a. Unknown CDNI-MEL variable name referenced in an expression
 - b. Unknown CDNI-MEL operator, key-word, or functions referenced in an expression
 - c. Incorrect number of arguments used in a CDNI-MEL expression operator or function
 - d. Incorrect type of argument used in a CDNI-MEL expression operator or function

3.4.2. Runtime Errors

If a runtime error is detected when processing a request, the request should be terminated, and a HTTP 500 'Internal Server Error' returned to the caller. To avoid security leaks, sensitive information **MUST** be removed from the error message before it is returned to an external client. In addition to returning the HTTP 500 error, the dCDN **SHOULD** log additional diagnostics information to assist in troubleshooting.

Examples of runtime errors:

1. Failure to allocate memory (or other server resources) when evaluating a CDNI-MEL expression
2. Incorrect runtime argument type in a CDNI-MEL expression. E.g., trying to convert a non-numeric string to a number

3.5. Expression Examples

3.5.1. ComputedCacheKey

Sets the MI.ComputedCacheKey to the value of the X-Cache-Key header from the client request.

```
{
  "generic-metadata-type": "MI.ComputedCacheKey",
  "generic-metadata-value": {
    "expression": "req.h.x-cache-key"
  }
}
```

Sets the MI.ComputedCacheKey to the lowercase version of the URI.

```
{
  "generic-metadata-type": "MI.ComputedCacheKey",
  "generic-metadata-value": {
    "expression": "lower(req.uri)"
  }
}
```

3.5.2. ExpressionMatch

ExpressionMatch where the expression is true if the user-agent (glob) matches *Safari* and the referrer equals www.example.com.

```
{
  "expression": "req.h.user-agent *= '*Safari*'
    and req.h.referrer == 'www.example.com'"
}
```

3.5.3. ResponseTransform

Adds X-custom-response-header with a value equal to the value of user-agent - host header.

```
{
  "response-transform": {
    "headers": {
      "add": [
        {
          "name": "X-custom-response-header",
          "value": "req.h.user-agent . ' - ' . req.h.host",
          "value-is-expression": true
        }
      ],
      "response-status": "403"
    }
  }
}
```

Adds a Set-Cookie header with a dynamically computed cookie value (concatenating user agent and host name) and forces a 403 response.

```
{
  "response-transform": {
    "headers": {
      "add": [
        {
          "name": "Set-Cookie",
          "value": "req.h.user-agent . ' - ' . req.h.host",
          "value-is-expression": true
        }
      ]
    }
  }
}
```

3.5.4. MI.ServiceIDs

Extracts the first path element from the URI. For example, if the URI = /789/second/third/test.txt, property-id is set to the first-path (789).

```
{
  "generic-metadata-type":"MI.ServiceIDs",
  "generic-metadata-value":{
    "service-id":"12345",
    "service-name":"My Streaming Service",
    "property-id":"path_element(req.uri, 1)",
    "property-id-is-expression":true
  }
}
```

4. CDNI Capabilities Extensions

Since not all dCDNs will be capable of supporting all the extensions proposed in this document, they need the ability to inform uCDNs about their capabilities. [RFC8008] (the CDNI Footprint & Capabilities Interface) was designed for this purpose and is extended here to express these new capabilities.

4.1. FCI Metadata Object

Whenever a capability is represented as a top-level GenericMetadata object, a dCDN will be able to declare its support simply by including that object name in the capability-value list of the standard FCI.Metadata object.

For each of the new GenericMetadata objects documented within the SVA Configuration Interface, the default assumption should be that the capability is not supported by the dCDN unless named within the FCI metadata object.

Example: A capabilities object declaring support for several of the newly defined GenericMetadata types:

```

{
  "capabilities": [
    {
      "capability-type": "FCI.Metadata",
      "capability-value": {
        "metadata": [
          "MI.SourceMetadataExtended",
          "MI.ProcessingStages",
          "MI.CrossoriginPolicy",
          "MI.CachePolicy",
          "MI.NegativeCachePolicy",
          "MI.PrivateFeatureList",
          "MI.RequestRouting"
        ]
      },
      "footprints": [
        < Footprint Objects >
      ]
    }
  ]
}

```

4.2. FCI Model Extensions

In most cases, the presence or absence of a `GenericMetadata` object name in `FCI.Metadata` (as described above), is sufficient to convey support for a capability. There are cases, however, where more fine-grained capabilities declarations are required. Specifically, a dCDN may support some, but not all, of the capabilities specified by one of the new `GenericMetadata` objects. In these cases, new FCI objects will be created to allow a dCDN to express these fine-grained capabilities.

4.2.1. FCI.AuthTypes

The `AuthTypes` object is used to indicate the support of authentication methods to be used for content acquisition (while interacting with an origin server) and authorization methods to be used for content delivery.

This specification document defines two new authentication methods (see `MI.Auth`) while there is one authorization method currently under specification in CDNI called `[URI.signing]`

Property: stage-metadata

- Description: Specifies the set of StageMetadata to be applied at the processing stage if the match expression evaluates to "True" or is not present.
- Type: Array of StageMetadata objects, applied in order.
- Mandatory-to-Specify: Yes

Property: authe-types

- Description: List of supported authentication methods (possibly required for content acquisition)
- Type: Array of strings
- Values: "AWSv4Auth", "HeaderAuth"
- Mandatory-to-Specify: No. No authentication method is supported in this case.

Property: autho-types

- Description: List of supported authorization methods (possibly required for content delivery)
- Type: Array of strings
- Values: "UriSigning"
- Mandatory-to-Specify: No. No authorization method is supported in this case.

FCI.AuthTypes example:


```
{
  "capabilities": [
    {
      "capability-type": "FCI.AuthTypes",
      "capability-value": {
        "authe-types": [
          "AWSv4Auth",
          "HeaderAuth"
        ],
        "autho-types": [
          "UriSigning"
        ]
      }
    }
  ]
}
```

4.2.2. FCI.ProcessingStages

This object is used to signal the set of features that are supported in relation to the ProcessingStages configuration object (see MI.ProcessingStages). Those optional features depend on the CDNI-MEL language support.

Property: features

- Description: List of supported optional processing stages features. Note that these features all have some dependencies on support of the CDNI MEL expression language.
- Type: Array of strings
- Values: "ExpressionMatch", "RequestTransform", "ResponseTransform"
- Mandatory-to-Specify: No. None of these optional features are supported in this case.

Example:

```
{
  "capabilities": [
    {
      "capability-type": "FCI.ProcessingStages",
      "capability-value": {
        "features": [
          "ExpressionMatch",
          "RequestTransform",
          "ResponseTransform"
        ]
      }
    }
  ]
}
```

4.2.3. FCI.SourceMetadataExtended

This object is used to signal the supported features related to the SourceMetadataExtended configuration object.

Property: load-balance

- Description: List of supported load balancing algorithms in relation to the SourceMetadataExtended configuration object (see MI.SourceMetadataExtended)
- Type: Array of strings
- Values: "random", "content-hash", "ip-hash
- Mandatory-to-Specify: No. load balancing is not supported among sources.

If the FCI.SourceMetadtaExtended object is not exposed/advertised or if the "load-balance" array is empty, the dCDN does not support the usage of the load-balance property attached to the SourceMetadataExtended configuration object (see MI.SourceMetadataExtended).

Example:

```
{
  "capabilities": [
    {
      "capability-type": "FCI.SourceMetadataExtended",
      "capability-value": {
        "load-balance": [
          "random",
          "content-hash",
          "ip-hash"
        ]
      }
    }
  ]
}
```

4.2.4. FCI.RequestRouting

This object is used by the dCDN to signal/announce the supported request routing modes. This can be optionally used by the uCDN to further select a subset of those modes when operating one of the iterative delegation modes. See the section on the GenericMetadata RequestRouting object..

Property: request-routing-modes

- Description: List of supported request routing modes by the dCDN. This information is useful when the uCDN decides to perform a delegation in iterative mode.
- Type: Array of strings
- Values: "DNS", "HTTP-R", "MANIFEST_REWRITE"
- Mandatory-to-Specify: No. If the dCDN does not advertise the supported request routing modes, they are all supported by default.

Example:

```
{
  "capabilities": [
    {
      "capability-type": "FCI.RequestRouting",
      "capability-value": {
        "request-routing-modes": [
          "DNS",
          "HTTP",
          "MANIFEST_REWRITE"
        ]
      }
    }
  ]
}
```

4.2.5. FCI.PrivateFeatures

This object is used by the dCDN to signal/announce the list of supported private features. See the section on the GenericMetadata PrivateFeatureList object.

Property: features

- Description: The list of supported private feature
- Type: List nested objects of FCI.PrivateFeature

Example:

```
{
  "capabilities": [
    {
      "capability-type": "FCI.PrivateFeatures",
      "capability-value": {
        "features": [
          {
            "feature-oid": "Broadpeak",
            "feature-type": "S4Streaming"
          }
        ]
      }
    }
  ]
}
```

4.2.5.1. FCI.PrivateFeature

This object contains the following properties:

Property: feature-oid

- Description: The owner/organization that has specified the feature.
- Type: String
- Mandatory-to-Specify: Yes

Property: feature-type

- Description: Indicates the type/name of the private feature configuration object.
- Type: String
- Mandatory-to-Specify: Yes

4.2.6. FCI.OcnSelection

This object is used by the dCDN to signal/announce the supported OCN types and/or their transport arrangement and/or medium supported by OCNs.

Property ocn-delivery-list

1. Description: List of supported medium and/or transport arrangements.
2. Type: Array of nested objects, each containing the following properties:
 - o Property: ocn-medium
 - Description: This property lists the supported mediums.
 - Type: Array of strings. The following values are specified: "SATELLITE"
 - Mandatory-to-Specify: No
 - o Property: ocn-transport

- Description: Instructs the dCDN to perform delegation operating a particular transport arrangement. The following values are specified: "MABR".
- Type: Array of strings
- Mandatory-to-Specify: No
-Property: ocn-type-list

o Description: List of supported OCN types. Examples include: "HOME" or "EDGE".

o Type: Array of strings

o Mandatory-to-Specify: No

5. IANA Considerations

5.1. CDNI Payload Types

This document requests the registration of the following entries under the "CDNI Payload Types" registry hosted by IANA

Payload type	Specification
MI.CachePolicy	RFCthis
MI.NegativeCachePolicy	RFCthis
MI.StaleContentCachePolicy	RFCthis
MI.CacheBypassPolicy	RFCthis
MI.ComputedCacheKey	RFCthis
MI.AllowCompress	RFCthis
MI.SourceMetadataExtended	RFCthis
MI.SourceExtended	RFCthis
MI.LoadBalanceMetadata	RFCthis
MI.HeaderAuth	RFCthis
MI.AWSSv4Auth	RFCthis

MI.CrossOriginPolicy	RFCthis
MI.AuthTokenMetadata (TBD)	RFCthis
MI.CertificateMetadata (TBD)	RFCthis
MI.OcnSelection	RFCthis
MI.RequestRouting	RFCthis
MI.ProcessingStages	RFCthis
MI.StageRules	RFCthis
MI.ExpressionMatch	RFCthis
MI.StageMetadata	RFCthis
MI.RequestTransform	RFCthis
MI.ResponseTransform	RFCthis
MI.SyntheticResponse	RFCthis
MI.HeaderTransform	RFCthis
MI.HTTPHeader	RFCthis
MI.ServiceIDs	RFCthis
MI.TrafficType	RFCthis
MI.LoggingMetadata (TBD)	RFCthis
MI.PrivateFeatureList	RFCthis
FCI.AuthTypes	RFCthis
FCI.ProcessingStages	RFCthis
FCI.SourceMetadataExtended	RFCthis
FCI.RequestRouting	RFCthis
FCI.PrivateFeatures	RFCthis
FCI.OcnSelection	RFCthis

+-----+-----+

Table 9: Payload Types

6. Security Considerations

This specification is in accordance with the CDNI Request Routing: Footprint and Capabilities Semantics. As such, it is subject to the security and privacy considerations as defined in Section 8 of [RFC8006] and in Section 7 of [RFC8008] respectively.

7. Conclusion

This document presents requirements and extensions to the CDNI metadata model to cover typical use cases found in the commercial CDN and Open Caching ecosystems. By limiting the scope of these extensions to new GenericMetadata objects, backward compatibility can be maintained with any existing CDNI Metadata Interface implementations.

8. References

8.1. Normative References

- [RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <<https://www.rfc-editor.org/info/rfc1034>>.
- [RFC1123] Braden, R., Ed., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, DOI 10.17487/RFC1123, October 1989, <<https://www.rfc-editor.org/info/rfc1123>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<https://www.rfc-editor.org/info/rfc7231>>.
- [RFC8006] Niven-Jenkins, B., Murray, R., Caulfield, M., and K. Ma, "Content Delivery Network Interconnection (CDNI) Metadata", RFC 8006, DOI 10.17487/RFC8006, December 2016, <<https://www.rfc-editor.org/info/rfc8006>>.

- [RFC8007] Murray, R. and B. Niven-Jenkins, "Content Delivery Network Interconnection (CDNI) Control Interface / Triggers", RFC 8007, DOI 10.17487/RFC8007, December 2016, <<https://www.rfc-editor.org/info/rfc8007>>.
- [RFC8008] Seedorf, J., Peterson, J., Previdi, S., van Brandenburg, R., and K. Ma, "Content Delivery Network Interconnection (CDNI) Request Routing: Footprint and Capabilities Semantics", RFC 8008, DOI 10.17487/RFC8008, December 2016, <<https://www.rfc-editor.org/info/rfc8008>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8804] Finkelman, O. and S. Mishra, "Content Delivery Network Interconnection (CDNI) Request Routing Extensions", RFC 8804, DOI 10.17487/RFC8804, September 2020, <<https://www.rfc-editor.org/info/rfc8804>>.
- [URI.signing] van Brandenburg, R., Leung, K., and P. Sorber, "URI Signing for CDN Interconnection (CDNI)", 8 October 2019, <<http://www.ietf.org/internet-drafts/draft-ietf-cdni-uri-signing-19.txt>>.
- [W3C] "Cross-Origin Resource Sharing", <<https://www.w3.org/TR/2020/SPSD-cors-20200602/>>.

8.2. Informative References

- [AWSv4Method] "Authenticating Requests (AWS Signature Version 4)", <<https://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html>>.
- [OC-CI] Goldstein, G., Ed., Power, W., Bichot, G., and A. Sioniz, "Open Caching - Configuration Interface Functional Specification (Parts 1,2,3)", Version 0.1, 2 July 2021.
- [RFC5861] Nottingham, M., "HTTP Cache-Control Extensions for Stale Content", RFC 5861, DOI 10.17487/RFC5861, May 2010, <<https://www.rfc-editor.org/info/rfc5861>>.
- [RFC6707] Niven-Jenkins, B., Le Faucheur, F., and N. Bitar, "Content Distribution Network Interconnection (CDNI) Problem Statement", RFC 6707, DOI 10.17487/RFC6707, September 2012, <<https://www.rfc-editor.org/info/rfc6707>>.

- [RFC7336] Peterson, L., Davie, B., and R. van Brandenburg, Ed.,
"Framework for Content Distribution Network
Interconnection (CDNI)", RFC 7336, DOI 10.17487/RFC7336,
August 2014, <<https://www.rfc-editor.org/info/rfc7336>>.
- [RFC7694] Reschke, J., "Hypertext Transfer Protocol (HTTP) Client-
Initiated Content-Encoding", RFC 7694,
DOI 10.17487/RFC7694, November 2015,
<<https://www.rfc-editor.org/info/rfc7694>>.
- [SVA] "Streaming Video Alliance Home Page",
<<https://www.streamingvideoalliance.org>>.

Authors' Addresses

Glenn Goldstein
Lumen Technologies
United States of America
Email: glenn1215@gmail.com

Will Power
Lumen Technologies
United States of America
Email: wrpower@gmail.com

Guillaume Bichot
Broadpeak
France
Email: guillaume.bichot@gmail.com

Alfonso Siloniz
Telefonica
Spain
Email: alfonsosiloniz@gmail.com

CDNI Working Group
Internet-Draft
Intended status: Standards Track
Expires: 8 September 2022

F. Fieau, Ed.
E. Stephan
Orange
S. Mishra
Verizon
7 March 2022

CDNI extensions for HTTPS delegation
draft-ietf-cdni-interfaces-https-delegation-08

Abstract

The delivery of content over HTTPS involving one or more CDNs raises credential management issues. This document defines new CDNI FCI and Metadata objects to support HTTPS delegation, especially the ACME-STAR [RFC9115] method.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	2
2. Terminology	2
3. Delegation metadata for CDNI FCI	3
4. Delegation metadata for CDNI	3
4.1. Usage example related to an HostMatch object	3
4.2. AcmeStarDelegationMethod object	4
5. IANA considerations	5
5.1. CDNI MI AcmeStarDelegationMethod Payload Type	6
5.2. CDNI FCI SupportedDelegationMethods Payload Type	6
6. Security considerations	6
7. Privacy considerations	7
8. References	7
8.1. Normative References	7
8.2. Informative References	7
Authors' Addresses	7

1. Introduction

Content delivery over HTTPS using one or more CDNs along the path requires credential management. This specifically applies when an entity delegates delivery of encrypted content to another trusted entity.

The ACME WG has published ACME STAR [RFC9115] allowing a dCDN to request a x.509 certificate from uCDN.

This document proposes the CDNI Metadata interface to setup HTTPS delegation between an upstream CDN (uCDN) and downstream CDN (dCDN) using the ACME STAR proposal. Furthermore, it includes a proposal of IANA registry to enable adding of new methods.

Section 2 is about terminology used in this document. Section 3 presents delegation methods specified at the IETF. Section 4 addresses the extension for handling HTTPS delegation in CDNI. Section 5 describes simple data types. Section 6 addresses IANA registry for delegation methods. Section 7 covers the security issues. Section 8 is about comments and questions.

2. Terminology

This document uses terminology from CDNI framework documents such as: CDNI framework document [RFC7336], CDNI requirements [RFC7337] and CDNI interface specifications documents: CDNI Metadata interface [RFC8006] and CDNI Control interface / Triggers [RFC8007].

3. Delegation metadata for CDNI FCI

The Footprint and Capabilities interface as defined in RFC8008, allows a dCDN to send a FCI capability type object to a uCDN. This draft adds an object named FCI.SupportedDelegationMethods.

This object will allow a dCDN to advertise the capabilities regarding the supported delegation methods and their configuration.

The following is an example of the supported delegated methods capability object for a CDN supporting STAR delegation method.

```
{
  "capabilities": [
    {
      "capability-type": "FCI.SupportedDelegationMethods",
      "capability-value": {
        "delegation-methods": [
          "AcmeStarDelegationDelegationMethod",
          "... Other delegation methods ..."
        ]
      }
    }
  ]
}
```

4. Delegation metadata for CDNI

This section defines Delegation metadata using the current Metadata interface model. This allows bootstrapping delegation methods between a uCDN and a delegate dCDN.

4.1. Usage example related to an HostMatch object

This section presents the use of CDNI Delegation metadata of an HostMatch object, as defined in [RFC8006] as specified in the following sections.

The existence of the delegation methods in metadata in a CDNI Object shall enable the use of one of this methods, chosen by the delegating entity. In the case of an HostMatch object, the delegation method will be activated for the set of Host defined in the HostMatch. See Section 4.2 for more details about delegation methods metadata specification.

The HostMatch object can reference a host metadata that points at the delegation information. Delegation metadata are added to a Metadata object.

Below shows both HostMatch its Metadata related to a host, for example, here is a HostMatch object referencing "video.example.com":

HostMatch:

```
{
  "host": "video.example.com",
  "host-metadata": {
    "type": "MI.HostMetadata",
    "href": "https://metadata.ucdn.example/host1234"
  }
}
```

Following the example above, the metadata can be modeled for ACMEStarDelegationMethod as:

```
"generic-metadata-value": {
  "acme-delegations": [
    "https://acme.ucdn.example/acme/delegation/ogfr8Ecol0T",
    "https://acme.ucdn.example/acme/delegation/wSi5Lbb6lE4"
  ]
}
```

This extension allows to explicitly indicate support for a given method. Therefore, the presence (or lack thereof) of an AcmeStarDelegationMethod, and/or further delegation methods, implies support (or lack thereof) for the given method.

Those metadata can apply to other MI objects such as PathMatch object metadata.

4.2. AcmeStarDelegationMethod object

This section defines the AcmeStarDelegationMethod object which describes metadata related to the use of ACME/STAR API presented in [RFC9115]

As expressed in [RFC9115], when an origin has set a delegation to a specific domain (i.e. dCDN), the dCDN should present to the end-user client, a short-term certificate bound to the master certificate.

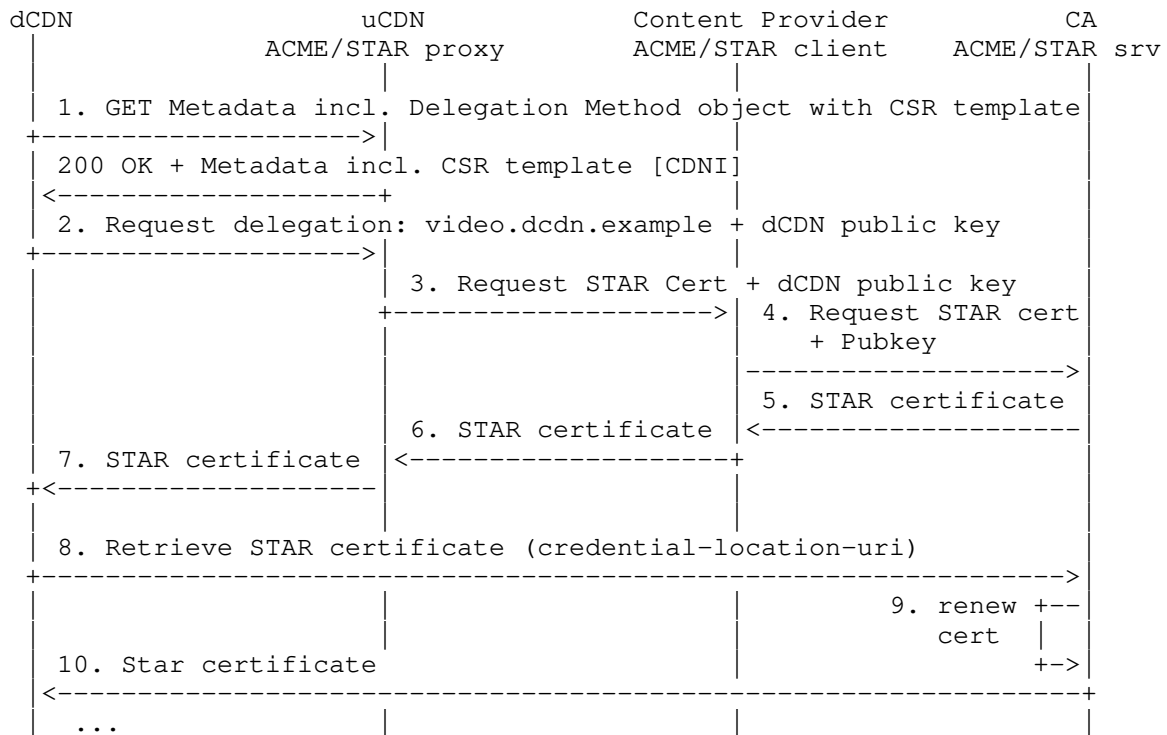


Figure 1: Example call-flow of STAR delegation in CDNI showing 2 levels of delegation

Property: acme-delegations

Description: an array of delegation objects associated with the dCDN account on the uCDN ACME server (see Section 2.3.1 of [RFC9115] for the details).

Type: Objects

Mandatory-to-Specify: Yes

5. IANA considerations

This document requests the registration of the following entries under the "CDNI Payload Types" registry hosted by IANA regarding "CDNI delegation":

Payload Type	Specification
MI.AcmeStarDelegationMethod	RFCthis
FCI.SupportedDelegationMethods	RFCthis

[RFC Editor: Please replace RFCthis with the published RFC number for this document.]

5.1. CDNI MI AcmeStarDelegationMethod Payload Type

Purpose: The purpose of this Payload Type is to distinguish AcmeStarDelegationMethod MI objects (and any associated capability advertisement)

Interface: MI

Encoding: see Section 5

5.2. CDNI FCI SupportedDelegationMethods Payload Type

Purpose: The purpose of this Payload Type is to distinguish SupportedDelegationMethods FCI objects (and any associated capability advertisement)

Interface: FCI

Encoding: see Section 4

6. Security considerations

Extensions proposed here do not alter nor change Security Considerations as outlined in the CDNI Metadata and Footprint and Capabilities RFCs [RFC8006].

However there are still some security questions that should be addressed such as: Are there concerns about using this incorrectly or limitations on how this can safely be used?

7. Privacy considerations

Some privacy questions are still pending: Are there any concerns with sharing the information that is in the metadata? Is the metadata safe to redistribute, or is it something that is only valid between adjacent CDNs?

8. References

8.1. Normative References

- [RFC8006] Niven-Jenkins, B., Murray, R., Caulfield, M., and K. Ma, "Content Delivery Network Interconnection (CDNI) Metadata", RFC 8006, DOI 10.17487/RFC8006, December 2016, <<https://www.rfc-editor.org/info/rfc8006>>.
- [RFC8007] Murray, R. and B. Niven-Jenkins, "Content Delivery Network Interconnection (CDNI) Control Interface / Triggers", RFC 8007, DOI 10.17487/RFC8007, December 2016, <<https://www.rfc-editor.org/info/rfc8007>>.
- [RFC8739] Sheffer, Y., Lopez, D., Gonzalez de Dios, O., Pastor Perales, A., and T. Fossati, "Support for Short-Term, Automatically Renewed (STAR) Certificates in the Automated Certificate Management Environment (ACME)", RFC 8739, DOI 10.17487/RFC8739, March 2020, <<https://www.rfc-editor.org/info/rfc8739>>.
- [RFC9115] Sheffer, Y., López, D., Pastor Perales, A., and T. Fossati, "An Automatic Certificate Management Environment (ACME) Profile for Generating Delegated Certificates", RFC 9115, DOI 10.17487/RFC9115, September 2021, <<https://www.rfc-editor.org/info/rfc9115>>.

8.2. Informative References

- [RFC7336] Peterson, L., Davie, B., and R. van Brandenburg, Ed., "Framework for Content Distribution Network Interconnection (CDNI)", RFC 7336, DOI 10.17487/RFC7336, August 2014, <<https://www.rfc-editor.org/info/rfc7336>>.
- [RFC7337] Leung, K., Ed. and Y. Lee, Ed., "Content Distribution Network Interconnection (CDNI) Requirements", RFC 7337, DOI 10.17487/RFC7337, August 2014, <<https://www.rfc-editor.org/info/rfc7337>>.

Authors' Addresses

Frederic Fieau (editor)
Orange
40-48, avenue de la Republique
92320 Chatillon
France
Email: frederic.fieau@orange.com

Emile Stephan
Orange
2, avenue Pierre Marzin
22300 Lannion
France
Email: emile.stephan@orange.com

Sanjay Mishra
Verizon
13100 Columbia Pike
Silver Spring, MD 20904
United States of America
Email: sanjay.mishra@verizon.com

CDNI
Internet-Draft
Intended status: Standards Track
Expires: 23 September 2022

R. van Brandenburg
Tiledmedia
K. Leung

P. Sorber
Apple, Inc.
22 March 2022

URI Signing for Content Delivery Network Interconnection (CDNI)
draft-ietf-cdni-uri-signing-26

Abstract

This document describes how the concept of URI Signing supports the content access control requirements of Content Delivery Network Interconnection (CDNI) and proposes a URI Signing method as a JSON Web Token (JWT) profile.

The proposed URI Signing method specifies the information needed to be included in the URI to transmit the signed JWT, as well as the claims needed by the signed JWT to authorize a User Agent (UA). The mechanism described can be used both in CDNI and single Content Delivery Network (CDN) scenarios.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 23 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
1.1. Terminology	4
1.2. Background and overview on URI Signing	5
1.3. CDNI URI Signing Overview	6
1.4. URI Signing in a non-CDNI context	9
2. JWT Format and Processing Requirements	9
2.1. JWT Claims	10
2.1.1. Issuer (iss) claim	10
2.1.2. Subject (sub) claim	11
2.1.3. Audience (aud) claim	11
2.1.4. Expiry Time (exp) claim	11
2.1.5. Not Before (nbf) claim	12
2.1.6. Issued At (iat) claim	12
2.1.7. JWT ID (jti) claim	12
2.1.8. CDNI Claim Set Version (cdniv) claim	13
2.1.9. CDNI Critical Claims Set (cdnicrit) claim	13
2.1.10. Client IP Address (cdniip) claim	13
2.1.11. CDNI URI Container (cdniuc) claim	14
2.1.12. CDNI Expiration Time Setting (cdniets) claim	14
2.1.13. CDNI Signed Token Transport (cdnistt) claim	14
2.1.14. CDNI Signed Token Depth (cdnistd) claim	15
2.1.15. URI Container Forms	15
2.1.15.1. URI Hash Container (hash:)	16
2.1.15.2. URI Regular Expression Container (regex:)	16
2.2. JWT Header	16
3. URI Signing Token Renewal	17
3.1. Overview	17
3.2. Signed Token Renewal mechanism	17
3.2.1. Required Claims	18
3.3. Communicating a signed JWTs in Signed Token Renewal	18
3.3.1. Support for cross-domain redirection	18
4. Relationship with CDNI Interfaces	19
4.1. CDNI Control Interface	19
4.2. CDNI Footprint & Capabilities Advertisement Interface	19
4.3. CDNI Request Routing Redirection Interface	19
4.4. CDNI Metadata Interface	19
4.5. CDNI Logging Interface	21

5. URI Signing Message Flow	22
5.1. HTTP Redirection	22
5.2. DNS Redirection	25
6. IANA Considerations	28
6.1. CDNI Payload Type	28
6.1.1. CDNI UriSigning Payload Type	28
6.2. CDNI Logging Record Type	28
6.2.1. CDNI Logging Record Version 2 for HTTP	29
6.3. CDNI Logging Field Names	29
6.4. CDNI URI Signing Verification Code	30
6.5. CDNI URI Signing Signed Token Transport	31
6.6. JSON Web Token Claims Registration	32
6.6.1. Registry Contents	32
6.7. Expert Review Guidance	33
7. Security Considerations	33
8. Privacy	35
9. Acknowledgements	35
10. Contributors	35
11. References	35
11.1. Normative References	35
11.2. Informative References	37
Appendix A. Signed URI Package Example	39
A.1. Simple Example	40
A.2. Complex Example	40
A.3. Signed Token Renewal Example	41
Authors' Addresses	42

1. Introduction

This document describes the concept of URI Signing and how it can be used to provide access authorization in the case of redirection between cooperating CDNs and between a Content Service Provider (CSP) and a CDN. The primary goal of URI Signing is to make sure that only authorized UAs are able to access the content, with a CSP being able to authorize every individual request. It should be noted that URI Signing is not a content protection scheme; if a CSP wants to protect the content itself, other mechanisms, such as Digital Rights Management (DRM), are more appropriate. In addition to access control, URI Signing also has benefits in reducing the impact of denial-of-service attacks.

The overall problem space for CDN Interconnection (CDNI) is described in CDNI Problem Statement [RFC6707]. This document, along with the CDNI Requirements [RFC7337] document and the CDNI Framework [RFC7336], describes the need for interconnected CDNs to be able to implement an access control mechanism that enforces a CSP's distribution policies.

Specifically, the CDNI Framework [RFC7336] states:

The CSP may also trust the CDN operator to perform actions such as delegating traffic to additional downstream CDNs, and to enforce per-request authorization performed by the CSP using techniques such as URI Signing.

In particular, the following requirement is listed in the CDNI Requirements [RFC7337]:

MI-16 {HIGH} The CDNI Metadata interface shall allow signaling of authorization checks and verification that are to be performed by the Surrogate before delivery. For example, this could potentially include the need to verify information (e.g., Expiry time, Client IP address) required for access authorization.

This document defines a method of signing URIs that allows Surrogates in interconnected CDNs to enforce a per-request authorization initiated by the CSP. Splitting the role of initiating per-request authorization by the CSP and the role of verifying this authorization by the CDN allows any arbitrary distribution policy to be enforced across CDNs without the need of CDNs to have any awareness of the specific CSP distribution policies.

The method is implemented using Signed JSON Web Tokens (JWTs) [RFC7519].

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the terminology defined in the CDNI Problem Statement [RFC6707].

This document also uses the terminology of the JSON Web Token (JWT) [RFC7519].

In addition, the following terms are used throughout this document:

* Signed URI: A URI for which a signed JWT is provided.

- * Target CDN URI: URI created by the CSP to direct a UA towards the Upstream CDN (uCDN). The Target CDN URI can be signed by the CSP and verified by the uCDN and possibly further Downstream CDNs (dCDNs).
- * Redirection URI: URI created by the uCDN to redirect a UA towards the dCDN. The Redirection URI can be signed by the uCDN and verified by the dCDN. In a cascaded CDNI scenario, there can be more than one Redirection URI.
- * Signed Token Renewal: A series of signed JWTs that are used for subsequent access to a set of related resources in a CDN, such as a set of HTTP Adaptive Streaming files. Every time a signed JWT is used to access a particular resource, a new signed JWT is sent along with the resource that can be used to request the next resource in the set. When generating a new signed JWT in Signed Token Renewal, parameters are carried over from one signed JWT to the next.

1.2. Background and overview on URI Signing

A CSP and CDN are assumed to have a trust relationship that enables the CSP to authorize access to a content item, which is realized in practice by including a set of claims in a signed JWT in the URI before redirecting a UA to the CDN. Using these attributes, it is possible for a CDN to check an incoming content request to see whether it was authorized by the CSP (e.g., based on a time window or pattern matching the URI). To prevent the UA from altering the claims the JWT MUST be signed.

Figure 1, shown below, presents an overview of the URI Signing mechanism in the case of a CSP with a single CDN. When the UA browses for content on CSP's website (#1), it receives HTML web pages with embedded content URIs. Upon requesting these URIs, the CSP redirects to a CDN, creating a Target CDN URI (#2) (alternatively, the Target CDN URI itself is embedded in the HTML). The Target CDN URI is the Signed URI which may include the IP address of the UA and/or a time window. The signed URI always contains a signed JWT generated by the CSP using a shared secret or private key. Once the UA receives the response with the Signed URI, it sends a new HTTP request using the Signed URI to the CDN (#3). Upon receiving the request, the CDN authenticates the Signed URI by verifying the signed JWT. If applicable, the CDN checks whether the time window is still valid in the Signed URI and the pattern matches the URI of the request. After these claims are verified, the CDN delivers the content (#4).

Note: While using a symmetric shared key is supported, it is NOT RECOMMENDED. See the Security Considerations (Section 7) section about the limitations of shared keys.

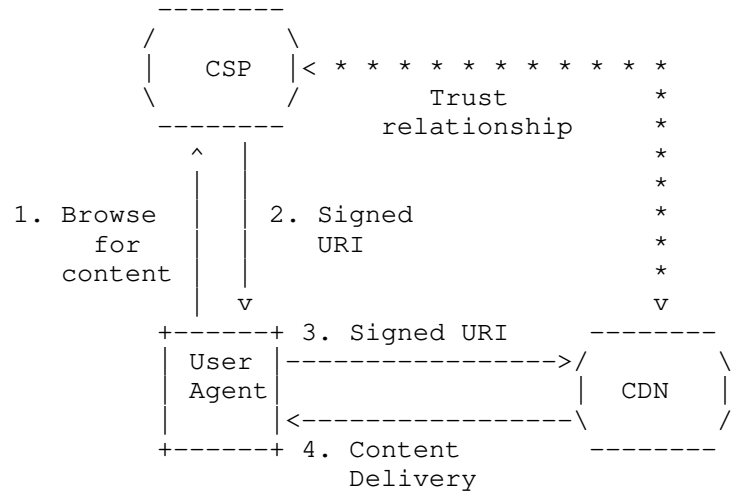


Figure 1: URI Signing in a CDN Environment

1.3. CDNI URI Signing Overview

In a CDNI environment, as shown in Figure 2 below, URI Signing operates the same way in the initial steps #1 and #2, but the later steps involve multiple CDNs delivering the content. The main difference from the single CDN case is a redirection step between the uCDN and the dCDN. In step #3, the UA may send an HTTP request or a DNS request, depending on whether HTTP-based or DNS-based request routing is used. The uCDN responds by directing the UA towards the dCDN using either a Redirection URI (i.e., a Signed URI generated by the uCDN) or a DNS reply, respectively (#4). Once the UA receives the response, it sends the Redirection URI/Target CDN URI to the dCDN (#5). The received URI is verified by the dCDN before delivering the content (#6). Note: The CDNI call flows are covered in Detailed URI Signing Operation (Section 5).

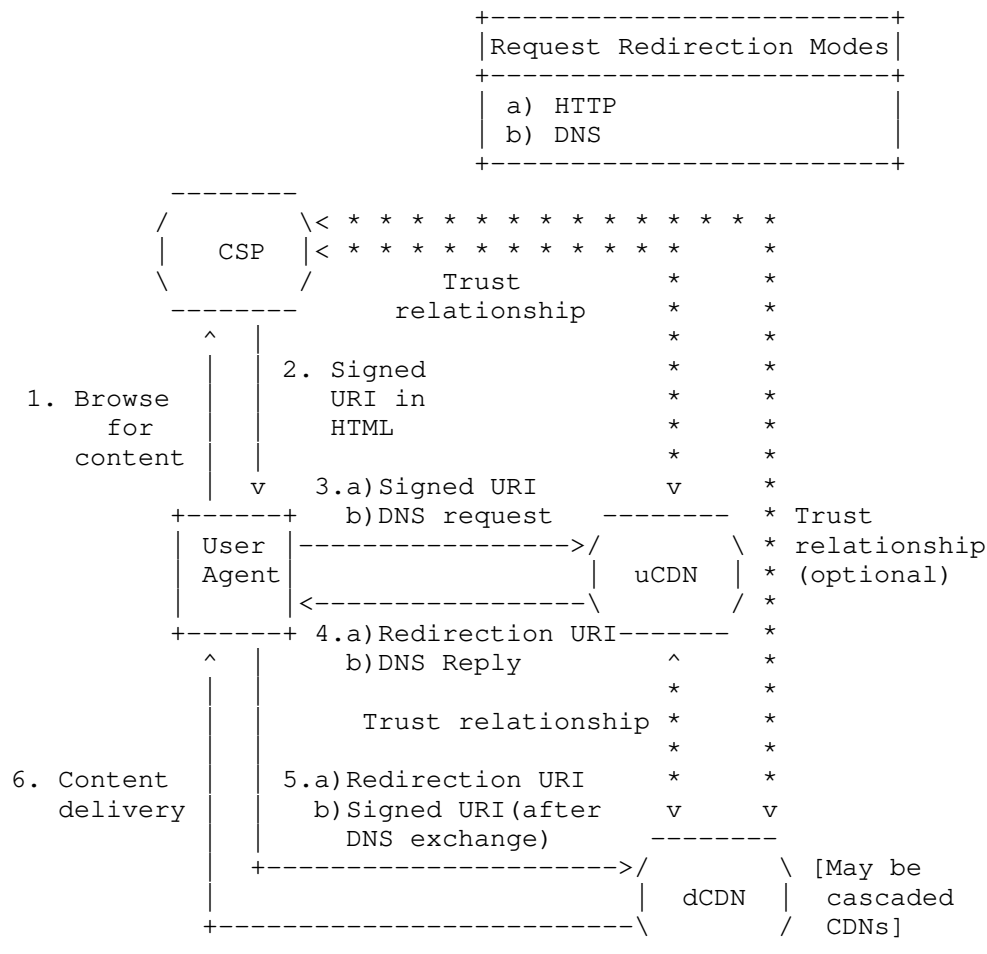


Figure 2: URI Signing in a CDNI Environment

The trust relationships between CSP, uCDN, and dCDN have direct implications for URI Signing. In the case shown in Figure 2, the CSP has a trust relationship with the uCDN. The delivery of the content may be delegated to a dCDN, which has a relationship with the uCDN but may have no relationship with the CSP.

In CDNI, there are two methods for request routing: DNS-based and HTTP-based. For DNS-based request routing, the Signed URI (i.e., the Target CDN URI) provided by the CSP reaches the CDN directly. In the case where the dCDN does not have a trust relationship with the CSP, this means that either an asymmetric public/private key method needs to be used for computing the signed JWT (because the CSP and dCDN are not able to exchange symmetric shared secret keys). Shared keys MUST NOT be redistributed.

For HTTP-based request routing, the Signed URI (i.e., the Target CDN URI) provided by the CSP reaches the uCDN. After this URI has been verified by the uCDN, the uCDN creates and signs a new Redirection URI, redirecting the UA to the dCDN. Since this new URI can have a new signed JWT, the relationship between the dCDN and CSP is not relevant. Because a relationship between uCDN and dCDN always exists, either asymmetric public/private keys or symmetric shared secret keys can be used for URI Signing with HTTP-based request routing. Note that the signed Redirection URI MUST maintain HTTPS as the scheme if it was present in the original and it MAY be upgraded from http: to https:.

Two types of keys can be used for URI Signing: asymmetric keys and symmetric shared keys. Asymmetric keys are based on a public/private key pair mechanism and always contain a private key known only to the entity signing the URI (either CSP or uCDN) and a public key for the verification of the Signed URI. With symmetric keys, the same key is used by both the signing entity for signing the URI and the verifying entity for verifying the Signed URI. Regardless of the type of keys used, the verifying entity has to obtain the key in a manner that allows trust to be placed in the assertions made using that key (either the public or the symmetric key). There are very different requirements (outside the scope of this document) for distributing asymmetric keys and symmetric keys. Key distribution for symmetric keys requires confidentiality to prevent third parties from getting access to the key, since they could then generate valid Signed URIs for unauthorized requests. Key distribution for asymmetric keys does not require confidentiality since public keys can typically be distributed openly (because they cannot be used to sign URIs) and the corresponding private keys are kept secret by the URI signer.

Note: While using a symmetric shared key is supported, it is NOT RECOMMENDED. See the Security Considerations (Section 7) section about the limitations of shared keys.

1.4. URI Signing in a non-CDNI context

While the URI Signing method defined in this document was primarily created for the purpose of allowing URI Signing in CDNI scenarios, i.e., between a uCDN and a dCDN, there is nothing in the defined URI Signing method that precludes it from being used in a non-CDNI context. As such, the described mechanism could be used in a single-CDN scenario such as shown in Figure 1 in Section 1.2, for example to allow a CSP that uses different CDNs to only have to implement a single URI Signing mechanism.

2. JWT Format and Processing Requirements

The concept behind URI Signing is based on embedding a signed JSON Web Token (JWT) [RFC7519] in an HTTP or HTTPS URI [RFC7230] (see [RFC7230] Section 2.7). The signed JWT contains a number of claims that can be verified to ensure the UA has legitimate access to the content.

This document specifies the following attribute for embedding a signed JWT in a Target CDN URI or Redirection URI:

- * URI Signing Package (URISigningPackage): The URI attribute that encapsulates all the URI Signing claims in a signed JWT encoded format. This attribute is exposed in the Signed URI as a path-style parameter or a form-style parameter.

The parameter name of the URI Signing Package Attribute is defined in the CDNI Metadata (Section 4.4). If the CDNI Metadata interface is not used, or does not include a parameter name for the URI Signing Package Attribute, the parameter name can be set by configuration (out of scope of this document).

The URI Signing Package will be found by parsing any path-style parameters and form-style parameters looking for a key name matching the URI Signing Package Attribute. Both parameter styles MUST be supported to allow flexibility of operation. The first matching parameter SHOULD be taken to provide the signed JWT, though providing more than one matching key is undefined behavior. Path-style parameters generated in the form indicated by Section 3.2.7 of [RFC6570] and Form-style parameters generated in the form indicated by Sections 3.2.8 and 3.2.9 of [RFC6570] MUST be supported.

The following is an example where the URI Signing Package Attribute name is "token" and the signed JWT is "SIGNEDJWT":

`http://example.com/media/path?come=data&token=SIGNEDJWT&other=data`

2.1. JWT Claims

This section identifies the set of claims that can be used to enforce the CSP distribution policy. New claims can be introduced in the future to extend the distribution policy capabilities.

In order to provide distribution policy flexibility, the exact subset of claims used in a given signed JWT is a runtime decision. Claim requirements are defined in the CDNI Metadata (Section 4.4). If the CDNI Metadata interface is not used, or does not include claim requirements, the claim requirements can be set by configuration (out of scope of this document).

The following claims (where the "JSON Web Token Claims" registry claim name is specified in parentheses below) are used to enforce the distribution policies. All of the listed claims are mandatory to implement in a URI Signing implementation, but are not necessarily mandatory to use in a given signed JWT. (The "optional" and "mandatory" identifiers in square brackets refer to whether or not a given claim **MUST** be present in a URI Signing JWT.)

Note: The time on the entities that generate and verify the signed URI **MUST** be in sync. In the CDNI case, this means that CSP, uCDN, and dCDN servers need to be time-synchronized. It is **RECOMMENDED** to use NTP [RFC5905] for time synchronization.

Note: See the Security Considerations (Section 7) section on the limitations of using an expiration time and client IP address for distribution policy enforcement.

2.1.1. Issuer (iss) claim

Issuer (iss) [optional] - The semantics in [RFC7519] Section 4.1.1 **MUST** be followed. If this claim is used, it **MUST** be used to identify the issuer (signer) of the JWT. In particular, the recipient will have already received, in trusted configuration, a mapping of issuer name to one or more keys used to sign JWTs, and must verify that the JWT was signed by one of those keys. If this claim is used and the CDN verifying the signed JWT does not support Issuer verification, or if the Issuer in the signed JWT does not match the list of known acceptable Issuers, or if the Issuer claim does not match the key used to sign the JWT, the CDN **MUST** reject the request. If the received signed JWT contains an Issuer claim, then any JWT subsequently generated for CDNI redirection **MUST** also contain an Issuer claim, and the Issuer value **MUST** be updated to identify the redirecting CDN. If the received signed JWT does not contain an Issuer claim, an Issuer claim **MAY** be added to a signed JWT generated for CDNI redirection.

2.1.2. Subject (sub) claim

Subject (sub) [optional] - The semantics in [RFC7519] Section 4.1.2 MUST be followed. If this claim is used, it MUST be a JSON Web Encryption (JWE [RFC7516]) Object in compact serialization form, because it contains personally identifiable information. This claim contains information about the subject (for example, a user or an agent) that MAY be used to verify the signed JWT. If the received signed JWT contains a Subject claim, then any JWT subsequently generated for CDNI redirection MUST also contain a Subject claim, and the Subject value MUST be the same as in the received signed JWT. A signed JWT generated for CDNI redirection MUST NOT add a Subject claim if no Subject claim existed in the received signed JWT.

2.1.3. Audience (aud) claim

Audience (aud) [optional] - The semantics in [RFC7519] Section 4.1.3 MUST be followed. This claim is used to ensure that the CDN verifying the JWT is an intended recipient of the request. The claim MUST contain an identity belonging to the chain of entities involved in processing the request (e.g., identifying the CSP or any CDN in the chain) that the recipient is configured to use for the processing of this request. A CDN MAY modify the claim as long it can generate a valid signature.

2.1.4. Expiry Time (exp) claim

Expiry Time (exp) [optional] - The semantics in [RFC7519] Section 4.1.4 MUST be followed, though URI Signing implementations MUST NOT allow for any time synchronization "leeway". If this claim is used and the CDN verifying the signed JWT does not support Expiry Time verification, or if the Expiry Time in the signed JWT corresponds to a time equal to or earlier than the time of the content request, the CDN MUST reject the request. If the received signed JWT contains an Expiry Time claim, then any JWT subsequently generated for CDNI redirection MUST also contain an Expiry Time claim, and the Expiry Time value MUST be the same as in the received signed JWT. A signed JWT generated for CDNI redirection MUST NOT add an Expiry Time claim if no Expiry Time claim existed in the received signed JWT.

2.1.5. Not Before (nbf) claim

Not Before (nbf) [optional] - The semantics in [RFC7519] Section 4.1.5 MUST be followed, though URI Signing implementations MUST NOT allow for any time synchronization "leeway". If this claim is used and the CDN verifying the signed JWT does not support Not Before time verification, or if the Not Before time in the signed JWT corresponds to a time later than the time of the content request, the CDN MUST reject the request. If the received signed JWT contains a Not Before time claim, then any JWT subsequently generated for CDNI redirection MUST also contain a Not Before time claim, and the Not Before time value MUST be the same as in the received signed JWT. A signed JWT generated for CDNI redirection MUST NOT add a Not Before time claim if no Not Before time claim existed in the received signed JWT.

2.1.6. Issued At (iat) claim

Issued At (iat) [optional] - The semantics in [RFC7519] Section 4.1.6 MUST be followed. If the received signed JWT contains an Issued At claim, then any JWT subsequently generated for CDNI redirection MUST also contain an Issued At claim, and the Issued At value MUST be updated to identify the time the new JWT was generated. If the received signed JWT does not contain an Issued At claim, an Issued At claim MAY be added to a signed JWT generated for CDNI redirection.

2.1.7. JWT ID (jti) claim

JWT ID (jti) [optional] - The semantics in [RFC7519] Section 4.1.7 MUST be followed. A JWT ID can be used to prevent replay attacks if the CDN stores a list of all previously used values, and verifies that the value in the current JWT has never been used before. If the signed JWT contains a JWT ID claim and the CDN verifying the signed JWT either does not support JWT ID storage or has previously seen the value used in a request for the same content, then the CDN MUST reject the request. If the received signed JWT contains a JWT ID claim, then any JWT subsequently generated for CDNI redirection MUST also contain a JWT ID claim, and the value MUST be the same as in the received signed JWT. If the received signed JWT does not contain a JWT ID claim, a JWT ID claim MUST NOT be added to a signed JWT generated for CDNI redirection. Sizing of the JWT ID is application dependent given the desired security constraints.

2.1.1.8. CDNI Claim Set Version (cdniv) claim

CDNI Claim Set Version (cdniv) [optional] - The CDNI Claim Set Version (cdniv) claim provides a means within a signed JWT to tie the claim set to a specific version of this specification. The cdniv claim is intended to allow changes in and facilitate upgrades across specifications. The type is JSON integer and the value MUST be set to "1", for this version of the specification. In the absence of this claim, the value is assumed to be "1". For future versions this claim will be mandatory. Implementations MUST reject signed JWTs with unsupported CDNI Claim Set versions.

2.1.1.9. CDNI Critical Claims Set (cdnicrit) claim

CDNI Critical Claims Set (cdnicrit) [optional] - The CDNI Critical Claims Set (cdnicrit) claim indicates that extensions to this specification are being used that MUST be understood and processed. Its value is a comma separated listing of claims in the Signed JWT that use those extensions. If any of the listed extension claims are not understood and supported by the recipient, then the Signed JWT MUST be rejected. Producers MUST NOT include claim names defined by this specification, duplicate names, or names that do not occur as claim names within the Signed JWT in the cdnicrit list. Producers MUST NOT use the empty list "" as the cdnicrit value. Recipients MAY consider the Signed JWT to be invalid if the cdnicrit list contains any claim names defined by this specification or if any other constraints on its use are violated. This claim MUST be understood and processed by all implementations.

2.1.1.10. Client IP Address (cdniip) claim

Client IP Address (cdniip) [optional] - The Client IP Address (cdniip) claim holds an IP address or IP prefix for which the Signed URI is valid. This is represented in CIDR notation, with dotted decimal format for IPv4 addresses [RFC0791] or canonical text representation for IPv6 addresses [RFC5952]. The request MUST be rejected if sourced from a client outside the specified IP range. Since the client IP is considered personally identifiable information this field MUST be a JSON Web Encryption (JWE [RFC7516]) Object in compact serialization form. If the CDN verifying the signed JWT does not support Client IP verification, or if the Client IP in the signed JWT does not match the source IP address in the content request, the CDN MUST reject the request. The type of this claim is a JSON string that contains the JWE. If the received signed JWT contains a Client IP claim, then any JWT subsequently generated for CDNI redirection MUST also contain a Client IP claim, and the Client IP value MUST be the same as in the received signed JWT. A signed JWT generated for CDNI redirection MUST NOT add a Client IP claim if no Client IP claim

existed in the received signed JWT.

It should be noted that use of this claim can cause issues, for example, in situations with dual-stack IPv4 and IPv6 networks, MPTCP, QUIC, and mobile clients switching from Wi-Fi to Cellular networks where the client's source address can change, even between address families. This claim exists mainly for legacy feature parity reasons, therefore use of this claim should be done judiciously. An example of a reasonable use case would be making a signed JWT for an internal preview of an asset where the end consumer understands that they must be originated from the same IP for the entirety of the session. Using this claim at large is NOT RECOMMENDED.

2.1.11. CDNI URI Container (cdniuc) claim

URI Container (cdniuc) [mandatory] - The URI Container (cdniuc) holds the URI representation before a URI Signing Package is added. This representation can take one of several forms detailed in Section 2.1.15. If the URI Container used in the signed JWT does not match the URI of the content request, the CDN verifying the signed JWT MUST reject the request. When comparing the URI, the percent encoded form as defined in [RFC3986] Section 2.1 MUST be used. When redirecting a URI, the CDN generating the new signed JWT MAY change the URI Container to comport with the URI being used in the redirection.

2.1.12. CDNI Expiration Time Setting (cdniets) claim

CDNI Expiration Time Setting (cdniets) [optional] - The CDNI Expiration Time Setting (cdniets) claim provides a means for setting the value of the Expiry Time (exp) claim when generating a subsequent signed JWT in Signed Token Renewal. Its type is a JSON numeric value. It denotes the number of seconds to be added to the time at which the JWT is verified that gives the value of the Expiry Time (exp) claim of the next signed JWT. The CDNI Expiration Time Setting (cdniets) SHOULD NOT be used when not using Signed Token Renewal and MUST be present when using Signed Token Renewal.

2.1.13. CDNI Signed Token Transport (cdnistt) claim

CDNI Signed Token Transport (cdnistt) [optional] - The CDNI Signed Token Transport (cdnistt) claim provides a means of signalling the method through which a new signed JWT is transported from the CDN to the UA and vice versa for the purpose of Signed Token Renewal. Its type is a JSON integer. Values for this claim are defined in Section 6.5. If using this claim you MUST also specify a CDNI Expiration Time Setting (cdniets) as noted above.

2.1.14. CDNI Signed Token Depth (cdnistd) claim

CDNI Signed Token Depth (cdnistd) [optional] - The CDNI Signed Token Depth (cdnistd) claim is used to associate a subsequent signed JWT, generated as the result of a CDNI Signed Token Transport claim, with a specific URI subset. Its type is a JSON integer. Signed JWTs MUST NOT use a negative value for the CDNI Signed Token Depth claim.

If the transport used for Signed Token Transport allows the CDN to associate the path component of a URI with tokens (e.g., an HTTP Cookie Path as described in section 4.1.2.4 of [RFC6265]), the CDNI Signed Token Depth value is the number of path segments that should be considered significant for this association. A CDNI Signed Token Depth of zero means that the client SHOULD be directed to return the token with requests for any path. If the CDNI Signed Token Depth is greater than zero, then the CDN SHOULD send the client a token to return for future requests wherein the first CDNI Signed Token Depth segments of the path match the first CDNI Signed Token Depth segments of the signed URI path. This matching MUST use the URI with the token removed, as specified in Section 2.1.15.

If the URI path to match contains fewer segments than the CDNI Signed Token Depth claim, a signed JWT MUST NOT be generated for the purposes of Signed Token Renewal. If the CDNI Signed Token Depth claim is omitted, it means the same thing as if its value were zero. If the received signed JWT contains a CDNI Signed Token Depth claim, then any JWT subsequently generated for CDNI redirection or Signed Token Transport MUST also contain a CDNI Signed Token Depth claim, and the value MUST be the same as in the received signed JWT.

2.1.15. URI Container Forms

The URI Container (cdniuc) claim takes one of the following forms: 'hash:' or 'regex:'. More forms may be added in the future to extend the capabilities.

Before comparing a URI with contents of this container, the following steps MUST be performed:

- * Prior to verification, remove the signed JWT from the URI. This removal is only for the purpose of determining if the URI matches; all other purposes will use the original URI. If the signed JWT is terminated by anything other than a sub-delimiter (as defined in [RFC3986] Section 2.2), everything from the reserved character (as defined in [RFC3986] Section 2.2) that precedes the URI Signing Package Attribute to the last character of the signed JWT will be removed, inclusive. Otherwise, everything from the first character of the URI Signing Package Attribute to the sub-delimiter that terminates the signed JWT will be removed, inclusive.
- * Normalize the URI according to section 2.7.3 [RFC7230] and sections 6.2.2 and 6.2.3 [RFC3986]. This applies to both generation and verification of the signed JWT.

2.1.15.1. URI Hash Container (hash:)

Prefixed with 'hash:', this string is a URL Segment form ([RFC6920] Section 5) of the URI.

2.1.15.2. URI Regular Expression Container (regex:)

Prefixed with 'regex:', this string is any POSIX Section 9 [POSIX.1] Extended Regular Expression compatible regular expression used to match against the requested URI. These regular expressions MUST be evaluated in the POSIX locale (POSIX Section 7.2 [POSIX.1]).

Note: Because '\' has special meaning in JSON [RFC8259] as the escape character within JSON strings, the regular expression character '\' MUST be escaped as '\\'.

An example of a 'regex:' is the following:

```
[^:]*\\://[^\/*]/folder/content/quality_[^\/*]/segment.{3}\\\.mp4(\\?.*)?
```

Note: Due to computational complexity of executing arbitrary regular expressions, it is RECOMMENDED to only execute after verifying the JWT to ensure its authenticity.

2.2. JWT Header

The header of the JWT MAY be passed via the CDNI Metadata interface instead of being included in the URISigningPackage. The header value MUST be transmitted in the serialized encoded form and prepended to the JWT payload and signature passed in the URISigningPackage prior to verification. This reduces the size of the signed JWT token.

3. URI Signing Token Renewal

3.1. Overview

For content that is delivered via HTTP in a segmented fashion, such as MPEG-DASH [MPEG-DASH] or HTTP Live Streaming (HLS) [RFC8216], special provisions need to be made in order to ensure URI Signing can be applied. In general, segmented protocols work by breaking large objects (e.g., videos) into a sequence of small independent segments. Such segments are then referenced by a separate manifest file, which either includes a list of URLs to the segments or specifies an algorithm through which a User Agent can construct the URLs to the segments. Requests for segments therefore originate from the manifest file and, unless the URLs in the manifest file point to the CSP, are not subjected to redirection and URI Signing. This opens up a vulnerability to malicious User Agents sharing the manifest file and deep-linking to the segments.

One method for dealing with this vulnerability would be to include, in the manifest itself, Signed URIs that point to the individual segments. There exist a number of issues with that approach. First, it requires the CDN delivering the manifest to rewrite the manifest file for each User Agent, which would require the CDN to be aware of the exact segmentation protocol used. Secondly, it could also require the expiration time of the Signed URIs to be valid for an extended duration if the content described by the manifest is meant to be consumed in real time. For instance, if the manifest file were to contain a segmented video stream of more than 30 minutes in length, Signed URIs would require to be valid for at least 30 minutes, thereby reducing their effectiveness and that of the URI Signing mechanism in general. For a more detailed analysis of how segmented protocols such as HTTP Adaptive Streaming protocols affect CDNI, see Models for HTTP-Adaptive-Streaming-Aware CDNI [RFC6983].

The method described in this section allows CDNs to use URI Signing for segmented content without having to include the Signed URIs in the manifest files themselves.

3.2. Signed Token Renewal mechanism

In order to allow for effective access control of segmented content, the URI Signing mechanism defined in this section is based on a method through which subsequent segment requests can be linked together. As part of the JWT verification procedure, the CDN can generate a new signed JWT that the UA can use to do a subsequent request. More specifically, whenever a UA successfully retrieves a segment, it receives, in the HTTP 2xx Successful message, a signed JWT that it can use whenever it requests the next segment. As long

as each successive signed JWT is correctly verified before a new one is generated, the model is not broken, and the User Agent can successfully retrieve additional segments. Given the fact that with segmented protocols, it is usually not possible to determine a priori which segment will be requested next (i.e., to allow for seeking within the content and for switching to a different representation), the Signed Token Renewal uses the URI Regular Expression Container scoping mechanisms in the URI Container (cdniuc) claim to allow a signed JWT to be valid for more than one URL.

In order for this renewal of signed JWTs to work, it is necessary for a UA to extract the signed JWT from the HTTP 2xx Successful message of an earlier request and use it to retrieve the next segment. The exact mechanism by which the client does this is outside the scope of this document. However, in order to also support legacy UAs that do not include any specific provisions for the handling of signed JWTs, Section 3.3 defines a mechanism using HTTP Cookies [RFC6265] that allows such UAs to support the concept of renewing signed JWTs without requiring any additional UA support.

3.2.1. Required Claims

The cdnistt claim (Section 2.1.13) and cdniets claim (Section 2.1.12) MUST both be present for Signed Token Renewal. cdnistt MAY be set to a value of '0' to mean no Signed Token Renewal, but there still MUST be a corresponding cdniets that verifies as a JSON number. However, if use of Signed Token Renewal is not desired, it is RECOMMENDED to simply omit both.

3.3. Communicating a signed JWTs in Signed Token Renewal

This section assumes the value of the CDNI Signed Token Transport (cdnistt) claim has been set to 1.

When using the Signed Token Renewal mechanism, the signed JWT is transported to the UA via a 'URISigningPackage' cookie added to the HTTP 2xx Successful message along with the content being returned to the UA, or to the HTTP 3xx Redirection message in case the UA is redirected to a different server.

3.3.1. Support for cross-domain redirection

For security purposes, the use of cross-domain cookies is not supported in some application environments. As a result, the Cookie-based method for transport of the Signed Token described in Section 3.3 might break if used in combination with an HTTP 3xx Redirection response where the target URL is in a different domain. In such scenarios, Signed Token Renewal of a signed JWT SHOULD be

communicated via the query string instead, in a similar fashion to how regular signed JWTs (outside of Signed Token Renewal) are communicated. Note the value of the CDNI Signed Token Transport (cdnistt) claim MUST be set to 2.

Note that the process described herein only works in cases where both the manifest file and segments constituting the segmented content are delivered from the same domain. In other words, any redirection between different domains needs to be carried out while retrieving the manifest file.

4. Relationship with CDNI Interfaces

Some of the CDNI Interfaces need enhancements to support URI Signing. A dCDN that supports URI Signing needs to be able to advertise this capability to the uCDN. The uCDN needs to select a dCDN based on such capability when the CSP requires access control to enforce its distribution policy via URI Signing. Also, the uCDN needs to be able to distribute via the CDNI Metadata interface the information necessary to allow the dCDN to verify a Signed URI. Events that pertain to URI Signing (e.g., request denial or delivery after an access authorization decision has been made) need to be included in the logs communicated through the CDNI Logging interface.

4.1. CDNI Control Interface

URI Signing has no impact on this interface.

4.2. CDNI Footprint & Capabilities Advertisement Interface

The CDNI Request Routing: Footprint and Capabilities Semantics document [RFC8008] defines support for advertising CDNI Metadata capabilities, via CDNI Payload Type. The CDNI Payload Type registered in Section 6.1 can be used for capability advertisement.

4.3. CDNI Request Routing Redirection Interface

The CDNI Request Routing Redirection Interface [RFC7975] describes the recursive request redirection method. For URI Signing, the uCDN signs the URI provided by the dCDN. URI Signing therefore has no impact on this interface.

4.4. CDNI Metadata Interface

The CDNI Metadata Interface [RFC8006] describes the CDNI metadata distribution needed to enable content acquisition and delivery. For URI Signing, a new CDNI metadata object is specified.

The UriSigning Metadata object contains information to enable URI Signing and verification by a dCDN. The UriSigning properties are defined below.

Property: enforce

Description: URI Signing enforcement flag. Specifically, this flag indicates if the access to content is subject to URI Signing. URI Signing requires the dCDN to ensure that the URI is signed and verified before delivering content. Otherwise, the dCDN does not perform verification, regardless of whether or not the URI is signed.

Type: Boolean

Mandatory-to-Specify: No. The default is true.

Property: issuers

Description: A list of valid Issuers against which the Issuer claim in the signed JWT may be cross-referenced.

Type: Array of Strings

Mandatory-to-Specify: No. The default is an empty list. An empty list means that any Issuer in the trusted key store of issuers is acceptable.

Property: package-attribute

Description: The attribute name to use for the URI Signing Package.

Type: String

Mandatory-to-Specify: No. The default is "URISigningPackage".

Property: jwt-header

Description: The header part of JWT that is used for verifying a signed JWT when the JWT token in the URI Signing Package does not contain a header part.

Type: String

Mandatory-to-Specify: No. By default, the header is assumed to be included in the JWT token.

The following is an example of a URI Signing metadata payload with all default values:

```
{
  "generic-metadata-type": "MI.UriSigning"
  "generic-metadata-value": {}
}
```

The following is an example of a URI Signing metadata payload with explicit values:

```
{
  "generic-metadata-type": "MI.UriSigning"
  "generic-metadata-value":
    {
      "enforce": true,
      "issuers": ["csp", "ucdn1", "ucdn2"],
      "package-attribute": "usp",
      "jwt-header":
        {
          "alg": "ES256",
          "kid": "P5UpOv0eMqlwcxLf7WxIg09JdSYGYFDOWkldueaImf0"
        }
    }
}
```

4.5. CDNI Logging Interface

For URI Signing, the dCDN reports that enforcement of the access control was applied to the request for content delivery. When the request is denied due to enforcement of URI Signing, the reason is logged.

The following CDNI Logging field for URI Signing SHOULD be supported in the HTTP Request Logging Record as specified in CDNI Logging Interface [RFC7937], using the new "cdni_http_request_v2" record-type registered in Section 6.2.1.

* s-uri-signing (mandatory):

- format: 3DIGIT
- field value: this characterises the URI Signing verification performed by the Surrogate on the request. The allowed values are registered in Section 6.4.

- occurrence: there MUST be zero or exactly one instance of this field.
- * s-uri-signing-deny-reason (optional):
 - format: QSTRING
 - field value: a string for providing further information in case the signed JWT was rejected, e.g., for debugging purposes.
 - occurrence: there MUST be zero or exactly one instance of this field.

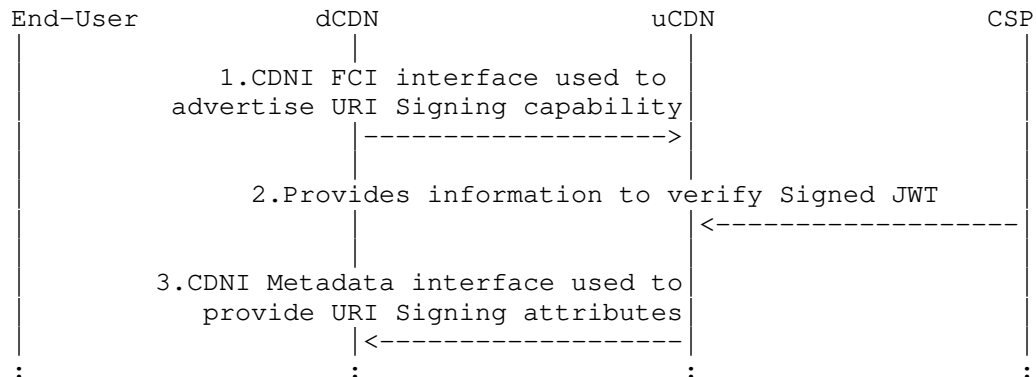
5. URI Signing Message Flow

URI Signing supports both HTTP-based and DNS-based request routing. JSON Web Token (JWT) [RFC7519] defines a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a Signed JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC).

5.1. HTTP Redirection

For HTTP-based request routing, a set of information that is unique to a given end user content request is included in a Signed JWT, using key information that is specific to a pair of adjacent CDNI hops (e.g., between the CSP and the uCDN or between the uCDN and a dCDN). This allows a CDNI hop to ascertain the authenticity of a given request received from a previous CDNI hop.

The URI Signing method (assuming HTTP redirection, iterative request routing, and a CDN path with two CDNs) includes the following steps:



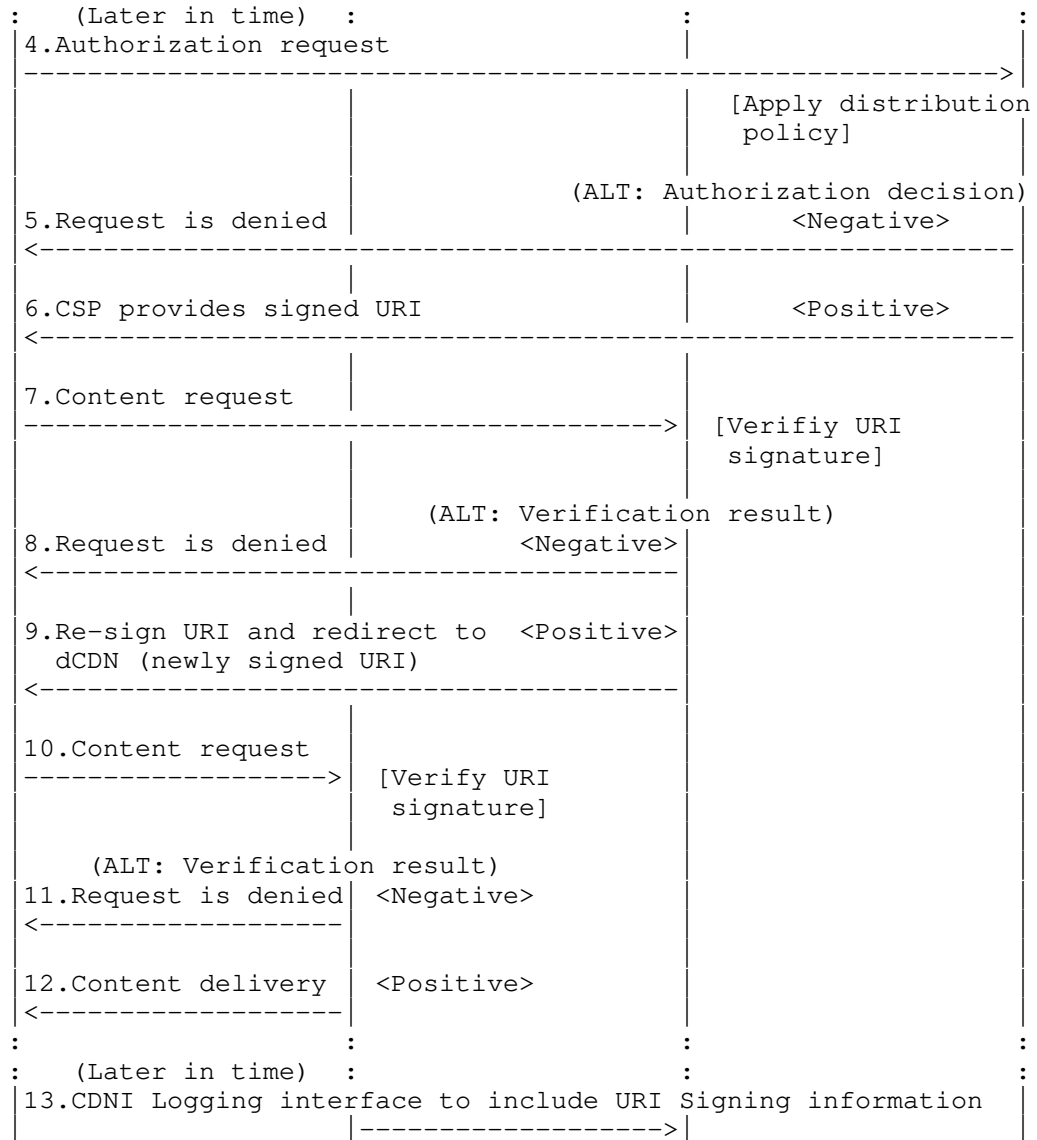


Figure 3: HTTP-based Request Routing with URI Signing

1. Using the CDNI Footprint & Capabilities Advertisement interface, the dCDN advertises its capabilities including URI Signing support to the uCDN.

2. CSP provides to the uCDN the information needed to verify signed URIs from that CSP. For example, this information will include one or more keys used for validation.
3. Using the CDNI Metadata interface, the uCDN communicates to a dCDN the information needed to verify signed URIs from the uCDN for the given CSP. For example, this information may include the URI query string parameter name for the URI Signing Package Attribute in addition to keys used for validation.
4. When a UA requests a piece of protected content from the CSP, the CSP makes a specific authorization decision for this unique request based on its local distribution policy.
5. If the authorization decision is negative, the CSP rejects the request and sends an error code (e.g., 403 Forbidden) in the HTTP response.
6. If the authorization decision is positive, the CSP computes a Signed JWT that is based on unique parameters of that request and conveys it to the end user as the URI to use to request the content.
7. On receipt of the corresponding content request, the uCDN verifies the Signed JWT in the URI using the information provided by the CSP.
8. If the verification result is negative, the uCDN rejects the request and sends an error code 403 Forbidden in the HTTP response.
9. If the verification result is positive, the uCDN computes a Signed JWT that is based on unique parameters of that request and provides it to the end user as the URI to use to further request the content from the dCDN.
10. On receipt of the corresponding content request, the dCDN verifies the Signed JWT in the signed URI using the information provided by the uCDN in the CDNI Metadata.
11. If the verification result is negative, the dCDN rejects the request and sends an error code 403 Forbidden in the HTTP response.
12. If the verification result is positive, the dCDN serves the request and delivers the content.

13. At a later time, the dCDN reports logging events that include URI Signing information.

With HTTP-based request routing, URI Signing matches well the general chain of trust model of CDNI both with symmetric and asymmetric keys because the key information only needs to be specific to a pair of adjacent CDNI hops.

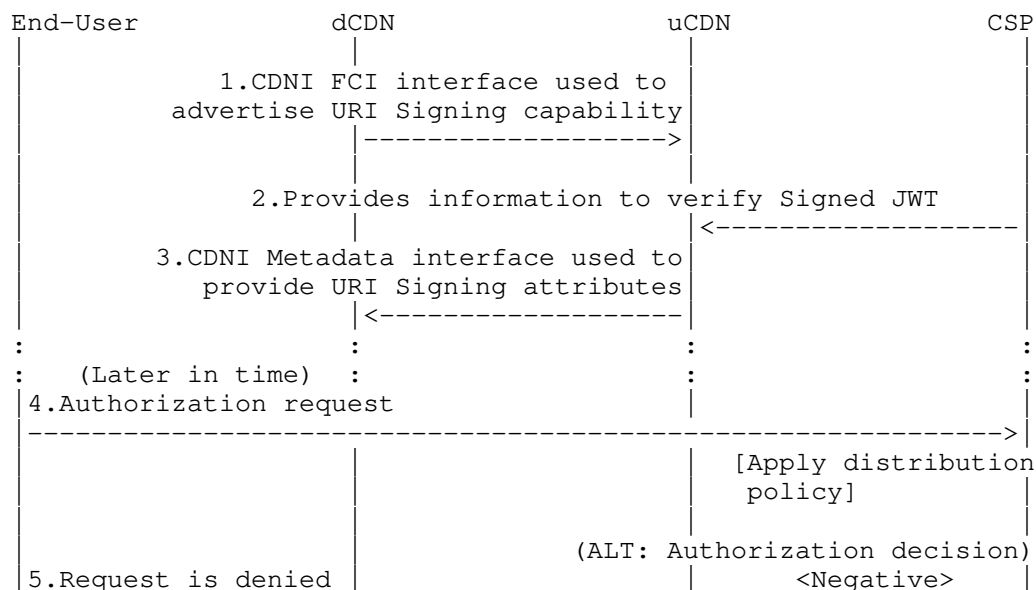
Note: While using a symmetric shared key is supported, it is NOT RECOMMENDED. See the Security Considerations (Section 7) section about the limitations of shared keys.

5.2. DNS Redirection

For DNS-based request routing, the CSP and uCDN must agree on a trust model appropriate to the security requirements of the CSP's particular content. Use of asymmetric public/private keys allows for unlimited distribution of the public key to dCDNs. However, if a shared secret key is required, then the distribution SHOULD be performed by the CSP directly.

Note: While using a symmetric shared key is supported, it is NOT RECOMMENDED. See the Security Considerations (Section 7) section about the limitations of shared keys.

The URI Signing method (assuming iterative DNS request routing and a CDN path with two CDNs) includes the following steps.



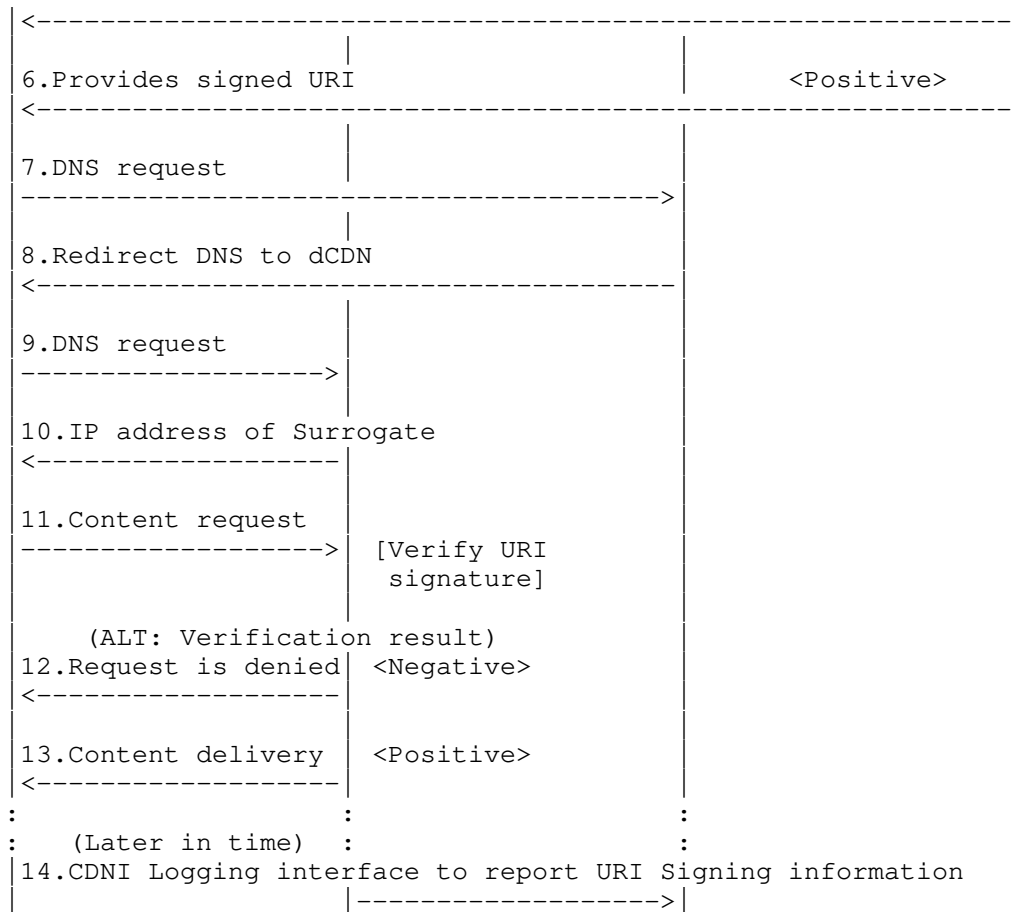


Figure 4: DNS-based Request Routing with URI Signing

1. Using the CDNI Footprint & Capabilities Advertisement interface, the dCDN advertises its capabilities including URI Signing support to the uCDN.
2. CSP provides to the uCDN the information needed to verify Signed JWTs from that CSP. For example, this information will include one or more keys used for validation.
3. Using the CDNI Metadata interface, the uCDN communicates to a dCDN the information needed to verify Signed JWTs from the CSP (e.g., the URI query string parameter name for the URI Signing Package Attribute). In the case of symmetric shared key, the uCDN MUST NOT share the key with a dCDN.

4. When a UA requests a piece of protected content from the CSP, the CSP makes a specific authorization decision for this unique request based on its local distribution policy.
5. If the authorization decision is negative, the CSP rejects the request and sends an error code (e.g., 403 Forbidden) in the HTTP response.
6. If the authorization decision is positive, the CSP computes a Signed JWT that is based on unique parameters of that request and includes it in the URI provided to the end user to request the content.
7. End user sends DNS request to the uCDN.
8. On receipt of the DNS request, the uCDN redirects the request to the dCDN.
9. End user sends DNS request to the dCDN.
10. On receipt of the DNS request, the dCDN responds with IP address of one of its Surrogates.
11. On receipt of the corresponding content request, the dCDN verifies the Signed JWT in the URI using the information provided by the uCDN in the CDNI Metadata.
12. If the verification result is negative, the dCDN rejects the request and sends an error code 403 Forbidden in the HTTP response.
13. If the verification result is positive, the dCDN serves the request and delivers the content.
14. At a later time, dCDN reports logging events that includes URI Signing information.

With DNS-based request routing, URI Signing matches well the general chain of trust model of CDNI when used with asymmetric keys because the only key information that needs to be distributed across multiple, possibly untrusted, CDNI hops is the public key, which is generally not confidential.

With DNS-based request routing, URI Signing does not match well with the general chain of trust model of CDNI when used with symmetric keys because the symmetric key information needs to be distributed across multiple CDNI hops, to CDNs with which the CSP may not have a trust relationship. This raises a security concern for applicability

of URI Signing with symmetric keys in case of DNS-based inter-CDN request routing. Due to these flaws, this architecture MUST NOT be implemented.

Note: While using a symmetric shared key is supported, it is NOT RECOMMENDED. See the Security Considerations (Section 7) section about the limitations of shared keys.

6. IANA Considerations

6.1. CDNI Payload Type

This document requests the registration of the following CDNI Payload Type under the IANA "CDNI Payload Types" registry:

Payload Type	Specification
MI.UriSigning	RFcthis

Table 1

[RFC Editor: Please replace RFcthis with the published RFC number for this document.]

6.1.1. CDNI UriSigning Payload Type

Purpose: The purpose of this payload type is to distinguish UriSigning MI objects (and any associated capability advertisement).

Interface: MI/FCI

Encoding: see Section 4.4

6.2. CDNI Logging Record Type

This document requests the registration of the following CDNI Logging record-type under the IANA "CDNI Logging record-types" registry:

record-types	Reference	Description
cdni_http_request_v2	RFCThis	Extension to CDNI Logging Record version 1 for content delivery using HTTP, to include URI Signing logging fields

Table 2

[RFC Editor: Please replace RFCThis with the published RFC number for this document.]

6.2.1. CDNI Logging Record Version 2 for HTTP

The "cdni_http_request_v2" record-type supports all of the fields supported by the "cdni_http_request_v1" record-type [RFC7937] plus the two additional fields "s-uri-signing" and "s-uri-signing-deny-reason", registered by this document in Section 6.3. The name, format, field value, and occurrence information for the two new fields can be found in Section 4.5 of this document.

6.3. CDNI Logging Field Names

This document requests the registration of the following CDNI Logging fields under the IANA "CDNI Logging Field Names" registry:

Field Name	Reference
s-uri-signing	RFCThis
s-uri-signing-deny-reason	RFCThis

Table 3

[RFC Editor: Please replace RFCThis with the published RFC number for this document.]

6.4. CDNI URI Signing Verification Code

The IANA is requested to create a new "CDNI URI Signing Verification Code" subregistry, in the "Content Delivery Networks Interconnection (CDNI) Parameters" registry. The "CDNI URI Signing Verification Code" namespace defines the valid values associated with the s-uri-signing CDNI Logging Field. The CDNI URI Signing Verification Code is a 3DIGIT value as defined in Section 4.5. Additions to the CDNI URI Signing Verification Code namespace will conform to the "Specification Required" policy as defined in [RFC8126]. Updates to this subregistry are expected to be infrequent.

Value	Reference	Description
000	RFCThis	No signed JWT verification performed
200	RFCThis	Signed JWT verification performed and verified
400	RFCThis	Signed JWT verification performed and rejected because of incorrect signature
401	RFCThis	Signed JWT verification performed and rejected because of Issuer enforcement
402	RFCThis	Signed JWT verification performed and rejected because of Subject enforcement
403	RFCThis	Signed JWT verification performed and rejected because of Audience enforcement
404	RFCThis	Signed JWT verification performed and rejected because of Expiration Time enforcement
405	RFCThis	Signed JWT verification performed and rejected because of Not Before enforcement
406	RFCThis	Signed JWT verification performed and rejected because only one of CDNI Signed Token Transport or CDNI

		Expiration Time Setting present.
407	RFCthis	Signed JWT verification performed and rejected because of JWT ID enforcement
408	RFCthis	Signed JWT verification performed and rejected because of Version enforcement
409	RFCthis	Signed JWT verification performed and rejected because of Critical Extension enforcement
410	RFCthis	Signed JWT verification performed and rejected because of Client IP enforcement
411	RFCthis	Signed JWT verification performed and rejected because of URI Container enforcement
500	RFCthis	Unable to perform signed JWT verification because of malformed URI

Table 4

[RFC Editor: Please replace RFCthis with the published RFC number for this document.]

6.5. CDNI URI Signing Signed Token Transport

The IANA is requested to create a new "CDNI URI Signing Signed Token Transport" subregistry in the "Content Delivery Networks Interconnection (CDNI) Parameters" registry. The "CDNI URI Signing Signed Token Transport" namespace defines the valid values that may be in the Signed Token Transport (cdnistt) JWT claim. Additions to the Signed Token Transport namespace conform to the "Specification Required" policy as defined in [RFC8126]. Updates to this subregistry are expected to be infrequent.

The following table defines the initial Enforcement Information Elements:

Value	Description	RFC
0	Designates token transport is not enabled	RFCthis
1	Designates token transport via cookie	RFCthis
2	Designates token transport via query string	RFCthis

Table 5

[RFC Editor: Please replace RFCthis with the published RFC number for this document.]

6.6. JSON Web Token Claims Registration

This specification registers the following Claims in the IANA "JSON Web Token Claims" registry [IANA.JWT.Claims] established by [RFC7519].

6.6.1. Registry Contents

- * Claim Name: cdniv
- * Claim Description: CDNI Claim Set Version
- * Change Controller: IESG
- * Specification Document(s): Section 2.1.8 of [[this specification]]
- * Claim Name: cdnicrit
- * Claim Description: CDNI Critical Claims Set
- * Change Controller: IESG
- * Specification Document(s): Section 2.1.9 of [[this specification]]
- * Claim Name: cdniip
- * Claim Description: CDNI IP Address
- * Change Controller: IESG
- * Specification Document(s): Section 2.1.10 of [[this specification]]
- * Claim Name: cdniuc
- * Claim Description: CDNI URI Container
- * Change Controller: IESG
- * Specification Document(s): Section 2.1.11 of [[this specification]]
- * Claim Name: cdniets

- * Claim Description: CDNI Expiration Time Setting for Signed Token Renewal
- * Change Controller: IESG
- * Specification Document(s): Section 2.1.12 of [[this specification]]

- * Claim Name: cdnistt
- * Claim Description: CDNI Signed Token Transport Method for Signed Token Renewal
- * Change Controller: IESG
- * Specification Document(s): Section 2.1.13 of [[this specification]]

- * Claim Name: cdnistd
- * Claim Description: CDNI Signed Token Depth
- * Change Controller: IESG
- * Specification Document(s): Section 2.1.14 of [[this specification]]

6.7. Expert Review Guidance

Generally speaking, we should determine the registration has a rational justification and does not duplicate a previous registration. Early assignment should be permissible as long as there is a reasonable expectation that the specification will become formalized. Expert Reviewers should be empowered to make determinations, but generally speaking they should allow new claims that do not otherwise introduce conflicts with implementation or things that may lead to confusion. They should also follow the guidelines of [RFC8126] Section 5 when sensible.

7. Security Considerations

This document describes the concept of URI Signing and how it can be used to provide access authorization in the case of CDNI. The primary goal of URI Signing is to make sure that only authorized UAs are able to access the content, with a CSP being able to authorize every individual request. It should be noted that URI Signing is not a content protection scheme; if a CSP wants to protect the content itself, other mechanisms, such as DRM, are more appropriate.

CDNI URI Signing Signed Tokens leverage JSON Web Tokens and thus guidelines in [RFC8725] are applicable for all JWT interactions.

In general, it holds that the level of protection against illegitimate access can be increased by including more claims in the signed JWT. The current version of this document includes claims for enforcing Issuer, Client IP Address, Not Before time, and Expiration

Time, however this list can be extended with other, more complex, attributes that are able to provide some form of protection against some of the vulnerabilities highlighted below.

That said, there are a number of aspects that limit the level of security offered by URI Signing and that anybody implementing URI Signing should be aware of.

- * **Replay attacks:** A (valid) Signed URI may be used to perform replay attacks. The vulnerability to replay attacks can be reduced by picking a relatively short window between the Not Before time and Expiration Time attributes, although this is limited by the fact that any HTTP-based request needs a window of at least a couple of seconds to prevent sudden network issues from denying legitimate UAs access to the content. One may also reduce exposure to replay attacks by including a unique one-time access ID via the JWT ID attribute (jti claim). Whenever the dCDN receives a request with a given unique ID, it adds that ID to the list of 'used' IDs. In the case an illegitimate UA tries to use the same URI through a replay attack, the dCDN can deny the request based on the already-used access ID. This list should be kept bounded. A reasonable approach would be to expire the entries based on the exp claim value. If no exp claim is present then a simple LRU could be used, however this would allow values to eventually be reused.
- * **Illegitimate clients behind a NAT:** In cases where there are multiple users behind the same NAT, all users will have the same IP address from the point of view of the dCDN. This results in the dCDN not being able to distinguish between different users based on Client IP Address which can lead to illegitimate users being able to access the content. One way to reduce exposure to this kind of attack is to not only check for Client IP but also for other attributes, e.g., attributes that can be found in HTTP headers. However, this may be easily circumvented by a sophisticated attacker.

A shared key distributed between CSP and uCDN is more likely to be compromised. Since this key can be used to legitimately sign a URL for content access authorization, it is important to know the implications of a compromised shared key. While using a shared key scheme can be convenient, this architecture is NOT RECOMMENDED due to the risks associated. It is included for legacy feature parity and is highly discouraged in new implementations.

If a shared key usable for signing is compromised, an attacker can use it to perform a denial-of-service attack by forcing the CDN to evaluate prohibitively expensive regular expressions embedded in a URI Container (cdniuc) claim. As a result, compromised keys should be timely revoked in order to prevent exploitation.

The URI Container (cdniuc) claim can be given a wildcard value. This, combined with the fact that it is the only mandatory claim, means you can effectively make a skeleton key. Doing this does not sufficiently limit the scope of the JWT and is NOT RECOMMENDED. The only way to prevent such a key from being used after it is distributed is to revoke the signing key so it no longer validates.

8. Privacy

The privacy protection concerns described in CDNI Logging Interface [RFC7937] apply when the client's IP address (cdniip) or Subject (sub) is embedded in the Signed URI. For this reason, the mechanism described in Section 2 encrypts the Client IP or Subject before including it in the URI Signing Package (and thus the URL itself).

9. Acknowledgements

The authors would like to thank the following people for their contributions in reviewing this document and providing feedback: Scott Leibrand, Kevin Ma, Ben Niven-Jenkins, Thierry Magnien, Dan York, Bhaskar Bhupalam, Matt Caulfield, Samuel Rajakumar, Iuniana Opreescu, Leif Hedstrom, Gancho Tenev, Brian Campbell, and Chris Lemmons.

10. Contributors

In addition, the authors would also like to make special mentions for certain people who contributed significant sections to this document.

- * Matt Caulfield provided content for the CDNI Metadata Interface section.
- * Emmanuel Thomas provided content for HTTP Adaptive Streaming.
- * Matt Miller provided consultation on JWT usage as well as code to generate working JWT examples.

11. References

11.1. Normative References

- [POSIX.1] "The Open Group Base Specifications Issue 7", IEEE Std 1003.1 2018 Edition, 31 January 2018, <<http://pubs.opengroup.org/onlinepubs/9699919799/>>.
- [RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <<https://www.rfc-editor.org/info/rfc791>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <<https://www.rfc-editor.org/info/rfc5905>>.
- [RFC5952] Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, DOI 10.17487/RFC5952, August 2010, <<https://www.rfc-editor.org/info/rfc5952>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/info/rfc6265>>.
- [RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <<https://www.rfc-editor.org/info/rfc6570>>.
- [RFC6707] Niven-Jenkins, B., Le Faucheur, F., and N. Bitar, "Content Distribution Network Interconnection (CDNI) Problem Statement", RFC 6707, DOI 10.17487/RFC6707, September 2012, <<https://www.rfc-editor.org/info/rfc6707>>.
- [RFC6920] Farrell, S., Kutscher, D., Dannewitz, C., Ohlman, B., Keranen, A., and P. Hallam-Baker, "Naming Things with Hashes", RFC 6920, DOI 10.17487/RFC6920, April 2013, <<https://www.rfc-editor.org/info/rfc6920>>.

- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <<https://www.rfc-editor.org/info/rfc7516>>.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<https://www.rfc-editor.org/info/rfc7519>>.
- [RFC7937] Le Faucheur, F., Ed., Bertrand, G., Ed., Oprescu, I., Ed., and R. Peterkofsky, "Content Distribution Network Interconnection (CDNI) Logging Interface", RFC 7937, DOI 10.17487/RFC7937, August 2016, <<https://www.rfc-editor.org/info/rfc7937>>.
- [RFC8006] Niven-Jenkins, B., Murray, R., Caulfield, M., and K. Ma, "Content Delivery Network Interconnection (CDNI) Metadata", RFC 8006, DOI 10.17487/RFC8006, December 2016, <<https://www.rfc-editor.org/info/rfc8006>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.

11.2. Informative References

- [IANA.JWT.Claims] IANA, "JSON Web Token Claims", <<http://www.iana.org/assignments/jwt>>.

- [MPEG-DASH] ISO, "Information technology -- Dynamic adaptive streaming over HTTP (DASH) -- Part 1: Media presentation description and segment format", ISO/IEC 23009-1:2014, Edition 2, May 2014, <<http://www.iso.org/standard/65274.html>>.
- [RFC6983] van Brandenburg, R., van Deventer, O., Le Faucheur, F., and K. Leung, "Models for HTTP-Adaptive-Streaming-Aware Content Distribution Network Interconnection (CDNI)", RFC 6983, DOI 10.17487/RFC6983, July 2013, <<https://www.rfc-editor.org/info/rfc6983>>.
- [RFC7336] Peterson, L., Davie, B., and R. van Brandenburg, Ed., "Framework for Content Distribution Network Interconnection (CDNI)", RFC 7336, DOI 10.17487/RFC7336, August 2014, <<https://www.rfc-editor.org/info/rfc7336>>.
- [RFC7337] Leung, K., Ed. and Y. Lee, Ed., "Content Distribution Network Interconnection (CDNI) Requirements", RFC 7337, DOI 10.17487/RFC7337, August 2014, <<https://www.rfc-editor.org/info/rfc7337>>.
- [RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <<https://www.rfc-editor.org/info/rfc7517>>.
- [RFC7975] Niven-Jenkins, B., Ed. and R. van Brandenburg, Ed., "Request Routing Redirection Interface for Content Delivery Network (CDN) Interconnection", RFC 7975, DOI 10.17487/RFC7975, October 2016, <<https://www.rfc-editor.org/info/rfc7975>>.
- [RFC8008] Seedorf, J., Peterson, J., Previdi, S., van Brandenburg, R., and K. Ma, "Content Delivery Network Interconnection (CDNI) Request Routing: Footprint and Capabilities Semantics", RFC 8008, DOI 10.17487/RFC8008, December 2016, <<https://www.rfc-editor.org/info/rfc8008>>.
- [RFC8216] Pantos, R., Ed. and W. May, "HTTP Live Streaming", RFC 8216, DOI 10.17487/RFC8216, August 2017, <<https://www.rfc-editor.org/info/rfc8216>>.
- [RFC8725] Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", BCP 225, RFC 8725, DOI 10.17487/RFC8725, February 2020, <<https://www.rfc-editor.org/info/rfc8725>>.

Appendix A. Signed URI Package Example

This section contains three examples of token usage: a simple example with only the required claim present, a complex example which demonstrates the full JWT claims set, including an encrypted Client IP Address (cdniip), and one that uses a Signed Token Renewal.

Note: All of the examples have whitespace added to improve formatting and readability, but are not present in the generated content.

All examples use the following JWK Set [RFC7517]:

```
{ "keys": [
  {
    "kty": "EC",
    "kid": "P5UpOv0eMqlwcxLf7WxIg09JdSYGYFDOWkldueaImf0",
    "use": "sig",
    "alg": "ES256",
    "crv": "P-256",
    "x": "be807S407dzB6I4hTiCUvmxCI6FuxWba1xYB1LSSsZ8",
    "y": "rOGC4vI69g-WF9AGEVI37sNNwbjIzBxSjLvIL7f3RBA"
  },
  {
    "kty": "EC",
    "kid": "P5UpOv0eMqlwcxLf7WxIg09JdSYGYFDOWkldueaImf0",
    "use": "sig",
    "alg": "ES256",
    "crv": "P-256",
    "x": "be807S407dzB6I4hTiCUvmxCI6FuxWba1xYB1LSSsZ8",
    "y": "rOGC4vI69g-WF9AGEVI37sNNwbjIzBxSjLvIL7f3RBA",
    "d": "yaowezrCLTU6yIwUL5RQw67cHgvZeMTLVZXjUGb1A1M"
  },
  {
    "kty": "oct",
    "kid": "f-WbjxBC3dPuI3d24kP2hfvos7Qz688UTi6aB0hN998",
    "use": "enc",
    "alg": "A128GCM",
    "k": "4uFxxV7fhNmrtiah2dlfFg"
  }
]}
```

Note: They are the public signing key, the private signing key, and the shared secret encryption key, respectively. The public and private signing keys have the same fingerprint and only vary by the 'd' parameter that is missing from the public signing key.

A.1. Simple Example

This example is a simple common usage example containing a minimal subset of claims that the authors find most useful.

The JWT Claim Set before signing:

Note: "sha-256;2tderfWPa86Ku7YnzW51YUp7dGUjBS_3SW3ELx4hmWY" is the URL Segment form ([RFC6920] Section 5) of "http://cdni.example/foo/bar".

```
{
  "exp": 1646867369,
  "iss": "uCDN Inc",
  "cdniuc": "hash:sha-256;2tderfWPa86Ku7YnzW51YUp7dGUjBS_3SW3ELx4hmWY"
}
```

The signed JWT:

```
eyJhbGciOiJFUzI1NiIsImtpZCI6IiA1VXBpdjBlTXExd2N4TGZ3V3hJZzA5SmRTWU
dZRRkRPV2tsZHVlYUltZjAifQ.eyJleHAiOiJlE2NDY4NjczNjksImZcyI6InVDRE4gS
W5jIiwiY2RuaXVjIjoiaGFzaDpzaGEtMjU2OzJ0ZGVyZldQYTg2S3U3WW56VzUxWVV
wN2RHVWpCU18zU1czRUx4NGhtV1kifQ.TaNlJM3D96i_9J9Xv1ICO6FUIDFTqt3E2Y
JkEUOLfcH0b89wYRKtbJ9Yj6h_GRgSoZoQ00cps3yUPcWGK3smCw
```

A.2. Complex Example

This example uses all fields except for those dealing with Signed Token Renewal, including Client IP Address (cdniip) and Subject (sub) which are encrypted. This significantly increases the size of the signed JWT token.

JWE for Client IP Address (cdniip) of [2001:db8::1/32]:

```
eyJlbnMiOiJBMTI4R0NNIiwiaWxnIjoizGlyIiwia2lkIjoizilXYmp4QkMzZFB1ST
NkMjRrUDJoZnZvczdRejY4OFVUaTZhQjBoTjk5OCJ9..aUDDFEQBic3nWjOb.bGXWT
HPkntmPCKn0pPPNEQ.iyTttnFyb02YBLqwl_YSjA
```

JWE for Subject (sub) of "UserToken":

```
eyJlbnMiOiJBMTI4R0NNIiwiaWxnIjoizGlyIiwia2lkIjoizilXYmp4QkMzZFB1ST
NkMjRrUDJoZnZvczdRejY4OFVUaTZhQjBoTjk5OCJ9..CLAu80xclc8Bp-Ui.6P1A3
F6ip2Dv.CohdtLLpgBnTvRJQCFuz-g
```

The JWT Claim Set before signing:

```
{
  "aud": "dCDN LLC",
  "sub": "eyJlbmMiOiJBMTI4R0NNIiwiaWxnIjoizGlyIiwia2lkIjoizilXYmp4QkMzZFB1STNkMjRrUDJoZnZvczdRejY4OFVUaTZhQjBoTjk5OCJ9..CLAu80xclc8Bp-Ui.6PlA3F6ip2Dv.CohdtLLpgBnTvRJQCFuz-g",
  "cdniip": "eyJlbmMiOiJBMTI4R0NNIiwiaWxnIjoizGlyIiwia2lkIjoizilXYmp4QkMzZFB1STNkMjRrUDJoZnZvczdRejY4OFVUaTZhQjBoTjk5OCJ9..aUDDFEQBIc3nWjOb.bGXWTHPkntmPCKn0pPPNEQ.iyTttnFyb02YBLqwl_YsJA",
  "cdniv": 1,
  "exp": 1646867369,
  "iat": 1646694569,
  "iss": "uCDN Inc",
  "jti": "5DAafLhZAfhsbe",
  "nbf": 1646780969,
  "cdniuc": "regex:http://cdni\\.example/foo/bar/[0-9]{3}\\..png"
}
```

The signed JWT:

```
eyJhbGciOiJFUzI1NiIsImtpZCI6IiA1VXBpdjBlTXExd2N4TGZ3V3hJZzA5SmRTWUdzRkRPV2tsZHVlYUltZjAiOiJkQ0ROIEExMQyIsInN1YiI6ImV5SmxibUlpT2lkQk1USTRSME5OSWl3aVlXeG5Jam9pWkdseUlpd2lhMmxrSWpvaVppMVhZbXA0UWtNelplGQjFTVE5rTWpSc1VESm9ablP2Y3pkUmVqWTRPRlZVYVRaaFFqQm9Uams1T0NKOS4uQ0xBdTgweGNsYzhCcClVaS42UDF0Y2aXAyRHYuQ29oZHRMTHBnQm5UdlJKUUNgdXotZyIsImNkbmlpcCI6ImV5SmxibUlpT2lkQk1USTRSME5OSWl3aVlXeG5Jam9pWkdseUlpd2lhMmxrSWpvaVppMVhZbXA0UWtNelplGQjFTVE5rTWpSc1VESm9ablP2Y3pkUmVqWTRPRlZVYVRaaFFqQm9Uams1T0NKOS4uYVVEREZFUUJJYzNuV2pPYi5iRlhXVEhQa250bVBDS24wcFBQTKVRLml1VHR0bkZ5Yk8yWUJMcXdsX1lTakEiLCJjZG5pdilI6MSwiZlXhwIjoXNjQ2ODY3MzY5LCJpYXQiojE2NDY2OTQ1NjksImIzcyI6InVDRlE4gSW5jIiwianRpIjoiaURBYWZMaFpBZmhzYmUiLCJpYmYiOiJlY2NDY3ODANjksImNkbml1YyI6InJlZ2V4Omh0dHA6Ly9jZG5pXWZlXGh0bXBSZS9mb28vYmFyLlswLTldeZn9XFWucG5nIn0.IjmVX0uD5MYqArc-M08uEsEeoDQn8kuYXZ9HGHDmDDxsHikT0c8jcX8xYD0z3LzQc1MG65i1kT2sRbZ7isUw8w
```

A.3. Signed Token Renewal Example

This example uses fields for Signed Token Renewal.

The JWT Claim Set before signing:

```
{
  "cdniets": 30,
  "cdnistt": 1,
  "cdnistd": 2,
  "exp": 1646867369,
  "cdniuc": "regex:http://cdni\\.example/foo/bar/[0-9]{3}\\..ts"
}
```

The signed JWT:

```
eyJhbGciOiJFUzI1NiIsImtpZCI6IiA1VXBpdjBlTXExd2N4TGZ3V3hJZzA5SmRTWU
dZRkRPV2tsZHVlYUltZjAifQ.eyJjZG5pZXRzIjozMCI6MSwiY2Ruan0ZCI6MiwiZ
XhwIjoxNjQ2ODY3MzY5LCJjZG5pdWMiOiJyZWdleDpodHRwOi8vY2RuanVxcLmV4YW
lwbGUvZm9vL2Jhcn9bMC05XXszfVxcLnRzIn0.tlPvoKw3BCClw4Lx9PQu7MK6b2IN
55ZoCPSaxovGK0zS53Wpb1MbJBow7G8LiGR39h6-2Iq7PWUSr3MdTIzHYw
```

Once the server verifies the signed JWT it will return a new signed JWT with an updated expiry time (exp) as shown below. Note the expiry time is increased by the expiration time setting (cdniets) value.

The JWT Claim Set before signing:

```
{
  "cdniets": 30,
  "cdnistt": 1,
  "cdnistd": 2,
  "exp": 1646867399,
  "cdniuc": "regex:http://cdni\\.example/foo/bar/[0-9]{3}\\..ts"
}
```

The signed JWT:

```
eyJhbGciOiJFUzI1NiIsImtpZCI6IiA1VXBpdjBlTXExd2N4TGZ3V3hJZzA5SmRTWU
dZRkRPV2tsZHVlYUltZjAifQ.eyJjZG5pZXRzIjozMCI6MSwiY2Ruan0ZCI6MiwiZ
XhwIjoxNjQ2ODY3MzY5LCJjZG5pdWMiOiJyZWdleDpodHRwOi8vY2RuanVxcLmV4YW
lwbGUvZm9vL2Jhcn9bMC05XXszfVxcLnRzIn0.ivY5d_fKGd-OHTpUs8uJUrnHvt-rdu
zu5H4zM7l67pUUAghub53FqDQ5G16jRYX2sY73mA_uLpYDdb-CPTs8FA
```

Authors' Addresses

Ray van Brandenburg
Tiledmedia
Anna van Buerenplein 1
Den Haag
Phone: +31 88 866 7000
Email: ray@tiledmedia.com

Kent Leung
Email: mail4kentl@gmail.com

Phil Sorber
Apple, Inc.
1800 Wazee Street
Suite 410
Denver, CO 80202
United States
Email: sorber@apple.com

Network Working Group
Internet-Draft
Updates: 8006, 8008 (if approved)
Intended status: Standards Track
Expires: 4 September 2022

A. Ryan
Disney Streaming
B. Rosenblum
Vecima
N. Sopher
Qwilt
3 March 2022

CDNI Capacity Capability Advertisement Extensions
draft-ryan-cdni-capacity-insights-extensions-02

Abstract

Open Caching architecture is a use case of Content Delivery Networks Interconnection (CDNI) in which the commercial Content Delivery Network (CDN) is the upstream CDN (uCDN) and the ISP caching layer serves as the downstream CDN (dCDN). This document supplements to the CDNI Capacity Objects defined in RFC 8008 the defined capability objects structure and interface for advertisements and management of a downstream CDN capacity.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 4 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights

and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	2
1.1. Terminology	3
1.2. Requirements Language	3
1.3. Objectives	4
2. CDNI Additional Capability Objects	4
2.1. Telemetry Capability Object	5
2.1.1. Telemetry Source Object	6
2.1.1.1. Telemetry Source Types	7
2.1.1.2. Telemetry Source Metric Object	7
2.1.2. Telemetry Capability Object Serialization	8
2.2. CapacityLimits Capability Object	9
2.2.1. Capacity Limit Object	9
2.2.1.1. Capacity Limit Types	11
2.2.1.2. Capacity Limit Telemetry Source Object	11
2.2.1.3. Capacity Limit Scope Object	12
2.2.2. Capacity Limit Object Serialization	13
3. IANA Considerations	14
3.1. CDNI Payload Types	14
3.1.1. CDNI FCI Telemetry Payload Type	15
3.1.2. CDNI FCI Capacity Limits Payload Type	15
4. Security Considerations	15
5. Acknowledgements	15
6. References	15
6.1. Normative References	15
6.2. Informative References	16
Authors' Addresses	17

1. Introduction

The Streaming Video Alliance [SVA] is a global association that works to solve streaming video challenges in an effort to improve end-user experience and adoption. The Open Caching Working Group [OCWG] of the Streaming Video Alliance [SVA] is focused on the delegation of video delivery requests from commercial CDNs to a caching layer at the ISP's network. Open Caching architecture is a specific use case of CDNI where the commercial CDN is the upstream CDN (uCDN) and the ISP caching layer is the downstream CDN (dCDN). While delegating traffic from one CDN to the other, it is important to make sure that an appropriate amount of traffic is delegated. In order to achieve that, the SVA Open Caching Capacity Insight Specification [OC-CII] defines a feedback mechanism to inform the delegator how much traffic

is appropriate to delegate. The traffic level information provided by that interface will be consumed by entities, such as the Open Caching Request router [OC-RR], to help inform that entity's traffic delegation decisions. This document defines and registers CDNI Payload Types (as defined at section 7.1 of [RFC8006]). These Payload types are used for Capability Objects added to those defined at section 4 of [RFC8008], which are required for the Open Caching Capacity Insights Interface [OC-CII].

For consistency with other CDNI documents this document follows the CDNI convention of uCDN (upstream CDN) and dCDN (downstream CDN) to represent the commercial CDN and ISP caching layer respectively.

This document registers two CDNI Payload Types (section 7.1 of [RFC8006]) for the defined capability objects:

- * Telemetry Payload Type: A payload type for the capability object which defines supported telemetry sources, the metrics made available by that source, and corresponding configuration appropriate to the type of the source (host, port, protocol, etc..).
- * CapacityLimits Payload Type: a payload type for the capability object which defines Capacity Limits based on a set of defined limit types and a mapping from those limits to corresponding telemetry sources for supporting real-time metrics.

1.1. Terminology

The following terms are used throughout this document:

- * CDN - Content Delivery Network

Additionally, this document reuses the terminology defined in [RFC6707], [RFC7336], [RFC8006], [RFC8007], [RFC8008], and [RFC8804]. Specifically, we use the following CDNI acronyms:

- * uCDN, dCDN - Upstream CDN and Downstream CDN respectively (see [RFC7336])

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.3. Objectives

In order to enable information exchange between a uCDN and a dCDN about acceptable levels of traffic to delegate, the following process has been defined:

In normal operation a uCDN will communicate with a dCDN, via an interface, to collect and understand any limits that a dCDN has set forth for traffic delegation from a uCDN. These limits will come in the form of metrics such as bits per second, requests per second, etc.. These limits can be thought of as Not to Exceed (NTE) limits.

The dCDN should provide access to a telemetry source, of near real time metrics, that the uCDN can use to track current usage. The uCDN should compare it's current usage to the limits the dCDN has put forth and adjust traffic delegation decisions accordingly to keep current usage under the specified limits.

In summary, the dCDN will provide the uCDN of limits of how much traffic it should delegate towards the dCDN and then also provide a telemetry source that is coupled to the same scope as the limit, so that the uCDN can use to track its current usage against the advertised limit. Having a limit and a corresponding telemetry source for that limit allows for a non ambiguous definition of what a particular limit means for both the uCDN and dCDN.

Limits that are communicated from the dCDN to the uCDN should be considered valid based on the TTL of the response. The TTL of the response will be provided by the transport mechanism for the response i.e. an HTTP Cache-Control header. The intention is that the limits would have a long lived TTL and would represent a reasonable peak utilization limit that the uCDN should target.

In the event that a dCDN needs to inform a uCDN of an update to a previously communicated limit, the dCDN SHOULD be able to leverage a uCDN callback endpoint to inform the uCDN of adjusted limits. The most common use case for this would be related to dCDN infrastructure issues which reduced the amount of capacity previously advertised as being available.

2. CDNI Additional Capability Objects

Section 5 of [RFC8008] describes the FCI Capability Advertisement Object, which contains a CDNI Capability Object as well as the capability object type (a CDNI Payload Type). The section also defines the Capability Objects per such type. Below we define two additional Capability Objects.

Note: In the following sections, the term "mandatory-to-specify" is used to convey which properties MUST be included when serializing a given capability object. When mandatory-to-specify is defined as "Yes" for an individual property, it means that if the object containing that property is included in an FCI message, then the mandatory-to-specify property MUST also be included.

2.1. Telemetry Capability Object

The Telemetry Capability Object is used to define a list of telemetry sources made available by the dCDN to the uCDN. In this document, Telemetry data is being defined as near real time aggregated metrics of dCDN utilization, such as bits per second egress, and should be specific to the uCDN and dCDN traffic delegation relationship. Telemetry data is uniquely defined by a source id, a metrics name, along with the footprints that are associated with an FCI.Capability advertisement. When defining a Capacity Limit, the meaning of a limit might be considered ambiguous if the uCDN and dCDN are defining current usage via different data sources. Having the dCDN provide a data source defining usage that both itself and the uCDN reference, allows a non ambiguous metric to use when determining current usage and how that compares to a limit. Telemetry data is not only an important component for making informed traffic delegation decisions but also for providing visibility to traffic that has been delegated back through to upstream providers. In situations where there are multiple CDNI delegations, a uCDN will need to incorporate the usage information from any dCDN's it delegated to when itself is asked to provide usage information otherwise, the traffic may seem unaccounted for. An example of this situation is when a Content Provider delegates traffic directly to a CDN, and that CDN decides to further delegate that traffic to a dCDN, if the Content Provider polls the uCDN for traffic usage, if the uCDN does not integrate the Telemetry data of the dCDN it delegated to, any of the traffic the uCDN delegated to it's dCDN would become invisible to the Content Provider.

Property: sources

Description: Telemetry sources made available to the uCDN.

Type: A JSON array of Telemetry Source objects (see Section 2.1.1).

Mandatory-to-Specify: Yes.

2.1.1.1. Telemetry Source Object

The Telemetry Source Object is built of an associated type, a list of exposed metrics, and type-specific configuration data.

Property: id

Description: A unique identifier of a telemetry source.

Type: String.

Mandatory-to-Specify: Yes.

Property: type

Description: A valid telemetry source type. See Section 2.1.1.1.

Type: String.

Mandatory-to-Specify: Yes.

Property: metrics

Description: The metrics exposed by this source.

Type: A JSON array of Telemetry Source Metric objects (see Section 2.1.1.2).

Mandatory-to-Specify: Yes.

Property: configuration

Description: a source-specific representation of the Telemetry source configuration. For the generic source type, this configuration format is defined out-of-band. For other types, the configuration format will be specified in a yet to be defined Telemetry Interface specification. The goal of this element is to allow for forward compatability with a formal Telemetry interface.

Type: A JSON object: TBD

Mandatory-to-Specify: No.

2.1.1.1. Telemetry Source Types

Below are the listed valid telemetry source types. At the time of this draft, the type registry is limit to a single type of Generic. The intention of this type registry is to allow for future extension to reference a yet to be drafted specification for a CDNI Telemetry interface, which would standardize the definition, format, etc of Telemetry data between participants of a CDNI workflow.

Source Type	Description
generic	An object which allows for advertisement of generic datasources

Table 1

2.1.1.2. Telemetry Source Metric Object

The Telemetry Source Metric Object describe the metric to be exposed.

Property: name

Description: An identifier unique within this telemetry source.

Type: String.

Mandatory-to-Specify: Yes.

Property: time-granularity

Description: Represents the time frame that the data represents in seconds. I.e. is this a data set over 5 minutes, one hour, etc..

Type: Integer.

Mandatory-to-Specify: No.

Property: data-percentile

Description: The percentile calculation the data represents, i.e. 50 percentile would equate to the median over the time-granularity. Lack of a data-percentile will mean that the data is the average over the time representation.

Type: Integer.

Mandatory-to-Specify: No.

Property: latency

Description: Time in seconds that the data is behind of real time. This is important to specify to help the uCDN to understand how long it might take to reflect traffic adjustments in the metrics.

Type: Integer.

Mandatory-to-Specify: No.

2.1.2. Telemetry Capability Object Serialization

The following shows an example of Telemetry Capability including 2 metrics for a source, that is scoped to a footprint.

```
"capabilities": [
  {
    "capability-type": "FCI.Telemetry",
    "capability-value": {
      "sources": [
        {
          "id": "capacity_metrics_region1",
          "type": "generic",
          "metrics": [
            {
              "name": "egress_5m",
              "time-granularity": 300,
              "data-percentile": 50,
              "latency": 1500
            },
            {
              "name": "requests_5m",
              ...
            }
          ]
        }
      ]
    }
  },
  "footprints": [
    <footprint objects>
  ]
]
```

2.2. CapacityLimits Capability Object

The Capacity Limits Capability Object enables the dCDN to specify traffic delegation limits to a uCDN within an FCI.Capabilities advertisement. The limits specified by the dCDN will inform the uCDN on how much traffic can be delegated to the dCDN. The limits specified by the dCDN should be considered Not To Exceed (NTE) limits. The limits should be based on near real time telemetry data that the dCDN provides to the uCDN, or in other words, for each limit that is advertised, there should also exist a telemetry source which provides data of current utilization against the particular advertised limit.

Property: limits

Description: A collection of Capacity Limit objects.

Type: A JSON array of CapacityLimit objects (see Section 2.2.1).

Mandatory-to-Specify: Yes.

2.2.1. Capacity Limit Object

A CapacityLimit object is used to represent traffic limits for delegation from the uCDN towards the dCDN. By default the limit object will be scoped to the footprint associated with the FCI capability advertisement encompassing this object. The limit object can contain an optional scoping parameter which will allow specification of a limit for a subset of the encompassing footprint. Limits will be considered using a logical AND, such that a uCDN will need to ensure that all the limits are considered and honored rather than choosing the most specific only. There SHOULD be at least one limit object without an optional scope per capability-value which will define the overall limit for the footprint.

Property: scope

Description: Defines an additional scope requirement for a limit within the FCI footprint context. This CAN be specified if a dCDN would like to specify a more granular limit than what is currently possible via an FCI footprint alone. An example of using the optional scope would be to specify a limit that should be applied to a specific published hostname within a particular FCI footprint. A limit object that does not contain this optional scope, should be considered to apply to the entire encompassing footprint associated with the capability advertisement

Type: Capacity Limit Scope object (see Section 2.2.1.3).

Mandatory-to-Specify: No.

Property: limit-type

Description: The units of maximum-hard and maximum-soft.

Type: String. One of the values listed in Section 2.2.1.1.

Mandatory-to-Specify: Yes.

Property: id

Description: Specifies a unique identifier associated with a limit. The is CAN be used as a relational identifier to a specific Section 2.2.1.

Type: String.

Mandatory-to-Specify: No.

Property: maximum-hard

Description: The maximum unit of capacity that is available for use.

Type: Integer.

Mandatory-to-Specify: Yes.

Property: maximum-soft

Description: A soft limit at which an upstream should consider deducing traffic to prevent hitting the hard limit.

Type: Integer.

Mandatory-to-Specify: No.

Property: current

Description: Specifies the current usage value of the limit. It is not recommended to specify the current usage value inline with the FCI.CapacityLimits advertisements as it will reduce the ability to cache the response. The intended method for providing telemetry data is to reference a Section 2.2.1.2 to poll for the current usage.

Type: Integer.

Mandatory-to-Specify: No.

Property: telemetry-source

Description: Mapping of each a particular limit to a specific metric with relevant real-time data provided by a telemetry source.

Type: Capacity Limit Telemetry Source object (see Section 2.2.1.2).

Mandatory-to-Specify: No.

2.2.1.1. Capacity Limit Types

Below are listed the valid capacity limit types. Additional limits would need to be specified and extended into this list. The values specified here represent the types that were identified as being the most relevant metrics for the purposes of traffic delegation between CDNs.

Limit Type	Units
egress	Bits per second
requests	Requests per second
storage-size	Total bytes
storage-objects	Count
sessions	Count
cache-size	Total bytes

Table 2

2.2.1.2. Capacity Limit Telemetry Source Object

The Capacity Limit Telemetry Source Object refers to a specific metric within a Telemetry Source.

Property: id

Description: Reference to the "id" of a telemetry source defined by a Telemetry Capability object.

Type: String.

Mandatory-to-Specify: Yes.

Property: metric

Description: Reference to the "name" property of a metric defined within a telemetry source of an FCI.Telemetry Capability object.

Type: String.

Mandatory-to-Specify: Yes.

2.2.1.3. Capacity Limit Scope Object

A CapacityLimitScope object is used to define a more granular scope for a limit within the encompassing footprint. The object will define what type of scope is being declared (published host, service ids, etc..) and will then contain an array of items of the type defined (publishedhostA.cdn.com, publishedhostB.cdn.com,..). The scope types are meant to be extendible and reference already established identifiers commonly used in delivery via a CDN. The scope types MUST be mutually understood between the uCDN and dCDN.

Property: type

Description: Defines the type of scope definition.

Type: String. One of the values listed in Section 2.2.1.3.1.

Mandatory-to-Specify: Yes.

Property: values

Description: A collection of values of Section 2.2.1.3.1.

Type: A JSON array of strings of (see Section 2.2.1.3.1).

Mandatory-to-Specify: Yes.

2.2.1.3.1. Capacity Limit Scope Types

Below are listed the valid capacity limit types. Additional limits would need to be specified and extended into this list. The values specified here represent the types that were identified as being the most relevant metrics for the purposes of traffic delegation between CDNs. The most recognized value will be the published-host, while the other values listed correspond to identifiers commonly used with commercial CDN providers that either reference a specific configuration or a logical grouping of configurations.

Scope Type	Description
published-host	CDN-Domain
service-id	an identifier associated with a group of configurations
property-id	an identifier associated with a specific configuration

Table 3

2.2.2. Capacity Limit Object Serialization

The following shows an example of an FCI.CapacityLimits object.

```
"capabilities":[
  {
    "capability-type":"FCI.CapacityLimits",
    "capability-value":{
      "limits":[
        {
          "id":"capacity_limit_region1",
          "limit-type":"egress",
          "maximum-hard":500000000000,
          "maximum-soft":250000000000,
          "telemetry-source":{
            "id":"capacity_metrics_region1",
            "metric":"egress_5m"
          }
        },
        {
          "id":"capacity_limit_region1",
          "scope":{
            "type":"published-host",
```

```

        "values":[
            "serviceA.cdn.example.com"
        ]
    },
    "limit-type":"egress",
    "maximum-hard":200000000000,
    "maximum-soft":100000000000,
    "telemetry-source":{
        "id":"capacity_metrics_region1",
        "metric":"egress_service2_5m"
    }
},
{
    "scope":{
        "type":"service-id",
        "Values":[
            "abcd4567ef9a"
        ]
    },
    "limit-type":"egress",
    "maximum-hard":300000000000,
    "maximum-soft":150000000000,
    "current":200000000000,
    "telemetry-source":{
        "id":"capacity_metrics_region3",
        "metric":"egress_service3_5m"
    }
}
]
},
"footprints":[
    "<footprint objects>"
]
}
]

```

3. IANA Considerations

3.1. CDNI Payload Types

similar to the type definitions described in section 7.1 of [RFC8006] as well as the types described in section 6.1 of [RFC8008].

This document requests the registration of the two additional payload types:

Payload Type	Specification
FCI.Telemetry	RFCthis
FCI.CapacityLimits	RFCthis

Table 4

[RFC Editor: Please replace RFCthis with the published RFC number for this document.]

3.1.1. CDNI FCI Telemetry Payload Type

Purpose: The purpose of this Payload Type is to list the supported telemetry sources and the metrics made available by each source).

Interface: FCI.

Encoding: See section Section 2.1.

3.1.2. CDNI FCI Capacity Limits Payload Type

Purpose: The purpose of this Payload Type is to define Capacity Limits based on a utilization metrics corresponding to telemetry sources provided by the dCDN.

Interface: FCI.

Encoding: See section Section 2.2.

4. Security Considerations

This specification is in accordance with the CDNI Request Routing: Footprint and Capabilities Semantics. As such, it is subject to the security and privacy considerations as defined in Section 8 of [RFC8006] and in Section 7 of [RFC8008] respectively.

5. Acknowledgements

The authors would like to express their gratitude to TBD for TBD (their guidance / contribution / reviews ...)

6. References

6.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8006] Niven-Jenkins, B., Murray, R., Caulfield, M., and K. Ma, "Content Delivery Network Interconnection (CDNI) Metadata", RFC 8006, DOI 10.17487/RFC8006, December 2016, <<https://www.rfc-editor.org/info/rfc8006>>.
- [RFC8007] Murray, R. and B. Niven-Jenkins, "Content Delivery Network Interconnection (CDNI) Control Interface / Triggers", RFC 8007, DOI 10.17487/RFC8007, December 2016, <<https://www.rfc-editor.org/info/rfc8007>>.
- [RFC8008] Seedorf, J., Peterson, J., Previdi, S., van Brandenburg, R., and K. Ma, "Content Delivery Network Interconnection (CDNI) Request Routing: Footprint and Capabilities Semantics", RFC 8008, DOI 10.17487/RFC8008, December 2016, <<https://www.rfc-editor.org/info/rfc8008>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8804] Finkelman, O. and S. Mishra, "Content Delivery Network Interconnection (CDNI) Request Routing Extensions", RFC 8804, DOI 10.17487/RFC8804, September 2020, <<https://www.rfc-editor.org/info/rfc8804>>.

6.2. Informative References

- [OC-CII] Ryan, A., Ed., Rosenblum, B., Goldstein, G., Roskin, R., and G. Bichot, "Open Caching Capacity Insights - Functional Specification (Placeholder before publication)", <<https://www.streamingvideoalliance.org/books/open-cache-capacity-insights-functional-specification/>>.
- [OC-RR] Finkelman, O., Ed., Hofmann, J., Klein, E., Mishra, S., Ma, K., Sahar, D., and B. Zurat, "Open Caching Request Routing - Functional Specification", Version 1.1, 4 October 2019, <<https://www.streamingvideoalliance.org/books/open-cache-request-routing-functional-specification/>>.
- [OCWG] "Open Caching Home Page", <<https://opencaching.streamingvideoalliance.org/>>.

- [RFC6707] Niven-Jenkins, B., Le Faucheur, F., and N. Bitar, "Content Distribution Network Interconnection (CDNI) Problem Statement", RFC 6707, DOI 10.17487/RFC6707, September 2012, <<https://www.rfc-editor.org/info/rfc6707>>.
- [RFC7336] Peterson, L., Davie, B., and R. van Brandenburg, Ed., "Framework for Content Distribution Network Interconnection (CDNI)", RFC 7336, DOI 10.17487/RFC7336, August 2014, <<https://www.rfc-editor.org/info/rfc7336>>.
- [SVA] "Streaming Video Alliance Home Page", <<https://www.streamingvideoalliance.org>>.

Authors' Addresses

Andrew Ryan
Disney Streaming
1211 Avenue of the Americas
New York
 , NY 10036
United States of America
Email: andrew@andrewnryan.com

Ben Rosenblum
Vecima
4375 River Green Pkwy #100
Duluth
 , GA 30096
United States of America
Email: ben@rosenblum.dev

Nir B. Sopher
Qwilt
6, Ha'harash
Hod HaSharon
 4524079
Israel
Email: nir@apache.org

Network Working Group
Internet-Draft
Updates: 8006, 8008 (if approved)
Intended status: Standards Track
Expires: 11 May 2022

N. Sopher
Qwilt
S. Mishra
Verizon
7 November 2021

Content Delivery Network Interconnection (CDNI) Footprint Types:
Subdivision Code and Union
draft-sopher-cdni-footprint-types-extensions-06

Abstract

Open Caching architecture is a use case of Content Delivery Networks Interconnection (CDNI) in which the commercial Content Delivery Network (CDN) is the upstream CDN (uCDN) and the ISP caching layer serves as the downstream CDN (dCDN). This document supplements to the CDNI Metadata Footprint Types defined in RFC 8006. The Footprint Types defined in this document can be used for Footprint objects as part of the Footprint & Capabilities Advertisement interface (FCI) defined in RFC 8008. The defined Footprint Types are derived from requirements raised by Open Caching but are also applicable to CDNI use cases in general.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 11 May 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Terminology	3
1.2. Requirements Language	3
2. CDNI Metadata Additional Footprint Types	3
2.1. CDNI Metadata SubdivisionCode Footprint Type	4
2.1.1. CDNI Metadata SubdivisionCode Data Type	4
2.1.1.1. CDNI Metadata SubdivisionCode Data Type Description	4
2.1.2. CDNI Metadata SubdivisionCode Footprint Type Description	4
2.2. CDNI Metadata FootprintUnion Footprint Type	5
2.2.1. CDNI Metadata FootprintUnion Data Type	6
2.2.2. CDNI Metadata FootprintUnion Footprint Type Description	6
3. IANA Considerations	8
3.1. CDNI Metadata Footprint Types	8
4. Security Considerations	9
5. Acknowledgements	9
6. References	9
6.1. Normative References	9
6.2. Informative References	10
Authors' Addresses	11

1. Introduction

The Streaming Video Alliance [SVA] is a global association that works to solve streaming video challenges in an effort to improve end-user experience and adoption. The Open Caching Working Group [OCWG] of the Streaming Video Alliance [SVA] is focused on the delegation of video delivery requests from commercial CDNs to a caching layer at the ISP's network. Open Caching architecture is a specific use case of CDNI where the commercial CDN is the upstream CDN (uCDN) and the ISP caching layer is the downstream CDN (dCDN). The Open Caching Request Routing Specification [OC-RR] defines the Request Routing process and the interfaces that are required for its provisioning. This document defines and registers CDNI Footprint and Capabilities objects[RFC8008] that are required for Open Caching Request Routing.

For consistency with other CDNI documents this document follows the CDNI convention of uCDN (upstream CDN) and dCDN (downstream CDN) to represent the commercial CDN and ISP caching layer respectively.

This document registers two CDNI Metadata Footprint Types (section 7.2 of [RFC8006]) for the defined objects:

- * SubdivisionCode Footprint Type (e.g. for dCDN advertising a footprint that is specific to a State in the USA)
- * Collection Footprint Type (for dCDN advertising a footprint that consists of a group built from multiple Footprints Types. E.g. both IPv4 and IPv6 client addresses)

1.1. Terminology

The following terms are used throughout this document:

- * CDN - Content Delivery Network

Additionally, this document reuses the terminology defined in [RFC6707], [RFC7336], [RFC8006], [RFC8007], [RFC8008], and [RFC8804]. Specifically, we use the following CDNI acronyms:

- * uCDN, dCDN - Upstream CDN and Downstream CDN respectively (see [RFC7336])

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. CDNI Metadata Additional Footprint Types

Section 5 of [RFC8008] describes the FCI Capability Advertisement Object, which includes an array of CDNI Footprint Objects. Each such object has a footprint-type and a footprint-value, as described in section 4.2.2.2 of [RFC8006]. This document defines additional footprint types, beyond those mentioned in CDNI metadata [RFC8006]. For consistency, this document follows the CDNI notation of uCDN for (the commercial CDN) and dCDN (the ISP caching layer).

2.1. CDNI Metadata SubdivisionCode Footprint Type

Section 4.3.8 of [RFC8006] specifies the "Country Code" footprint type for listing [ISO3166-1] alpha-2 codes. Using Footprint Objects of this type, one can define an FCI Capability Advertisement Object footprint constraints that match a specific country. Here we define the subdivisioncode simple data type, as well as a footprint type allowing the dCDN to define constraints matching geographic areas with better granularity, specifically using the [ISO3166-2] Country Subdivision codes.

2.1.1. CDNI Metadata SubdivisionCode Data Type

The "SubdivisionCode" data type specified in Section 2.1.1.1, describes a country specific subdivision using an [ISO3166-2] code. The data type is added to the list of data types described in section 4.3 of [RFC8006] that are used as properties of CDNI Metadata objects.

2.1.1.1. CDNI Metadata SubdivisionCode Data Type Description

A [ISO3166-2] code in lower case. Each code consists of two parts, separated by a hyphen. The first part is the [ISO3166-1] code of the country. The second part is a string of up to three alphanumeric characters.

Type: String

Example SubdivisionCodes:

"ca-ns"

"us-ny"

2.1.2. CDNI Metadata SubdivisionCode Footprint Type Description

The "SubdivisionCode" simple data type specified in Section 2.1.1, is added to the data types listed as footprint types in section 4.2.2.2 of [RFC8006] .

Below is an adjustment for the example in Section 2.1.1, now embedding a footprint object of type "SubdivisionCode". The Footprint Object in this example creates a constraints matching clients both in Nova-Scotia province of Canada (ISO [ISO3166-2] code "CA-NS"), as well as in the state of New-York In the US (ISO [ISO3166-2] code "US-NY").

```

{
  "capabilities": [
    {
      "capability-type": <CDNI capability object type>,
      "capability-value": <CDNI capability object>,
      "footprints": [
        {
          "footprint-type": "subdivisioncode",
          "footprint-value": ["ca-ns", "us-ny"]
        }
      ]
    }
  ]
}

```

2.2. CDNI Metadata FootprintUnion Footprint Type

As described in section 5 of [RFC8008], the FCI Capability Advertisement Object includes an array of CDNI Footprint Objects. Appendix B of [RFC8008] specifies the semantics of a Footprint Objects array as a multiple, additive, footprint constraints. Meaning, the advertisement of different footprint types narrows the dCDN's candidacy cumulatively.

Sections 4.3.5 and 4.3.6 of [RFC8006] specify the "IPv4CIDR" and "IPv6CIDR" footprint types, respectively, for listing IP addresses blocks. Using Footprint Objects of these types, one can define an FCI Capability Advertisement Object footprint constraints that match IPv4 or IPv6 clients. However, the described "narrowing" semantic of the Footprint Objects array prevents the usage of these objects together in order to create a footprint constraint that matches IPv4 clients together with IPv6 clients.

Below is an example for an attempt at creating an object matching IPv4 clients of subnet "192.0.2.0/24", as well as IPv6 clients of subnet "2001:db8::/32". Such a definition results with an empty list of clients, as the constraints are additives and a client address cannot be both IPv4 and IPv6.

```

{
  "capabilities": [
    {
      "capability-type": <CDNI capability object type>,
      "capability-value": <CDNI capability object>,
      "footprints": [
        {
          "footprint-type": "ipv4cidr",
          "footprint-value": ["192.0.2.0/24"]
        },
        {
          "footprint-type": "ipv6cidr",
          "footprint-value": ["2001:db8::/32"]
        }
      ]
    }
  ]
}

```

To overcome the described limitation, and allow a list of footprint constraints that matches both IPv4 and IPv6 client addresses, we introduce below the "FootprintUnion" footprint type. This footprint type allows the collection of multiple footprint-objects into a unified object. It is useful for resolving the above limitation, as well as for unifying footprints of additional types such as countrycode and subdivisioncode.

2.2.1. CDNI Metadata FootprintUnion Data Type

The "FootprintUnion" data type is based on the Footprint Object already defined in section 4.2.2.2 of [RFC8006]. It includes a footprint-type property and a footprint-value array values.

2.2.2. CDNI Metadata FootprintUnion Footprint Type Description

The "footprintunion" data type specified in Section 2.2.1, is added to the data types listed as footprint types in section 4.2.2.2 of [RFC8006].

Below is an adjustment for the example in Section 2.2.1, now embedding a footprint object of type "footprintunion".

```

{
  "capabilities": [
    {
      "capability-type": <CDNI capability object type>,
      "capability-value": <CDNI capability object>,
      "footprints": [
        {
          "footprint-type": "footprintunion",
          "footprint-value": [
            {
              "footprint-type": "ipv4cidr",
              "footprint-value": ["192.0.2.0/24"]
            },
            {
              "footprint-type": "ipv6cidr",
              "footprint-value": ["2001:db8::/32"]
            }
          ]
        }
      ]
    }
  ]
}

```

An additional example is the collection of countrycode and subdivisioncode based footprint objects. In the example below we create a constraint covering autonomous system 64496 within the US (ISO [ISO3166-1] alpha-2 code "US"), as well as Nova-Scotia province of Canada (ISO [ISO3166-2] code "CA-NS").

```

{
  "capabilities": [
    {
      "capability-type": <CDNI capability object type>,
      "capability-value": <CDNI capability object>,
      "footprints": [
        {
          "footprint-type": "asn",
          "footprint-value": ["as64496"]
        },
        {
          "footprint-type": "footprintunion",
          "footprint-value": [
            {
              "footprint-type": "countrycode",
              "footprint-value": ["us"]
            },
            {
              "footprint-type": "subdivisioncode",
              "footprint-value": ["ca-ns"]
            }
          ]
        }
      ]
    }
  ]
}

```

3. IANA Considerations

3.1. CDNI Metadata Footprint Types

As described in section 7.2 of [RFC8006] , the "CDNI Metadata Footprint Types" subregistry was created within the "Content Delivery Network Interconnection (CDNI) Parameters" registry. The created namespace defines the valid values for Footprint Object Types, and is already populated with the types described in Section 4.2.2.2 of [RFC8006] .

This document requests the registration of the two additional footprint type as defined in Section 2.2 and Section 2.1 :

Footprint Type	Description	Specification
FCI.subdivisioncode	ISO 3166-2 Country or Subdivision Code	RFCthis
FCI.footprintunion	Footprint Object as specified in [RFC8006]	RFCthis

Table 1

[RFC Editor: Please replace RFCthis with the published RFC number for this document.]

4. Security Considerations

This specification is in accordance with the CDNI Request Routing: Footprint and Capabilities Semantics. As such, it is subject to the security and privacy considerations as defined in Section 8 of [RFC8006] and in Section 7 of [RFC8008] respectively.

5. Acknowledgements

The authors would like to express their gratitude to Ori Finkelman and Kevin J. Ma for their guidance and reviews throughout the development of this document.

6. References

6.1. Normative References

[ISO3166-1]

ISO, "Codes for the representation of names of countries and their subdivisions -- Part 1: Country code", ISO 3166-1:2020, Edition 4, August 2020, <<https://www.iso.org/standard/72482.html>>.

[ISO3166-2]

ISO, "Codes for the representation of names of countries and their subdivisions -- Part 2: Country subdivision code", ISO 3166-2:2020, Edition 4, August 2020, <<https://www.iso.org/standard/72483.html>>.

[RFC2119]

Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

- [RFC8006] Niven-Jenkins, B., Murray, R., Caulfield, M., and K. Ma, "Content Delivery Network Interconnection (CDNI) Metadata", RFC 8006, DOI 10.17487/RFC8006, December 2016, <<https://www.rfc-editor.org/info/rfc8006>>.
- [RFC8007] Murray, R. and B. Niven-Jenkins, "Content Delivery Network Interconnection (CDNI) Control Interface / Triggers", RFC 8007, DOI 10.17487/RFC8007, December 2016, <<https://www.rfc-editor.org/info/rfc8007>>.
- [RFC8008] Seedorf, J., Peterson, J., Previdi, S., van Brandenburg, R., and K. Ma, "Content Delivery Network Interconnection (CDNI) Request Routing: Footprint and Capabilities Semantics", RFC 8008, DOI 10.17487/RFC8008, December 2016, <<https://www.rfc-editor.org/info/rfc8008>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8804] Finkelman, O. and S. Mishra, "Content Delivery Network Interconnection (CDNI) Request Routing Extensions", RFC 8804, DOI 10.17487/RFC8804, September 2020, <<https://www.rfc-editor.org/info/rfc8804>>.

6.2. Informative References

- [OC-RR] Finkelman, O., Ed., Hofmann, J., Klein, E., Mishra, S., Ma, K., Sahar, D., and B. Zurat, "Open Caching - Request Routing Functional Specification", Version 1.1, 4 October 2019, <<https://www.streamingvideoalliance.org/books/open-cache-request-routing-functional-specification/>>.
- [OCWG] "Open Caching Home Page", <<https://www.streamingvideoalliance.org/technical-groups/open-caching/>>.
- [RFC6707] Niven-Jenkins, B., Le Faucheur, F., and N. Bitar, "Content Distribution Network Interconnection (CDNI) Problem Statement", RFC 6707, DOI 10.17487/RFC6707, September 2012, <<https://www.rfc-editor.org/info/rfc6707>>.
- [RFC7336] Peterson, L., Davie, B., and R. van Brandenburg, Ed., "Framework for Content Distribution Network Interconnection (CDNI)", RFC 7336, DOI 10.17487/RFC7336, August 2014, <<https://www.rfc-editor.org/info/rfc7336>>.

[SVA] "Streaming Video Alliance Home Page",
 <<https://www.streamingvideoalliance.org>>.

Authors' Addresses

Nir B. Sopher
Qwilt
6, Ha'harash
Hod HaSharon 4524079
Israel

Email: nir@apache.org

Sanjay Mishra
Verizon
13100 Columbia Pike
Silver Spring, MD 20904
United States of America

Email: sanjay.mishra@verizon.com