Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 1, 2012

A. Bierman
YumaWorks
M. Bjorklund
Tail-f Systems

May 30, 2012

YANG-API Protocol
draft-bierman-netconf-yang-api-00

Abstract

This document describes a RESTful protocol that provides a programmatic interface over HTTP for accessing data defined in YANG, using the datastores defined in NETCONF.

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 http://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 December 1, 2012.

Table of Contents

1. Introduction

There is a need for standard mechanisms to allow WEB applications to access the configuration data, operational data, and data-model specific RPC operations within a networking device, in a modular and extensible manner.

This document describes a RESTful protocol called YANG-API, running over HTTP [RFC2616], for accessing data defined in YANG [RFC6020], using datastores defined in NETCONF [RFC6241].

The NETCONF protocol defines configuration datastores and a set of Create, Retrieve, Update, Delete (CRUD) operations that can be used to access these datastores. The YANG language defines the syntax and semantics of datastore content and operational data. RESTful operations are used to access the hierarchical data within a datastore.

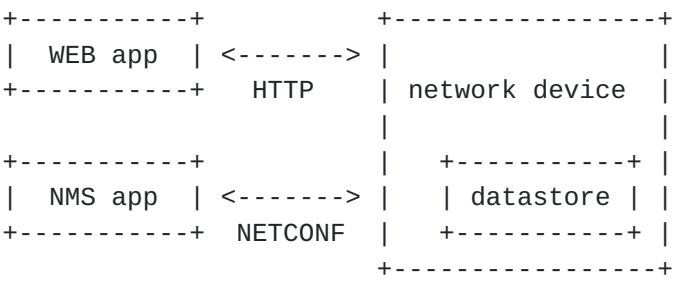
A RESTful API can be created that provides CRUD operations on a NETCONF datastore containing YANG-defined data. This can be done in a simplified manner, compatible with HTTP and RESTful design principles. Since NETCONF protocol operations are not relevant, the user should not need any prior knowledge of NETCONF in order to use the RESTful API.

Configuration data and state data are exposed as resources that can be retrieved with the GET method. Resources representing configuration data can be modified with the DELETE, PATCH, POST, and PUT methods. Data-model specific RPC operations defined with the YANG "rpc" statement can be invoked with the POST method.

The framework and meta-model used for a RESTful API does not need to mirror those used by the NETCONF protocol. It just needs to be compatible with NETCONF. A simplified framework and protocol is needed that aligns with the three NETCONF datastores (candidate, running, startup). A simplified yet more powerful transaction model is needed that exposes the proper functionality without over-restricting server design.

The RESTful API is not intended to replace NETCONF, but rather provide an additional simplified interface that follows RESTful principles and is compatible with a resource-oriented device abstraction. It is expected that applications that need the full feature set of NETCONF such as notifications will continue to use NETCONF.

The following figure shows the system components:



1.1. Terminology

The keywords "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].

1.1.1. NETCONF

The following terms are defined in [RFC6241]:

- candidate configuration datastore
- client
- configuration data
- datastore
- configuration datastore
- protocol operation
- running configuration datastore
- server
- startup configuration datastore
- state data
- user

1.1.2. HTTP

The following terms are defined in [RFC2616]:

- entity tag
- fragment
- header line
- message body
- method
- path
- query
- request URI
- response body

1.1.3. YANG

The following terms are defined in [RFC6020]:

- container
- data node
- key leaf
- leaf
- leaf-list
- list
- presence container (or P-container)
- RPC operation
- non-presence container (or NP-container)
- ordered-by system
- ordered-by user

1.1.4. Terms

The following terms are used within this document:

- API resource: a resource with the media type "application/vnd.yang.api+xml" or ""application/vnd.yang.api+json".
- data resource: a resource with the media type "application/vnd.yang.data+xml" or "application/vnd.yang.data+json".
- datastore resource: a resource with the media type "application/vnd.yang.datastore+xml" or "application/vnd.yang.datastore+json"
- edit operation: a YANG-API operation on a data resource using the POST, PUT, PATCH, or DELETE method.
- operation: the conceptual YANG-API operation for a message, derived from the method, request URI, headers, and message body.
- operation resource: a resource with the media type "vnd.yang.operation+xml" or "vnd.yang.operation+json"
- optional key: a key leaf for a YANG list data node, which MAY be omitted by the client when an instance of the list is created.
- query parameter: a parameter (and its value if any), encoded within the query portion of the request URI.
- resource: a conceptual object representing a manageable component within a device.
- retrieval request: an operation using the GET or HEAD methods.
- target resource: the resource that is associated with a particular message, identified by the "path" component of the request URI.
- transaction resource: a resource with the media type "vnd.yang.transaction+xml" or "vnd.yang.transaction+json"

1.2. Overview

This document defines the YANG-API protocol, a RESTful API for accessing conceptual datastores containing data defined with YANG language. YANG-API provides an application framework and meta-model, using HTTP operations.

The YANG-API resources are accessed via a set of URIs defined in this document. The set of YANG modules supported by the server will

determine the additional data model specific operations and top-level data node resources available on the server.

Not all YANG-API defined resources are mandatory-to-implement. The server implementor may choose the specific editing model and persistence model that is supported. The specific subset is identified and accessible via 3 capability fields. Refer to Section 5.1.1 for more details.

1.2.1. Resource URI Map

The URI hierarchy for the YANG-API resources consists of an entry point and up to 6 top-level resources and/or fields. Refer to Section 5 for details on each URI.

```
/yang-api
   /capabilities
      /edit-model
      /persist-model
      /transaction-model
   /datastore
      /<top-level-data-nodes> (config=true or false)
   /modules
      /module
   /operations
      /lock-datastore
      /save-datastore
      /unlock-datastore
      /<operations>
   /transaction
      /<transaction-id>
         /commit
         /datastore
            /<top-level-data-nodes> (config=true)
         /discard-changes
         /exclusive-mode
         /update
         /validate
   /version
```

1.2.2. YANG-API Message Examples

The examples within this document use the non-normative example YANG module defined in Section 12.

This section shows some typical YANG-API message exchanges.

In these examples, the server capabilities are as follows:

- the edit-model is "direct"
- the persist-model is "manual"
- the transaction-model is "none"

1.2.2.1. Retrieve the Top-level API Resource

By default, when a resource is retrieved, all of its fields are returned, but none (if any) of the nested resources are returned. Also, the default encoding is JSON. Data resources are encoded according to the encoding rules in [I-D.lhotka-yang-json].

The client starts by retrieving the top-level API resource, using the entry point URI "/yang-api".

```
GET /yang-api HTTP/1.1
Host: example.com
```

The server might respond as follows. The "module" lines below are split for display purposes only:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:01:00 GMT
Server: example-server
Content-Type: application/vnd.yang.api+json

{
  "yang-api": {
    "capabilities": {
      "edit-model": "direct",
      "persist-model": "automatic",
      "transaction-model": "none"
    },
    "modules": {
      "module": [
        "urn:ietf:params:xml:ns:yang:ietf-yang-api
           ?module=ietf-yang-api&revision=2012-05-27",
        "example.com?module=example-jukebox
           &revision=2012-05-30"
      ]
    },
    "version": "1.0"
  }
}
```

To request that the response content to be encoded in XML, the "Accept" header can be used, as in this example request:

```
   GET /yang-api HTTP/1.1
   Host: example.com
   Accept: application/vnd.yang.api+xml
```

An alternate approach is provided using the "format" query parameter, as in this example request:

```
   GET /yang-api?format=xml HTTP/1.1
   Host: example.com
```

The server will return the same response either way, which might be as follows :

```
   HTTP/1.1 200 OK
   Date: Mon, 23 Apr 2012 17:01:00 GMT
   Server: example-server
   Cache-Control: no-cache
   Pragma: no-cache
   Content-Type: application/vnd.yang.api+xml


   <yang-api>
     <capabilities>
       <edit-model>direct</edit-model>
       <persist-model>automatic</persist-model>
       <transaction-model>none</transaction-model>
     </capabilities>
     <modules> <!-- wrapped for display only -->
       <module>urn:ietf:params:xml:ns:yang:ietf-yang-api
          ?module=ietf-yang-api
          &amp;revision=2012-05-27</module>
       <module>example.com?module=example-jukebox
           &amp;revision=2012-05-30</module>
     </modules>
     <version>1.0</version>
   </yang-api>
```

Refer to Section 3.3 for details on the GET operation.

1.2.2.2. Create New Data Resources

To create a new "jukebox" resource, the client might send:

```
   POST /yang-api/datastore/jukebox HTTP/1.1
   Host: example.com
```

If the resource is created, the server might respond:

```
   HTTP/1.1 201 Created
   Date: Mon, 23 Apr 2012 17:01:00 GMT
   Server: example-server
   Location: http://example.com/yang-api/datastore/jukebox
   Last-Modified: Mon, 23 Apr 2012 17:01:00 GMT
   ETag: b3a3e673be2
```

To create a new "artist" resource within the "jukebox" resource, the client might send the following request, Note that the arbitrary integer "index" is not provided, since it is an optional key:

```
   POST /yang-api/datastore/jukebox/artist HTTP/1.1
   Host: example.com
   Content-Type: application/vnd.yang.data+json

   {
     "artist" : {
       "name" : "The Foo Fighters"
     }
   }
```

If the resource is created, the server might respond:

```
   HTTP/1.1 201 Created
   Date: Mon, 23 Apr 2012 17:02:00 GMT
   Server: example-server
   Location: http://example.com/yang-api/datastore/jukebox/artist/1
   Last-Modified: Mon, 23 Apr 2012 17:02:00 GMT
   ETag: b3830f23a4c
```

To create a new "album" resource for this artist within the "jukebox" resource, the client might send the following request,

```
   POST /yang-api/datastore/jukebox/artist/1/album HTTP/1.1
   Host: example.com
   Content-Type: application/vnd.yang.data+json

   {
     "album" : {
       "name" : "Wasting Light",
       "genre" : "example-jukebox:Alternative",
       "year" : 2012
     }
   }
```

If the resource is created, the server might respond as follows. Note that the "Location" header line is wrapped for display purposes only:

```
HTTP/1.1 201 Created
Date: Mon, 23 Apr 2012 17:03:00 GMT
Server: example-server
Location: http://example.com/yang-api/datastore/
  jukebox/artist/1/album/Wasting%20Light
Last-Modified: Mon, 23 Apr 2012 17:03:00 GMT
ETag: b8389233a4c
```

Refer to Section 3.4 for details on the POST operation.

1.2.2.3. Replace an Existing Data Resource

Note: replacing a resource is a fairly drastic operation. The PATCH operation is often more appropriate.

The album sub-resource is re-added here for example purposes only. To replace the "artist" resource contents, the client might send:

```
PUT /yang-api/datastore/jukebox/artist/1 HTTP/1.1
Host: example.com
If-Match: b3830f23a4c
Content-Type: application/vnd.yang.data+json

{
  "artist" : {
    "name" : "Foo Fighters",
    "album" : {
      "name" : "Wasting Light",
      "genre" : "example-jukebox:Alternative",
      "year" : 2012
    }
  }
}
```

If the resource is updated, the server might respond:

```
HTTP/1.1 204 No Content
Date: Mon, 23 Apr 2012 17:04:00 GMT
Server: example-server
Last-Modified: Mon, 23 Apr 2012 17:04:00 GMT
ETag: b27480aeda4c
```

Refer to Section 3.5 for details on the PUT operation.

1.2.2.4. Patch an Existing Data Resource

To replace just the "year" field in the "album" resource, the client might send:

```
PATCH /yang-api/datastore/jukebox/artist/1/album/
   Wasting%20Light/year HTTP/1.1
Host: example.com
If-Match: b8389233a4c
Content-Type: application/vnd.yang.data+json

{ "year" : 2011 }
```

If the resource is updated, the server might respond:

```
HTTP/1.1 204 No Content
Date: Mon, 23 Apr 2012 17:49:30 GMT
Server: example-server
Last-Modified: Mon, 23 Apr 2012 17:49:30 GMT
ETag: b2788923da4c
```

Refer to Section 3.6 for details on the PATCH operation.

1.2.2.5. Delete an Existing Data Resource

To delete a resource such as the "album" resource, the client might send:

```
DELETE /yang-api/datastore/jukebox/artist/1/album/
   Wasting%20Light HTTP/1.1
Host: example.com
```

If the resource is deleted, the server might respond:

```
HTTP/1.1 204 No Content
Date: Mon, 23 Apr 2012 17:49:40 GMT
Server: example-server
```

Refer to Section 3.7 for details on the DELETE operation.

1.2.2.6. Invoke a Data Model Specific Operation

To invoke a global operation, such as the "save-datastore" operation resource, the POST operation is used. A client might send a "save-datastore" request as follows:

```
POST /yang-api/operations/save-datastore HTTP/1.1
Host: example.com
```

The server might respond:

```
HTTP/1.1 204 No Content
Date: Mon, 23 Apr 2012 17:50:00 GMT
```

```
   Server: example-server
```

Refer to Section 3.9 for details on using the POST operation with operation resources.

2. Framework

The YANG-API protocol defines a framework that can be used to implement a common API for configuration management. This section describes the components of the YANG-API framework.

2.1. Message Model

The YANG-API protocol uses HTTP entities for messages. A single HTTP message corresponds to a single protocol operation in NETCONF. A message can perform a single task on a single resource, such as retrieving a resource or editing a resource. It cannot be used to combine multiple tasks. The client cannot provide multiple (possibly unrelated) edit operations within a single request, like the NETCONF <edit-config> protocol operation.

2.2. Resource Model

The YANG-API protocol operates on a hierarchy of resources, starting with the top-level API resource itself. Each resource represents a manageable component within the device.

A resource can be considered a collection of conceptual data and the set of allowed operations on that data. It can contain child nodes that are either "fields" or other resources. The child resource types and operations allowed on them are data-model specific.

A resource has its own media type identifier, represented by the "Content-Type" header in the HTTP response message. A resource can contain zero or more fields and zero or more resources. A resource can be created and deleted independently of its parent resource, as long as the parent resource exist.

A field is a child node defined within a resource. A field can contain zero or more fields and zero or more resources. A field cannot be created and deleted independently of its parent resource.

All YANG-API resources and fields are defined in this document except datastore contents and RPC operations. These resource types are defined with YANG data definition statements and the "rpc" statement. A default mapping is defined to differentiate sub-resources from fields within data resources.

2.2.1. YANG-API Resource Types

The YANG-API protocol defines some application specific media types to identify each of the available resource types. The following table summarizes the purpose of each resource.

Resource	Media Type
API	application/vnd.yang.api
Datastore	application/vnd.yang.datastore
Data	application/vnd.yang.data
Operation	application/vnd.yang.operation
Transaction	application/vnd.yang.transaction

YANG-API Media Types

These resources are described in Section 5.

2.2.2. Resource Discovery

A client SHOULD start by retrieving the top-level API resource, using the entry point URI "/yang-api".

The YANG-API protocol does not include a resource discovery mechanism. Instead, the definitions within the YANG modules advertised by the server are used to construct a predictable operation or data resource identifier.

The "depth" query parameter can be used to control how many descendant levels should be included when retrieving sub-resources. This parameter can be used with the GET operation to discover sub-resources within a particular resource.

Refer to Section 3.8.2 for more details on the "depth" parameter.

2.3. Datastore Model

A conceptual "unified datastore" is used to simplify resource management for the client. The YANG-API datastore is a combination of the running configuration and any non-configuration data supported by the device. By default only configuration data is returned by a GET operation on the datastore contents.

The underlying NETCONF datastores can be used to implement the unified datastore, but the server design is not limited to the exact datastore procedures defined in NETCONF.

Instead of a separate candidate configuration datastore to use as a globally shared scratchpad to collect edits, an optional transaction mechanism is provided (see Section 2.4).

Instead of a separate startup configuration datastore, a simplified

persistence model is used (see Section 2.3.4).

2.3.1. Content Model

The YANG-API protocol operates on a conceptual datastore defined with the YANG data modeling language. The server lists each YANG module it supports in the "/yang-api/modules/module" field in the top-level API resource type, using the YANG module capability URI format defined in RFC 6020.

The conceptual datastore contents and data-model-specific operations are identified by the set of YANG module capability URIs. All YANG-API content identified as either a data resource or an operation resource is defined with the YANG language.

The classification of data as configuration or non-configuration is derived from the YANG "config" statement. Data retrieval with the GET operation can be filtered in several ways, including the "config" parameter to retrieve configuration or non-configuration data.

The classification of data as a resource or field within a resource is derived from the rules specified in Section 5.3.2.

Data ordering behavior is derived from the YANG "ordered-by" statement. Editing mechanisms are provided to allow list or leaf-list resources to be inserted or moved in the same manner as NETCONF, and defined in YANG.

The server is not required to maintain system ordered data in any particular persistent order. The server SHOULD maintain the same data ordering for system ordered data until the next reboot or termination of the server.

2.3.2. Editing Model

The YANG-API datastore editing model is compatible with the NETCONF protocol but not exactly the same.

If the running configuration datastore is written directly, then each change takes place right away. This can have a negative impact on network behavior if multiple inter-related resources need to be edited at once, in order to achieve the new desired network state.

To address this problem, an optional transaction mechanism is defined (similar to the NETCONF :candidate capability) to allow multiple edits to be collected and validated, before being applied all-or-nothing to the running configuration datastore.

Private and shared transactions are supported. If the server uses a single shared datastore resource, or if multiple clients use the same private transaction, then it is often useful to know if the data resources being edited have changed (relative to the resource versions the client thinks are on the server).

This can be achieved in YANG-API using the edit collision detection mechanisms described in Section 2.3.2.2. If a collision is detected, then the client can retrieve the resource before proceeding with the edit.

2.3.2.1. Edit Operation Discovery

Sometimes a server does not implement every operation for every resource. Sometimes data model requirements cause a node to implement a subset of the edit operations. For example, a server may not allow modification of a particular configuration data node after the parent resource has been created.

The OPTIONS operation can be used to identify which operations are supported by the server for a particular resource. For example, if the server will allow a data resource node to be created then the POST operation will be returned in the response.

2.3.2.2. Edit Collision Detection

Two "edit collision detection" mechanisms are provided in YANG-API, for datastore and data resources.

- timestamp: the last change time is maintained and the "Last-Modified" and "Date" headers are returned in the response for a retrieval request. The "If-Unmodified-Since" header can be used in edit operation requests to cause the server to reject the request if the resource has been modified since the specified timestamp.

- entity tag: a unique opaque string is maintained and the "ETag" header is returned in the response for a retrieval request. The "If-Match" header can be used in edit operation requests to cause the server to reject the request if the resource entity tag does not match the specified value.

Note that the server is only required to maintain these fields for a datastore resource, not for individual data resources.

Example:

In this example, the server just supports the mandatory datastore

last-changed timestamp. The client has previously retrieved the "Last-Modified" header and has some value cached to provide in the following request to replace a list entry with key value "11":

```
PATCH /yang-api/datastore/jukebox/artist/1/album/
  Wasting%20Light/year HTTP/1.1
Host: example.com
Accept: application/vnd.yang.data+json
If-Unmodified-Since: Mon, 23 Apr 2012 17:01:00 GMT
Content-Type: application/vnd.yang.data+json

{ "year" : "2011" }
```

In this example the datastore resource has changed since the time specified in the "If-Unmodified-Since" header. The server might respond:

```
HTTP/1.1 304 Not Modified
Date: Mon, 23 Apr 2012 19:01:00 GMT
Server: example-server
Last-Modified: Mon, 23 Apr 2012 17:45:00 GMT
ETag: b34aed893a4c
```

2.3.3. Locking Model

Datastore locking is needed in order to allow a client to make several changes to the running configuration datastore contents in sequence, without disturbance from other clients.

The "lock-datastore" and "unlock-datastore" operations MUST be supported by the server. These correspond to the global locks defined in NETCONF. Only the running configuration datastore can be locked and unlocked in this manner. If the datastore is locked, then direct edits and transaction commits by other clients will fail.

The editing model allows for concurrent transactions to occur without locking, using the transaction "update" operation. This is similar to the "discard-changes" operation, except that the running configuration datastore is merged into the current transaction datastore (instead of replacing the contents). If the "update" cannot be done, a conflict error report is generated so the client can manually resolve the differences.

A client can request exclusive write access when a transaction resource is created. This is comparable to a global lock on the candidate configuration datastore if the server "transaction-model" capability field is set to "shared". In this case, the creation of the new transaction resource will fail if another exclusive

transaction already exists.

There is no partial datastore locking (i.e., per-resource or per YANG data node) at this time. Explicit partial locks are difficult to use and easy to misuse. Transactions are easier for a client to use, and allow more server design freedom as well.

2.3.4. Persistence Model

A client must be aware of how the server saves configuration data to non-volatile storage, so the server advertises its persistence model (either "automatic" or "manual").

If manual persistence of the running configuration datastore is required, then the "persist" operation MUST be supported by the server and MUST be used by the client to save the running configuration datastore contents to non-volatile storage.

If automatic persistence of the running configuration datastore is supported by the server, then the non-volatile storage of configuration changes is handled automatically by the server, and the "persist" operation MUST NOT be supported by the server.

2.3.5. Defaults Model

NETCONF has a rather complex defaults handling model for leafs. YANG-API attempts to avoid this complexity by restricting the operations that can be applied to a resource and fields within that resource.

The GET method returns only nodes that exist, which will be determined by the server. There is no mechanism for the client to ask the server for the default values that would be used for any nodes not present, but some default value is in use by the server. If a leaf definition has a default value, and the leaf has not been given a value yet, the server SHOULD NOT return any value for the leaf in the response for a GET operation.

2.4. Transaction Model

The "/yang-api/transaction" resource will be present if the server supports transactions. If so, the server MUST support at least one transaction at a time and MAY support multiple concurrent transactions, either by one client or multiple clients.

The "/yang-api/capabilities/transaction-model" field in the top-level API resource identifies which type of transactions the server supports, either "none", "shared", or "private". If shared, then all

clients are sharing the same "/yang-api/transaction/<id>/datastore" resource. If "private" then each instance of a "/yang-api/transaction/<id>/datastore" resource is independent of each another.

There are a small number of operations supported for a transaction resource.

- commit: attempt to commit the transaction.
- discard-changes: replace the contents of the transaction datastore to the contents of the running configuration datastore.
- update: merge the contents of the running configuration datastore into the transaction datastore.
- validate: Run commit validation tests against the running configuration datastore contents, according to section 8.3.3 of [RFC6020].

Refer to Section 5.5.4 for more details on these operations.

2.5. Extensibility Model

The YANG-API protocol is designed to be extensible for datastore content and data-model specific RPC operations. New RPC operations can be added without changing the entry point if they are optional and do not alter any existing operations.

Separate namespaces for each YANG module are used. Content encoded in XML will indicate the module using the "namespace" URI value in the YANG module. Content encoded in JSON will indicate the module using the module name specified in the YANG module. JSON encoding rules for module namespaces are specified in [I-D.lhotka-yang-json].

2.6. Versioning Model

The version of a resource instance is identified with an entity tag, as defined by HTTP. The version identifiers in this section apply to the version of the schema definition of a resource. There are two types of schema versioning information used in the YANG-API protocol:

- the YANG-API protocol version
- data and operation resource definition versions

The protocol version is identified by the string used for the well-known URI entry point "/yang-api". This would be changed (e.g., "/yang-api2") if non-backward compatible changes are ever needed.

Minor version changes that do not break backward-compatibility will not cause the entry point to change.

The API "yang-api/version" field can be used by the client to identify the exact version of the YANG-API protocol implemented by the server. This value will include the complete YANG-API protocol version. The "/yang-api" entry point will only change (e.g., "/yang-api2") if non-backward compatible changes are made to the protocol. The "/yang-api/version" field MUST be updated every time the protocol specification is republished.

The resource definition version for a data or operation resource is a date string, which is the revision date of the YANG module that defines the resource. The resource version for all other resource types is a numeric string, defined by the "/yang-api/version" field.

2.7. Retrieval Filtering Model

There are four types of filtering for retrieval of data resources in the YANG-API protocol.

- conditional all-or-nothing: use some conditional test mechanism in the request headers and retrieve either a complete "200 OK" response if the condition is met, or a "304 Not Modified" Status-Line if the condition is not met.

- data classification: request configuration or non-configuration data.

- subset: request a subset of all possible instances of a list or leaf-list data resource.

- filter: request a subset of all possible descendant nodes within the target resource. The "select" query parameter can be used for this purpose.

Refer to Section 5.3.4 for details on data retrieval filtering.

2.8. Access Control Model

The YANG-API protocol provides no granular access control for any content except for operation and data resources. The NETCONF Access Control Model (NACM) is defined in [RFC6536]. There is a specific mapping between YANG-API operations and NETCONF edit operations, defined in Table 1. The resource path also needs to be converted internally by the server to the corresponding YANG instance-identifier. Using this information, the server can apply the NACM access control rules to YANG-API messages.

The server MUST NOT allow any operation to any resources that the client is not authorized to access.

3. Operations

The YANG-API protocol uses HTTP methods to identify the CRUD operation requested for a particular resource or field within a resource. The following table shows how the YANG-API operations relate to NETCONF protocol operations:

YANG-API	NETCONF
OPTIONS	none
HEAD	none
GET	<get-config>, <get>
POST	<edit-config> (operation="create")
PUT	<edit-config> (operation="replace")
PATCH	<edit-config> (operation="merge")
DELETE	<edit-config> (operation="delete")

Table 1: CRUD Operations in YANG-API

The NETCONF "remove" operation attribute is not supported by the HTTP DELETE method. The resource must exist or the DELETE operation will fail.

This section defines the YANG-API protocol usage for each HTTP method.

3.1. OPTIONS

The OPTIONS method is sent by the client to discover which methods are supported by the server for a specific resource, or field within a resource. It is supported for all media types. Note that implementation of this operation is part of HTTP, and this section does not introduce any additional requirements.

The request MUST contain a request URI that contains at least the entry point component.

The server will return a "Status-Line" header containing "204 No Content". and include the "Allow" header in the response. This header will be filled in, based on the target resource media type. Other headers MAY also be included in the response.

Example 1:

A client might request the methods supported for a data resource called "library"

```
   OPTIONS /yang-api/datastore/jukebox/library HTTP/1.1
   Host: example.com
```

The server might respond (for a config=true list):

```
   HTTP/1.1 204 No Content
   Date: Mon, 23 Apr 2012 17:01:00 GMT
   Server: example-server
   Allow: OPTIONS,HEAD,GET,POST,PUT,PATCH,DELETE
```

Example 2:

A client might request the methods supported for a non-configuration leaf within a data resource:

```
   OPTIONS /yang-api/datastore/jukebox/library/
       song-count HTTP/1.1
   Host: example.com
```

The server might respond:

```
   HTTP/1.1 204 No Content
   Date: Mon, 23 Apr 2012 17:02:00 GMT
   Server: example-server
   Allow: OPTIONS,HEAD,GET
```

Example 3:

A client might request the methods supported for an operation resource called "play":

```
   OPTIONS /yang-api/operations/play HTTP/1.1
   Host: example.com
```

The server might respond:

```
   HTTP/1.1 204 No Content
   Date: Mon, 23 Apr 2012 17:02:00 GMT
   Server: example-server
   Allow: POST
```

3.2. HEAD

The HEAD operation is sent by the client to retrieve just the headers that would be returned for the comparable GET operation, without the response body. The HTTP HEAD method is used for this operation. It is supported for all resource types, except operation resources.

The request MUST contain a request URI that contains at least the entry point component.

The same query parameters supported by the GET operation are supported by the HEAD operation. For example, the "select" query parameter can be used to specify a field within the target resource.

The access control behavior is enforced as if the method was GET instead of HEAD. The server MUST respond the same as if the method was GET instead of HEAD, except that no response body is included.

Example:

The client might request the response headers for the default (JSON) representation of the "library" resource:

```
HEAD /yang-api/datastore/jukebox/library HTTP/1.1
Host: example.com
```

The server might respond:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:02:40 GMT
Server: example-server
Content-Type: application/vnd.yang.data+json
Cache-Control: no-cache
Pragma: no-cache
ETag: a74eefc993a2b
Last-Modified: Mon, 23 Apr 2012 11:02:14 GMT
```

3.3. GET

The GET operation is sent by the client to retrieve data and meta-data for a resource or field within a resource. The HTTP GET method is used for this operation. It is supported for all resource types, except operation resources. The request MUST contain a request URI that contains at least the entry point component.

The following query parameters are supported by the GET operation:

Name	Section	Description
config	3.8.1	Request either configuration or non-configuration data
depth	3.8.2	Control the depth of a retrieval request
format	3.8.3	Request either JSON or XML content in the response

| select | 3.8.6 | Specify a field within the target resource |
+--------+---------+---+

GET Query Parameters

The server MUST NOT return any data resources or fields within any data resources for which the user does not have read privileges.

If the user is not authorized to read any portion of the target resource, an error response containing a "403 Forbidden" Status-Line is returned to the client.

If the user is authorized to read some but not all of the target resource, the unauthorized content is omitted from the response message body, and the authorized content is returned to the client.

Example:

The client might request the response headers for a JSON representation of the "library" resource:

```
GET /yang-api/datastore/jukebox/library/artist/
  1/album HTTP/1.1
Host: example.com
```

The server might respond:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:02:40 GMT
Server: example-server
Content-Type: application/vnd.yang.data+json
Cache-Control: no-cache
Pragma: no-cache
ETag: a74eefc993a2b
Last-Modified: Mon, 23 Apr 2012 11:02:14 GMT

{
  "album" : {
    "name" : "Wasting Light",
    "genre" : "example-jukebox:Alternative",
    "year" : 2011
  }
}
```

3.4. POST

The POST operation is sent by the client for various reasons. The HTTP POST method is used for this purpose. The request MUST contain a request URI that contains a target resource that identifies one of the following resource types:

Type	Description
Data	Create a configuration data resource
Operation	Invoke RPC operation
Transaction	Create a new transaction

Resource Types that Support POST

The following query parameters are supported by the POST operation:

Name	Section	Description
insert	3.8.4	Specify where to insert a resource
point	3.8.5	Specify the insert point for a resource

POST Query Parameters

If the POST operation succeeds, a "200 OK" Status-Line is returned if there is no response message body, and a "204 No Content" Status-Line is returned if there is a response message body.

If the user is not authorized to invoke the target (operation) resource, or create the target resource, an error response containing a "403 Forbidden" Status-Line is returned to the client. All other error responses are handled according to the procedures defined in Section 6.

3.5. PUT

The PUT operation is sent by the client to replace the target resource.

The HTTP PUT method is used for this purpose. The request MUST contain a request URI that contains a target resource that identifies the data resource to replace.

The following query parameters are supported by the PUT operation:

Name	Section	Description
insert	3.8.4	Specify where to move a resource
point	3.8.5	Specify the move point for a resource

PUT Query Parameters

If the PUT operation succeeds, a "200 OK" Status-Line is returned, and there is no response message body.

If the user is not authorized to replace the target resource an error response containing a "403 Forbidden" Status-Line is returned to the client. All other error responses are handled according to the procedures defined in Section 6.

3.6. PATCH

The PATCH operation uses the HTTP PATCH method defined in [RFC5789] to provide a "merge" editing mode for data resources. Instead of replacing all or part of the target resource, the supplied values are merged into the target resource.

If the PATCH operation succeeds, a "200 OK" Status-Line is returned, and there is no response message body.

If the user is not authorized to alter the target resource an error response containing a "403 Forbidden" Status-Line is returned to the client. All other error responses are handled according to the procedures defined in Section 6.

3.7. DELETE

The DELETE operation uses the HTTP DELETE method to delete the target resource.

If the DELETE operation succeeds, a "200 OK" Status-Line is returned, and there is no response message body.

If the user is not authorized to delete the target resource then an error response containing a "403 Forbidden" Status-Line is returned to the client. All other error responses are handled according to the procedures defined in Section 6.

3.8. Query Parameters

Each YANG-API operation allows zero or more query parameters to be present in the request URI. Refer to Section 3 for details on the query parameters used in the definition of each operation.

Query parameters can be given in any order. Each parameter can appear zero or one time. A default value may apply if the parameter is missing.

This section defines all the YANG-API query parameters.

3.8.1. "config" Parameter

The "config" parameter is used to specify whether configuration or non-configuration data is requested.

This parameter is only supported for the GET and HEAD methods. It is also only supported if the target resource is a data resource.

```
syntax: config= true | false
default: true
```

Example:

This example request by the client would retrieve only the non-configuration data nodes that exist within the second-level "library" resource.

```
GET /yang-api/datastore/jukebox/library?config=false HTTP/1.1
Host: example.com
Accept: application/vnd.yang.data+xml
```

The server might respond:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:01:30 GMT
Server: example-server
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/vnd.yang.data+json

{
  "library" : {
     "artist-count" : 42,
     "album-count" : 59,
     "song-count" : 374
  }
```

```
    }
```

3.8.2. "depth" Parameter

The "depth" parameter is used to specify the number of nest levels returned in a response for a GET operation. A nest-level consists of the target resource and any child nodes which are optional data nodes (anyxml, leaf, or leaf-list). A non-presence container is transparent when determining the nest level. A child node (which is not a non-presence container) within a non-presence container is used to determine the nest-level.

The start level is determined by the target resource for the operation.

```
    syntax: depth=<range: 1..max> | unbounded
    default: 1
```

Example:

This example operation would retrieve 2 levels of configuration data nodes that exist within the top-level "jukebox" resource.

```
GET /yang-api/datastore/jukebox?depth=2 HTTP/1.1
Host: example.com
Accept: application/vnd.yang.data+json
```

The server might respond:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:11:30 GMT
Server: example-server
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/vnd.yang.data+json

{
  "jukebox" : {
    "library" : {
      "artist" : {
        "index" : 1,
        "name" : "Foo Fighters"
      }
    },
    "player" : {
      "gap" : 0.5
    }
  }
```

```
    }
```

3.8.3. "format" Parameter

The "format" parameter is used to specify the format of any content returned in the response. Note that the "Accept" header MAY be used instead of this parameter to identify the format desired in the response. For example:

```
GET /yang-api/datastore/routing HTTP/1.1
Host: example.com
Accept: application/vnd.yang.data+xml
```

This example request would retrieve only the configuration data nodes that exist within the top-level "routing" resource, and retrieve them in XML encoding instead of JSON encoding.

The "format" parameter is only supported for the GET and HEAD methods. It is supported for all YANG-API media types.

```
    syntax: format= xml | json
    default: json
```

Example:

```
GET /yang-api/datastore/routing?format=xml HTTP/1.1
Host: example.com
```

This example URI would retrieve only the configuration data nodes that exist within the top-level "routing" resource, and retrieve them in XML encoding instead of JSON encoding.

3.8.4. "insert" Parameter

The "insert" parameter is used to specify how a resource should be inserted (or moved) within the user-ordered list or leaf-list data resource.

This parameter is only supported for the POST and PUT methods. It is also only supported if the target resource is a data resource, and that data represents a YANG list or leaf-list that is ordered by the user, not the system.

If the values "before" or "after" are used, then a "point" parameter for the insertion parameter MUST also be present.

```
    syntax: insert= first | last | before | after
    default: last
```

Example:

Request from client:

```
POST /yang-api/datastore/jukebox/library/artist/1/album
   /Wasting%20Light/song?insert=first HTTP/1.1
Host: example.com
Content-Type: application/vnd.yang.data+json

{
  "song" : {
     "name" : "Bridge Burning",
     "location" : "/media/bridge_burning.mp3",
     "format" : "MP3",
     "length" : 286
  }
}
```

Response from server: 201 status

```
HTTP/1.1 201 Created
Date: Mon, 23 Apr 2012 13:01:20 GMT
Server: example-server
Last-Modified: Mon, 23 Apr 2012 13:01:20 GMT

Location: http://example.com/yang-api/datastore/jukebox
   /library/artist/1/album?Wasting%20Light/song/1
ETag: eeeada438af
```

3.8.5. "point" Parameter

The "point" parameter is used to specify the insertion point for a data resource that is being created or moved within a user ordered list or leaf-list. It is ignored unless the "insert" query parameter is also present, and has the value "before" or "after".

This parameter contains the instance identifier of the resource, or field within a resource, to be used as the insertion point for a POST or PUT operation. It is encoded according to the rules defined in Section 5.3.1. There is no default for this parameter.

```
syntax: point= <instance-identifier of insertion point node>
```

Example:

In this example, the client is moving an existing "song" resource within an "album" resource after another song. The request URI is split for display purposes only.

Request from client:

```
PUT /yang-api/datastore/jukebox/library/artist/1/album/
  Wasting%20Light/song/2?insert=after
  &point=/yang-api/datastore/jukebox/library/artist/1/
  album/Wasting%20Light/song/4   HTTP/1.1
Host: example.com
```

Response from server:

```
HTTP/1.1 204 No Content
Date: Mon, 23 Apr 2012 13:01:20 GMT
Server: example-server
Last-Modified: Mon, 23 Apr 2012 13:01:20 GMT
ETag: abcada438af
```

3.8.6. "select" Parameter

The "select" query parameter is used to specify an expression which can represent a subset of all data nodes within the target resource. It contains a relative path expression, using the target resource as the context node.

It is supported for all resource types except operation resources. The contents are encoded according to the "api-select" rule defined in Section 5.3.1. This parameter is only allowed for GET and HEAD operations.

[FIXME: the syntax of the select string is still TBD; XPath, schema-identifier, regular expressions, something else]

Refer to Section 1.2.2 for example request messages using the "select" parameter.

3.9. RPC Operations

The YANG-API also allows RPC operations to be invoked using the POST method. The media type "vnd.yang.operation+xml" or "vnd.yang.operation+json" MUST be used in the "Content-Type" field in the message header.

The following datastore specific operations are defined:

Operation	Description
lock-datastore	Lock the /yang-api/datastore resource for writing
save-datastore	Save the /yang-api/datastore resource to NV-storage
unlock-datastore	Unlock the /yang-api/datastore resource

YANG-API Datastore Operations

Refer to Section 5.2 for details on these operations.

The following transaction specific operations are defined:

Operation	Description
commit	Commit the transaction to the running config
discard-changes	replace transaction data with current running config
update	merge current running config into transaction data
validate	validate transaction datastore

YANG-API Transaction Operations

Refer to Section 5.5 for details on these operations.

3.9.1. Data Model Specific Operations

Data model specific operations are supported. The syntax and semantics of these operations exactly correspond to the YANG rpc statement definition for the operation.

Any input for a RPC operation is encoded in an element called "input", which corresponds to the <input> element in a NETCONF message. The child nodes of the "input" element are encoded according to the data definition statements in the input section of the rpc statement.

Any output for a RPC operation is encoded in an element called "output", which corresponds to the <rpc-reply> element in a NETCONF message. The child nodes of the "output" element are encoded according to the data definition statements in the output section of the rpc statement.

4. Messages

This section describes the messages that are used in the YANG-API protocol.

4.1. Request URI Structure

Resources are represented with URIs following the structure for generic URIs in [RFC3986].

A YANG-API operation is derived from the HTTP method and the request URI, using the following conceptual fields:

```
     <OP> /yang-api/<path>?<query>#<fragment>


       ^      ^       ^        ^         ^
       |      |       |        |         |
    method entry  resource  query    fragment

       M       M        O        O         I


    M=mandatory, O=optional, I=ignored


    <text> replaced by client with real values
```

- method: the HTTP method identifying the YANG-API operation requested by the client, to act upon the target resource specified in the request URI. YANG-API operation details are described in Section 3.

- entry: the well-known YANG-API entry point ("/yang-api").

- resource: the path expression identifying the resource that is being accessed by the operation. If this field is not present, then the target resource is the API itself, represented by the media type "vnd.yang.api".

- query: the set of parameters associated with the YANG-API message. These have the familiar form of "name=value" pairs. There is a specific set of parameters defined, although the server MAY choose to support additional parameters not defined in this document.

- fragment: This field is not used by the YANG-API protocol.

The client SHOULD NOT assume the final structure of a URI path for a

resource. Instead, existing resources can be discovered with the GET operation. When new resources are created by the client, a "Location" header is returned, which identifies the path of the newly created resource. The client MUST use this exact path identifier to access the resource once it has been created.

The "target" of an operation is a resource. The "path" field in the request URI represents the target resource for the operation.

4.2. Message Headers

There are several HTTP header lines utilized in YANG-API messages. Messages are not limited to the HTTP headers listed in this section.

HTTP defines which header lines are required for particular circumstances. Refer to each operation definition section in Section 3 for examples on how particular headers are used.

There are some request headers that are used within YANG-API, usually applied to data resources. The following tables summarize the headers most relevant in YANG-API message requests:

Name	Description
Accept	Response Content-Types that are acceptable
Content-Type	The media type of the request body
Host	The host address of the server
If-Match	Only perform the action if the entity matches ETag
If-Modified-Since	Only perform the action if modified since time
If-Range	Only retrieve range if resource unchanged
If-Unmodified-Since	Only perform the action if un-modified since time
Range	Specify a range of data resource entries

YANG-API Request Headers

The following tables summarize the headers most relevant in YANG-API message responses:

Name	Description
Allow	Valid actions when 405 error returned
Content-Type	The media type of the response body
Date	The date and time the message was sent
ETag	An identifier for a specific version of a resource
Last-Modified	The last modified date and time of a resource
Location	The resource identifier for a newly created resource

YANG-API Response Headers

4.3. Message Encoding

YANG-API messages are encoded in HTTP according to RFC 2616. The "utf-8" character set is used for all messages. YANG-API message content is sent in the HTTP message body.

Content is encoded in either JSON or XML format.

XML encoding rules for data nodes are defined in [RFC6020]. The same encoding rules are used for all XML content. XML attributes are not used and will be ignored if present in an XML-encoded message.

JSON encoding rules are defined in [I-D.lhotka-yang-json]. Special encoding rules are needed to handle multiple module namespaces and provide consistent data type processing.

Request input content encoding format is identified with the Content-Type header. This field MUST be present if message input is sent by the client.

Response output content encoding format is identified with the Accept header, the "format" query parameter, or if neither is specified, the request input encoding format is used. If there was no request input, then the default output encoding is JSON. File extensions encoded in the request are not used to identify format encoding.

4.4. Return Status

Each message represents some sort of resource access. An HTTP "Status-Line" header line is returned for each request. If a 4xx or 5xx range status code is returned in the Status-Line, then the error information will be returned in the response, according to the format defined in Section 6.1.

4.5. Message Caching

Since the datastore contents change at unpredictable times, responses from a YANG-API server generally SHOULD NOT be cached.

The server SHOULD include a "Cache-Control" header in every response that specifies whether the response should be cached. A "Pragma" header specifying "no-cache" MAY also be sent in case the "Cache-Control" header is not supported.

Instead of using HTTP caching, the client SHOULD track the "ETag" and/or "Last-Modified" headers returned by the server for the datastore resource (or data resource if the server supports it).

A retrieval request for a resource can include headers such as "If-None-Match" or "If-Modified-Since" which will cause the server to return a "304 Not Modified" Status-Line if the resource has not changed.

The client MAY use the HEAD operation to retrieve just the message headers, which SHOULD include the "ETag" and "Last-Modified" headers, if this meta-data is maintained for the target resource.

5. Resources

The resources used in the YANG-API protocol are identified by the "path" component in the request URI. Each operation is performed on a target resource.

5.1. API Resource (/yang-api)

The API resource contains the state and access points for the YANG-API features.

It is the top-level resource and has the media type "application/vnd.yang.api+xml" or "application/vnd.yang.api+json". It is accessible through the well-known URI "/yang-api".

This resource has the following fields:

Field Name	Description
capabilities	Server capabilities
datastore	Link to "datastore" resource
operations	Global operations
modules	YANG modules
transaction	Link to "transaction" resource

YANG-API Resource Fields

5.1.1. /yang-api/capabilities

This mandatory field represents the YANG-API server capabilities. The child nodes are read-only fields that MUST NOT change while the server is running, but MAY change after a reboot.

Example:

To retrieve just the YANG-API capabilities, the client might send the following request:

```
GET /yang-api?select=capabilities HTTP/1.1
Host: example.com
```

The server might respond:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:10:00 GMT
Server: example-server
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/vnd.yang.api+json

{
  "yang-api": {
    "capabilities": {
      "edit-model": "transaction",
      "persist-model": "manual",
      "transaction-model": "private"
    }
  }
}
```

5.1.1.1. /yang-api/capabilities/edit-model

The "edit-model" capability field is used to identify the editing model used by the server. There are 4 supported models:

- none: A server within a constrained device MAY choose to provide a read-only implementation, in which case no editing model is supported.

- direct: A device MAY allow the running configuration datastore to only be modified directly, and therefore will not support transactions.

- transaction: A device SHOULD support the transaction mechanism defined in this document. Datastore edits are collected in the transaction datastore and applied to the running configuration datastore with the "commit" operation.

- both: A device MAY support both the direct and transaction editing models, by allowing direct editing operations on the datastore and supporting the transaction mechanism.

The server SHOULD support 1 of the 2 datastore editing models, and MAY support both datastore editing models. If both are supported, then the client can decide which editing model it prefers.

This field is encoded with the rules for a "bits" data type, using the following leaf definition:

```
leaf edit-model {
  config false;
  type bits {
    bit direct {
      description
        "Direct writing to the datastore resource is allowed.";
    }
    bit transaction {
      description
        "Writing to the datastore via transactions is allowed.";
    }
  }
}
```

There is no default. The server MUST set zero, one, or both of these bits in the "edit-model" capability field.

5.1.1.2. /yang-api/capabilities/persist-model

The "persist-model" capability field is used to identify the persistence model used by the server. There are two supported models:

- automatic: The server will automatically save the running configuration datastore contents to non-volatile storage.

- manual: The client must manually save the running configuration datastore contents to non-volatile storage.

This field is encoded with the rules for an "enumeration" data type, using the following leaf definition:

```
leaf persist-model {
  config false;
  type enumeration {
    enum automatic {
      description
        "The server will automatically save the
         running configuration";
    }
    enum manual {
      description
        "The client must manually save the running
         configuration";
    }
  }
}
```

There is no default. The server MUST set one enumeration value in the "persist-model" capability field.

5.1.1.3. /yang-api/capabilities/transaction-model

The "transaction-model" capability field is used to identify the transaction model used by the server. There are 3 supported models:

- none: The server does not support transactions.
- shared: All clients are sharing the same conceptual transaction datastore (similar to NETCONF :candidate capability).
- private: Each transaction datastore resource is independent of one another.

This field is encoded with the rules for an "enumeration" data type, using the following leaf definition:

```
leaf transaction-model {
  config false;
  type enumeration {
    enum none {
      description
        "The server does not support transactions.";
    }
    enum shared {
      description
        "The server supports a shared transaction datastore
         resource.";
    }
    enum private {
      description
        "The server supports a private transaction datastore
         resource.";
    }
  }
}
```

There is no default. The server MUST set one enumeration value in the "transaction-model" capability field.

5.1.2. /yang-api/datastore

This mandatory resource represents the running configuration datastore and any non-configuration data available. It may be retrieved and edited directly or indirectly (via transactions). It cannot be created or deleted by the client. This resource type is

defined in Section 5.2.

5.1.3. /yang-api/operations

This optional field provides access to the global datastore and data-model specific RPC operations supported by the server. The datastore operation resources will be available depending on the server capabilities. If the server does not support any global operations, then this field SHOULD NOT not be present.

There are 3 global operations defined by YANG-API.

- lock-datastore
- save-datastore
- unlock-datastore

Any data-model specific global operations derived from YANG modules supported by the server will also be available through child node resources within the "operations" field. The YANG-API defined global operations are described in this section.

5.1.3.1. /yang-api/operations/lock-datastore

The "lock-datastore" operation resource is used to lock the datastore resource represented by the URI "/yang-api/datastore". It behaves exactly the same as the NETCONF <lock> operation on the running configuration datastore.

If the operation succeeds, a "204 No Content" value in the "Status-Line" is sent in the response. If the operation fails, the appropriate error code is set according to the rules in Section 6, and the error report is sent in the response, according to the format defined in Section 6.1.

The "lock-datastore" operation does not take any parameters. The YANG "rpc" statement definition for this operation is defined in Section 8.

Example:

The client might request a lock on the running configuration datastore as follows:

```
POST /yang-api/operations/lock-datastore HTTP/1.1
Host: example.com
```

If the operation succeeds the server might respond:

```
HTTP/1.1 204 No Content
Date: Mon, 23 Apr 2012 17:03:00 GMT
Server: example-server
```

If the operation fails the server might respond:

```
HTTP/1.1 204 No Content
Date: Mon, 23 Apr 2012 17:03:00 GMT
Server: example-server
```

5.1.3.2. /yang-api/operations/save-datastore

The "save-datastore" operation resource is used to save the datastore resource represented by the URI "/yang-api/datastore" to non-volatile storage. It behaves exactly the same as the NETCONF <copy-config> operation when used to copy the running configuration datastore to the startup configuration datastore.

If the operation succeeds, a "204 No Content" value in the "Status-Line" is sent in the response. If the operation fails, the appropriate error code is set according to the rules in Section 6, and the error report is sent in the response, according to the format defined in Section 6.1.

The "save-datastore" operation does not take any parameters. The YANG "rpc" statement definition for this operation is defined in Section 8.

Example:

The client might request that the running configuration datastore be saved in non-volatile storage as follows:

```
POST /yang-api/operations/save-datastore HTTP/1.1
Host: example.com
```

If the operation succeeds the server might respond:

```
HTTP/1.1 204 No Content
Date: Mon, 23 Apr 2012 17:03:00 GMT
Server: example-server
```

If the operation fails the server might respond:

```
HTTP/1.1 204 No Content
Date: Mon, 23 Apr 2012 17:03:00 GMT
```

```
   Server: example-server
```

5.1.3.3. /yang-api/operations/unlock-datastore

The "unlock-datastore" operation resource is used to unlock the datastore resource represented by the URI "/yang-api/datastore". It behaves exactly the same as the NETCONF <unlock> operation on the running configuration datastore.

If the operation succeeds, a "204 No Content" value in the "Status-Line" is sent in the response. If the operation fails, the appropriate error code is set according to the rules in Section 6, and the error report is sent in the response, according to the format defined in Section 6.1.

The "unlock-datastore" operation does not take any parameters. The YANG "rpc" statement definition for this operation is defined in Section 8.

Example:

The client might release a lock on the running configuration datastore as follows:

```
   POST /yang-api/operations/unlock-datastore HTTP/1.1
   Host: example.com
```

If the operation succeeds the server might respond:

```
   HTTP/1.1 204 No Content
   Date: Mon, 23 Apr 2012 17:03:00 GMT
   Server: example-server
```

If the operation fails the server might respond:

```
   HTTP/1.1 204 No Content
   Date: Mon, 23 Apr 2012 17:03:00 GMT
   Server: example-server
```

5.1.4. /yang-api/modules

This mandatory field contains the identifiers for the YANG data model modules supported by the server. There MUST be exactly one instance of this field.

The server MUST maintain a last-modified timestamp for this field, and return the "Last-Modified" header when this field is retrieved with the GET or HEAD methods.

5.1.4.1. /yang-api/modules/module

This mandatory field contains one URI string for each YANG data model module supported by the server. There MUST be an instance of this field for every YANG module that is accessible via an operation resource or a data resource.

The server MAY maintain a last-modified timestamp for each instance of this resource, and return the "Last-Modified" header when this resource is retrieved with the GET or HEAD methods. If not supported then the timestamp for the parent "modules" field MUST NOT be used instead.

The contents of this field are encoded with the "uri" derived type from the "ietf-iana-types" modules in [RFC6021].

There are additional encoding requirements for this field. The URI MUST follow the YANG module capability URI formatting defined in section 5.6.4 of [RFC6020].

5.1.4.2. Retrieval Example

In this example the client is retrieving the modules field from the server in the default JSON format:

```
GET /yang-api?select=modules HTTP/1.1
Host: example.com
Accept: application/vnd.yang.api+json
```

The server might respond as follows. Note that the content below is split across multiple lines for display purposes only:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:01:00 GMT
Server: example-server
Cache-Control: no-cache
Pragma: no-cache
Last-Modified: Sun, 22 Apr 2012 01:00:14 GMT
Content-Type: application/vnd.yang.api+json

{
  "yang-api": {
    "modules": {
      "module": [
        "example.com?module=foo&revision=2012-01-02",
        "example.com?module=bar&revision=2011-10-10"
        "example.com?module=itf&revision=2011-10-10
            &feature=restore"
      ]
    }
  }
}
```

5.1.5. /yang-api/transaction

This optional resource will be supported if the server implements transactions, identified by the "/yang-api/capabilities/edit-model" field in the API resource. It is used to allow one or more individual edits to be applied (all-or-nothing) to the running configuration datastore, and to facilitate concurrent editing transactions with a mechanism to update the transaction datastore contents with the latest running configuration datastore contents.

This resource is defined in Section 5.5.

5.1.6. /yang-api/version

This mandatory field identifies the specific version of the YANG-API protocol implemented by the server.

The same server-wide response MUST be returned each time this field is retrieved. It is assigned by the server when the server is started. The server MUST return the value "1.0" for this version of the YANG-API protocol.

This field is encoded with the rules for an "enumeration" data type, using the following leaf definition:

```
    leaf version {
      config false;
      type enum {
        enum "1.0" {
          description
            "Version 1.0 of the YANG-API protocol.";
        }
      }
    }
```

5.2. Datastore Resource

A datastore resource represents the conceptual root of a tree of data resources.

The server MUST maintain a last-modified timestamp for this resource, and return the "Last-Modified" header when this resource is retrieved with the GET or HEAD methods. Only changes to configuration data resources within the datastore affect this timestamp.

The server SHOULD maintain a resource entity tag for this resource, and return the "ETag" header when this resource is retrieved with the GET or HEAD methods. The resource entity tag SHOULD be changed to a new previously unused value if changes to any configuration data resources within the datastore are made.

A datastore resource can be retrieved with the GET operation, to retrieve either configuration data resources or non-configuration data resources within the datastore. The "config" query parameter is used to choose between them. Refer to Section 3.8.1 for more details.

The depth of the subtrees returned in retrieval operations can be controlled with the "depth" query parameter. The number of nest levels, starting at the target resource, can be specified, or an unlimited number can be returned. Refer to Section 3.8.2 for more details.

A datastore resource cannot be written directly with any edit operation. Only the configuration data resources within the datastore resource can be edited.

5.3. Data Resource

A data resource represents a YANG data node that is a descendant node of a datastore resource.

For configuration data resources, the server MAY maintain a last-

modified timestamp for the resource, and return the "Last-Modified" header when it is retrieved with the GET or HEAD methods.

For configuration data resources, the server MAY maintain a resource entity tag for the resource, and return the "ETag" header when it is retrieved as the target resource with the GET or HEAD methods. The resource entity tag SHOULD be changed to a new previously unused value if changes to the resource or any configuration field within the resource is altered.

A data resource can be retrieved with the GET operation, to retrieve either configuration data resources or non-configuration data resources within the target resource. The "config" query parameter is used to choose between them. Refer to Section 3.8.1 for more details.

The depth of the subtrees returned in retrieval operations can be controlled with the "depth" query parameter. The number of nest levels, starting at the target resource, can be specified, or an unlimited number can be returned. Refer to Section 3.8.2 for more details.

A configuration data resource can be altered by the client with some of all of the edit operations, depending on the target resource and the specific operation. Refer to Section 3 for more details on edit operations.

5.3.1. Encoding YANG Instance Identifiers in the Request URI

In YANG, data nodes are named with an absolute XPath expression, from the document root to the target resource. In YANG-API, URL friendly path expressions are used instead.

The YANG "instance-identifier" (i-i) data type is represented in YANG-API with the path expression format defined in this section.

Name	Comments
point	Insertion point is always a full i-i
path	Request URI path is a full or partial i-i

YANG-API instance-identifier Type Conversion

The "path" component of the request URI contains the absolute path expression that identifies the target resource. The "select" query parameter is used to optionally identify the requested data nodes

within the target resource to be retrieved in a GET operation.

A predictable location for a data resource is important, since applications will code to the YANG data model module, which uses static naming and defines an absolute path location for all data nodes.

A YANG-API data resource identifier is not an XPath expression. It is encoded from left to right, starting with the top-level data node, according to the "api-path" rule in Section 5.3.1.1. The node name of each ancestor of the target resource node is encoded in order, ending with the node name for the target resource.

If the "select" is present, it is encoded, starting with a child node of the target resource, according to the "api-select" rule defined in Section 5.3.1.1.

If a data node in the path expression is a YANG list node, then the key values for the list (if any) are encoded according to the "key-value" rule. If the list node is the target resource, then the key values MAY be omitted, according to the operation. For example, the POST operation to create a new data resource for a list node does not allow the key values to be present in the request URI.

The key leaf values for a data resource representing a YANG list MUST be encoded as follows:

- The value of each leaf identified in the "key" statement is encoded in order.
- All the components in the "key" statement MUST be encoded. Partial instance identifiers are not supported.
- Each value is encoded using the "key-value" rule in Section 5.3.1.1, according to the encoding rules for the data type of the key leaf.
- An empty string can be a valid key value (e.g., "/top/list/key1//key3").
- The "/" character MUST be URL-encoded (i.e., "%2F").
- All whitespace MUST be URL-encoded.
- A "null" value is not allowed since the "empty" data type is not allowed for key leafs.

- The XML encoding is defined in [RFC6020].

- The JSON encoding is defined in [I-D.lhotka-yang-json].

- The entire "key-value" MUST be properly URL-encoded, according to the rules defined in [RFC3986].

Examples:

```
/yang-api/datastore/jukebox/library/artist/17&select=name

/yang-api/datastore/newlist/17&select=nextlist/22/44/myleaf

/yang-api/datastore/somelist/fred%20and%20wilma

/yang-api/datastore/somelist/fred%20and%20wilma/address
```

5.3.1.1. ABNF For Data Resource Identifiers

The following ABNF syntax is used to construct YANG-API path identifiers:

```
api-path = "/" api-identifier
               0*("/" (api-identifier | key-value ))

[FIXME: the syntax for the select string is still TBD]
api-select = api-identifier
               0*("/" (api-identifier | key-value ))

api-identifier = [module-name ":"] identifier

module-name = identifier

key-value = string

;; An identifier MUST NOT start with
;; (('X'|'x') ('M'|'m') ('L'|'l'))
identifier  = (ALPHA / "_")
              *(ALPHA / DIGIT / "_" / "-" / ".")

string = <an unquoted string>
```

5.3.2. Identifying YANG-defined Data Resources

The data resources used in YANG-API are defined with YANG data definition statements.

Not every data node defined in a YANG module should be treated as a

resource. The YANG-API needs to know which YANG data nodes are resources, and which are fields within a resource.

For data resources, YANG-API uses a simple algorithm for defining resource boundaries, within the conceptual sub-trees described by YANG data definition statements.

All top-level data nodes are considered to be resources. For nodes within a top-level resource:

- a presence container starts a new resource
- a list starts a new resource
- an optional terminal node (anyxml, leaf, or leaf-list) starts a new resource
- a data node of type "anyxml" cannot have any sub-resources

A non-configuration data node cannot be a separate resource from its parent. Only top-level data nodes are considered to be resources (which only support retrieval methods).

5.3.3. Identifying Optional Keys

It is sometimes useful to have the server assign the key(s) for a new resource. The "Location" header will indicate the key value(s) that the server selected, so the client does not need to provide all the key leaf values.

It is useful to identify in the YANG data model module which key leafs are optional to provide, and which are not. The YANG extension statement "optional-key" is provided to indicate that the leaf definition represents an optional key.

The client MAY provide a value for a key leaf in a POST operation. Refer to Section 8 for details on the "optional-key" extension. Refer to Section 12 for usage examples of this YANG extension statement.

5.3.4. Data Resource Retrieval

There are four types of filtering for retrieval of data resources. This section defines each mode.

5.3.4.1. Conditional Retrieval

The HTTP headers (such as "If-Modified-Since" and "If-Match") can by used in for a request message for a GET operation to check a condition within the server state, such as the last time the datastore resource was modified, or the resource entity tag of the target resource.

If the condition is met according to the header definition, a "200 OK" Status-Line and the data requested is returned in the response message. If the condition is not met, a "304 Not Modified" Status-Line is returned in response message instead.

5.3.4.2. Data Classification Retrieval

The "config" query parameter can be used with the GET operation to specify whether configuration or non-configuration data is requested. Refer to Section 3.8.1 for more details on the "config" query parameter.

5.3.4.3. Subset Retrieval

The "Range" header is used to request a specific subset of the instances of a list or leaf-list data resource that are returned by the server for a retrieval operation. Normally, if the target resource in a request message does not specify an instance, then all instances are returned.

The YANG-API protocol uses the token "entries" instead of "bytes" as the range units.

The entries are numbered starting from "0". A list or leaf-list can change order between requests so the client needs to be aware of the data model semantics, and whether the list contents are stable enough to use the subset retrieval mechanism.

If the requested range cannot be returned because the range specification includes index values for entries that do not exist, then an error occurs, and the server MUST return a "416 Requested range not satisfiable" Status-Line.

If the range request can be satisfied, then a "200 OK" Status-Line is returned, and the response MUST include a "Content-Range" header indicating which entries are returned. The response message body contains the data for the requested range of entries.

Example:

In this example, the client is requesting 5 "artist" resource entries, starting with the 10th entry:

Request from client:

```
GET /yang-api/datastore/jukebox/library/artist HTTP/1.1
Host: example.com
Accept: application/vnd.yang.data+json
Range: entries 10-14
```

Response from server:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 13:01:20 GMT
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/vnd.yang.data+json
Content-Range: entries 10-14
Server: example-server
Last-Modified: Mon, 23 Apr 2012 02:12:20 GMT
ETag: abcada438af

{
  "artist" : {
      // content removed for brevity
  }
}
```

5.3.4.4. Filtered Retrieval

The "select" query parameter is used to specify a filter that should be applied to the target resource to request a subset of all possible descendant nodes within the target resource.

The format of the "select" parameter string is defined in Section 3.8.6. The set of nodes selected by the filter expression is applied to each context node identified by the target resource.

5.4. Operation Resource

An operation resource represents an RPC operation defined with the YANG "rpc" statement.

All operation resources share the same module namespace as any top-level data resources, so the name of an operation resource cannot conflict with the name of a top-level data resource defined within the same module.

If 2 different YANG modules define the same "rpc" identifier, then the module name MUST be used in the request URI. For example, if "module-A" and "module-B" both defined a "reset" operation, then invoking the operation from "module-A" would be requested as follows:

```
POST /yang-api/operations/module-A:reset HTTP/1.1
Server example.com
```

Any usage of an operation resource from the same module, with the same name, refers to the same "rpc" statement definition. This behavior can be used to design RPC operations that perform the same general function on different resource types.

If the "rpc" statement has an "input" section, then a message body MAY be sent by the client in the request, otherwise the request message MUST NOT include a message body. If the "rpc" statement has an "output" section, then a message body MAY be sent by the server in the response. Otherwise the server MUST NOT include a message body in the response message, and MUST send a "204 No Content" Status-Line instead.

5.4.1. Encoding Operation Input Parameters

If the "rpc" statement has an "input" section, then the "input" node is provided in the message body, corresponding to the YANG data definition statements within the "input" section.

Example:

The following YANG definition is used for the examples in this section.

```
rpc reboot {
  input {
    leaf delay {
      units seconds;
      type uint32;
      default 0;
    }
    leaf message { type string; }
    leaf language { type string; }
  }
}
```

The client might send the following POST request message:

```
POST /yang-api/datastore/operations/reboot HTTP/1.1
Host: example.com
Content-Type: application/vnd.yang.data+json

{
  "input" : {
    "delay" : 600,
    "message" : "Going down for system maintenance",
    "language" : "en-US"
  }
}
```

The server might respond:

```
HTTP/1.1 204 No Content
Date: Mon, 25 Apr 2012 11:01:00 GMT
Server: example-server
```

5.4.2. Encoding Operation Output Parameters

If the "rpc" statement has an "output" section, then the "output" node is provided in the message body, corresponding to the YANG data definition statements within the "output" section.

Example:

The following YANG definition is used for the examples in this section.

```
 rpc get-reboot-info {
   input {
     leaf reboot-time {
       units seconds;
       type uint32;
     }
     leaf message { type string; }
     leaf language { type string; }
   }
 }
```

The client might send the following POST request message:

```
POST /yang-api/datastore/operations/get-reboot-info HTTP/1.1
Host: example.com
```

The server might respond:

```
HTTP/1.1 200 OK
Date: Mon, 25 Apr 2012 11:10:30 GMT
Server: example-server
Content-Type: application/vnd.yang.data+json

{
  "output" : {
    "reboot-time" : 30,
    "message" : "Going down for system maintenance",
    "language" : "en-US"
  }
}
```

5.4.3. Identifying YANG-defined Operation Resources

The operation resources used in YANG-API are defined with YANG "rpc" statements. All "rpc" statements within a YANG module that are supported by the server are available as operation resources.

5.5. Transaction Resource

The "transaction" resource type is used to construct a set of one or more edit operations on data resources within a "scratchpad" datastore resource. The transaction can be committed when the client decides the data resource edits are complete. The transaction can also be reverted and updated, as described later in this section.

This resource type will only be supported if the "edit-model" capabilities field in the API resource includes the value "transaction". If transactions are supported, then the server will allow the client to create, use, and delete transaction resources.

The POST operation is used to create a new transaction resource. The DELETE operation is used to cleanup and delete an existing transaction resource. The PUT and PATCH operations are not supported for this resource type.

The media type for the transaction resource type is either "application/vnd.yang.transaction+xml" or "application/vnd.yang.transaction+json".

The procedures for editing the transaction datastore contents are the same as those for editing the running configuration datastore except the changes do not take effect right away and the datastore integrity validation tests are not done until the transaction is committed to running configuration datastore.

The following steps are typically followed to use transaction

resources:

- create a transaction resource using the URI "/yang-api/transaction".
- the server will allocate a new transaction and return its resource ID.
- add/alter/delete data resources within the scratchpad datastore
- commit the transaction to the running configuration datastore.
- delete the transaction resource

5.5.1. Creating a Transaction Resource

In order to reduce the complexity of query parameters and allow easier extensibility of transaction resource creation, the configuration parameters for the transaction are sent in the request message for the POST operation.

The only parameter at this time is the "exclusive-mode" parameter, which is used by the client to request that no other transactions or direct edits are allowed to alter the running configuration datastore while the exclusive mode transaction resource exists. An exclusive mode transaction if the server transaction-model is "shared" is conceptually equivalent in NETCONF to global locks on both the "candidate" and "running" datastores.

The following YANG leaf definition is used for the "exclusive-mode" parameter, for encoding purposes:

```
leaf exclusive-mode {
  type boolean;
  default false;
  description "Exclusive transaction mode";
}
```

When a transaction resource is created by the client, the server will generate an opaque string to identify the transaction. This transaction ID will be used by the server in the resource ID for the new transaction.

If the server uses a shared transaction model, then the transaction ID MAY be the same for multiple transaction resources. Otherwise the server SHOULD use a unique identifier for each transaction resource.

The server does not ensure exclusive access to a particular

transaction. The access control mechanisms for sharing transactions is out of scope for this document.

After a transaction has been successfully created, it can be accessed via the "Location" header returned in the response message.

Example:

The following message shows an exclusive transaction resource request. The client might send:

```
POST /yang-api/transaction HTTP/1.1
Host: example.com
Content-Type: application/vnd.yang.transaction+json

{
  "transaction" : {
    "exclusive-mode" : true
  }
}
```

The server might reply:

```
HTTP/1.1 201 Created
Date: Mon, 23 Apr 2012 17:01:00 GMT
Server: example-server
Location: http://example.com/yang-api/transaction/12345
Last-Modified: Mon, 23 Apr 2012 19:48:00 GMT
ETag: b38830de24c
```

5.5.2. Editing a Transaction Datastore

When a transaction resource is created, the server will create a child datastore resource, which is a conceptual scratchpad for collecting edits to later be applied all at once to the running configuration datastore. The initial contents of this datastore are the contents of the running configuration datastore at the time the transaction is created.

After a transaction has been successfully created, it can be accessed by using the previously retrieved "Location" header value in the request URI of new request messages. This datastore resource is a child node of the resource ID node, identified by a URI.

For example, the "path" component of a request URI for a datastore resource (for transaction ID "12345") would be:

```
"/yang-api/transaction/12345/datastore"
```

The client can add, edit, or delete the data resources within the transaction datastore. Refer to Section 5.3 for details on editing data resources.

Example:

The following message shows the creation of a new "artist" resource within the "jukebox" resource. The request URI is split across lines for display purposes only.

The client might send:

```
POST /yang-api/transaction/12345/datastore/jukebox/
   library/artist HTTP/1.1
Host: example.com
Content-Type: application/vnd.yang.data+json

{
  "artist" : {
    "name" : "Miles Davis"
  }
}
```

The server might reply as follows. The "Location" header is split across lines for display purposes only.

```
HTTP/1.1 201 Created
Date: Mon, 24 Apr 2012 11:01:00 GMT
Server: example-server
Location: http://example.com/yang-api/transaction/
  12345/datastore/jukebox/library/artist/2
Last-Modified: Mon, 24 Apr 2012 11:01:00 GMT
ETag: b38830de24c
```

5.5.3. Deleting a Transaction Resource

Once a client is finished with a transaction resource, it SHOULD be deleted by the client. A transaction resource is not deleted when a commit is completed. The DELETE operation is used to terminate the transaction, and discard the transaction database and all its data resource contents.

Example:

The following message shows the deletion of an existing transaction resource.

The client might send:

```
DELETE /yang-api/transaction/12345
Host: example.com
```

The server might reply as follows.

```
HTTP/1.1 204 No Content
Date: Mon, 24 Apr 2012 12:01:00 GMT
Server: example-server
```

5.5.4. Transaction Operations

There are a small number of operation resources available for transaction resources. These are protocol operations beyond the basic CRUD operations allowed for the data resources within the transaction datastore.

5.5.4.1. commit

The "commit" operation is used to apply the contents of the transaction datastore to the running configuration datastore.

If this operation succeeds then a "204 No Content" Status-Line is sent in the response message. If the operation fails, the appropriate error code is set according to the rules in Section 6, and the error report is sent in the response, according to the format defined in Section 6.1.

Example:

The following message exchange shows a commit operation. The client might send:

```
POST /yang-api/transaction/12345/commit
Host: example.com
```

The server might reply as follows:

```
HTTP/1.1 204 No Content
Date: Mon, 25 Apr 2012 01:21:00 GMT
Server: example-server
Last-Modified: Mon, 25 Apr 2012 01:21:00 GMT
ETag: ab34530de24c
```

5.5.4.2. discard-changes

The "discard-changes" operation is used to replace the contents of the transaction datastore with the contents of the running configuration datastore.

If this operation succeeds then a "204 No Content" Status-Line is sent in the response message. If the operation fails, the appropriate error code is set according to the rules in Section 6, and the error report is sent in the response, according to the format defined in Section 6.1.

Example:

The following message exchange shows a discard-changes operation. The client might send:

```
POST /yang-api/transaction/12345/discard-changes
Host: example.com
```

The server might reply as follows.

```
HTTP/1.1 204 No Content
Date: Mon, 25 Apr 2012 01:22:00 GMT
Server: example-server
Last-Modified: Mon, 25 Apr 2012 01:22:00 GMT
ETag: ee3498de24c
```

5.5.4.3. update

The "update" operation is used to merge the contents of the running configuration datastore into the transaction datastore. If any editing conflicts are detected that cannot be resolved by the server, then the update operation MUST fail, and the transaction datastore contents MUST remain unchanged after the operation is completed.

If this operation succeeds then a "204 No Content" Status-Line is sent in the response message. If the operation fails, the appropriate error code is set according to the rules in Section 6, and the error report is sent in the response, according to the format defined in Section 6.1.

Example:

The following message exchange shows an update operation. The client might send:

```
POST /yang-api/transaction/12345/update
Host: example.com
```

The server might reply as follows.

```
HTTP/1.1 204 No Content
Date: Mon, 25 Apr 2012 01:32:00 GMT
Server: example-server
Last-Modified: Mon, 25 Apr 2012 01:32:00 GMT
ETag: ab23984de125
```

5.5.4.4. validate

The "validate" operation is used to validate the contents of the transaction datastore. The server will verify that the transaction datastore can be committed to the running configuration datastore. If any editing conflicts are detected which cannot be resolved by the server, then the update operation MUST fail.

If this operation succeeds then a "204 No Content" Status-Line is sent in the response message. If the operation fails, the appropriate error code is set according to the rules in Section 6, and the error report is sent in the response, according to the format defined in Section 6.1.

Example:

The following message exchange shows a validate operation. The client might send:

```
POST /yang-api/transaction/12345/validate
Host: example.com
```

The server might reply as follows.

```
HTTP/1.1 204 No Content
Date: Mon, 25 Apr 2012 01:42:00 GMT
Server: example-server
Last-Modified: Mon, 25 Apr 2012 01:32:00 GMT
ETag: ab23984de125
```

6. Error Reporting

HTTP Status-Lines are used to report success or failure for YANG-API operations. The <rpc-error> element returned in NETCONF error responses contains some useful information. This error information is adapted for use in YANG-API, and error information is returned for "4xx" class of status codes.

The following table summarizes the return status codes used specifically by YANG-API operations:

Status-Line	Description
100 Continue	POST accepted, 201 should follow
200 OK	Success with response body
201 Created	POST to create a resource success
202 Accepted	POST to create a resource accepted
204 No Content	Success without response body
304 Not Modified	Conditional operation not done
400 Bad Request	Invalid request message
403 Forbidden	Access to resource denied
404 Not Found	Resource target or resource node not found
405 Method Not Allowed	Method not allowed for target resource
409 Conflict	Resource or lock in use
413 Request Entity Too Large	too-big error
414 Request-URI Too Large	too-big error
415 Unsupported Media Type	non YANG-API media type
416 Requested range not satisfiable	If-Range error
500 Internal Server Error	operation-failed
501 Not Implemented	unknown-operation
503 Service Unavailable	Recoverable server error

HTTP Status Codes used in YANG-API

Since an operation resource is defined with a YANG "rpc" statement, a mapping between the NETCONF <error-tag> value and the HTTP status code is needed. The specific error condition and response code to use are data-model specific and might be contained in the YANG "description" statement for the "rpc" statement.

<error-tag>	status code
in-use	409
invalid-value	400
too-big	413
missing-attribute	400
bad-attribute	400
unknown-attribute	400
bad-element	400
unknown-element	400
unknown-namespace	400
access-denied	403
lock-denied	409
resource-denied	409
rollback-failed	500
data-exists	409
data-missing	409
operation-not-supported	501
operation-failed	500
partial-operation	500
malformed-message	400

Mapping from error-tag to status code

6.1. Error Response Message

When an error occurs for a request message on a data resource or an operation resource, and a "4xx" class of status codes (except for status code "403"), then the server SHOULD send a response body containing the information described by the following YANG data definition statement:

```
      container errors {
        config false;

        list error {
          reference "RFC 6241, Section 4.3";
          leaf error-type {
            mandatory true;
            type enumeration {
              enum transport;
              enum rpc;
              enum protocol;
              enum application;
            }
          }
          leaf error-tag {
            mandatory true;
            type string;
          }
          leaf error-app-tag {
            type string;
          }
          leaf error-path {
            type string;  // YANG-API encoded instance-identifier
          }
          leaf error-message {
            type string;
          }
          container error-info {
             // anyxml content here
          }
        }
      }
```

Example:

The following example shows an error returned for an "lock-denied" error on a datastore resource.

```
   POST /yang-api/operations/lock-datastore HTTP/1.1
   Host: example.com
```

The server might respond:

```
HTTP/1.1 409 Conflict
Date: Mon, 23 Apr 2012 17:11:00 GMT
Server: example-server
Content-Type: application/vnd.yang.api+json

{
  "errors": {
    "error": {
      "error-type": "protocol",
      "error-tag": "lock-denied",
      "error-message": "Lock failed, lock is already held",
    }
  }
}
```

7. RelaxNG Grammar

TBD

8. YANG-API module

RFC Ed.: update the date below with the date of RFC publication and remove this note.

<CODE BEGINS> file "ietf-yang-api@2012-05-27.yang"

```
module ietf-yang-api {
  namespace "urn:ietf:params:xml:ns:yang:ietf-yang-api";
  prefix "api";

  organization
    "IETF NETCONF (Network Configuration) Working Group";

  contact
    "Editor:   Andy Bierman
               <mailto:andy@yumaworks.com>

     Editor:   Martin Bjorklund
               <mailto:mbj@tail-f.com>";

  description
    "This module contains a collection of YANG language extensions
     to describe REST API Resources using YANG data definition
     statements.

     Copyright (c) 2012 IETF Trust and the persons identified as
     authors of the code.  All rights reserved.

     Redistribution and use in source and binary forms, with or
     without modification, is permitted pursuant to, and subject
     to the license terms contained in, the Simplified BSD License
     set forth in Section 4.c of the IETF Trust's Legal Provisions
     Relating to IETF Documents
     (http://trustee.ietf.org/license-info).

     This version of this YANG module is part of RFC XXXX; see
     the RFC itself for full legal notices.";

  // RFC Ed.: replace XXXX with actual RFC number and remove this
  // note.

  // RFC Ed.: remove this note
  // Note: extracted from draft-bierman-netconf-yang-api-00.txt

  // RFC Ed.: update the date below with the date of RFC publication
  // and remove this note.
  revision 2012-05-27 {
```

```
    description
      "Initial revision.";
    reference
      "RFC XXXX: YANG-API Protocol.";
  }

  /*
   * Extensions
   */

  extension optional-key {
    description
      "This extension is used to allow the client to create
       a new instance of a resource without providing a
       value for the key leaf containing this statement.
       This extension is ignored for NETCONF, and only
       applies to YANG-API resources and fields.
       This extension is ignored unless it appears
       directly within a 'leaf' data definition statement.";
  }


  /*
   * Operations
   */

  rpc lock-datastore {
    description
      "Lock the running configuration datastore for writing.";
  }

  rpc save-datastore {
    description
      "Save the running configuration datastore to non-volatile
       storage.";
  }

  rpc unlock-datastore {
    description
      "Unlock the running configuration datastore.";
  }

  rpc commit {
    description
      "Commit the transaction datastore contents to
       the running configuration datastore.";
  }
```

```
    rpc discard-changes {
      description
        "Replace the transaction datastore contents with
         the running configuration datastore contents.";
    }

    rpc update {
      description
        "Attempt to merge the running configuration datastore
         contents into the transaction datastore contents.";
    }

    rpc validate {
      description
        "Validate the transaction datastore contents.";
    }

  }


  <CODE ENDS>
```

9. IANA Considerations

TBD

10. Security Considerations

TBD

11. Open Issues

- Resource creation order and other dependencies between resources are not well identified in YANG. YANG has leafrefs and instance-identifiers, which can be used to identify some order dependencies. Are any new mechanisms needed in YANG-API needed to identify resource creation order and other dependency requirements?

- There is no "message-id" field in a YANG-API message. Is a message identifier needed? If so, should either the "Message-ID" or "Content-ID" header from RFC 2392 be used for this purpose?

- The non-configuration data resources are combined with the configuration data resources within the YANG-API datastore. The "config" query parameter is used to pick 1 or the other for GET operations. Is this the best way to deal with YANG config-stmt? Should YANG-API follow the same data classifications as YANG (i.e. config=true|false), or create something new? Note that transactions are config=true only, like the candidate datastore in NETCONF.

- Should confirmed commit be added? If so, how? Should NETCONF "confirmed-commit" procedure be used exactly for the transaction commit operation, or should a new procedure be defined?

- Should datastore operations be added for "backup" and "restore" functionality?

- Should sessions be used or not? Should "reusable sessions" be used? Better for auditing? How does locking of the /yang-api/ datastore resource work for multiple edits if a session is 1 operation? When does the server release the lock and decide it has been abandoned or client was disconnected?

- What syntax should be used for the "select" query parameter?

- Should the "/yang-api/modules" field within the API resource be a separate resource, with its own timestamp? Currently the API timestamp is coupled to any changes to the list of loaded modules. Should the API resource be static and cacheable?

- How should resource discovery be done?

- What to do about no REMOVE operation, just DELETE? The effect is local to the request; in a NETCONF edit-config it is worse, since the netconf request might create/delete/modify many nodes

- Should every YANG data node be a data resource and every YANG RPC statement an operation resource? Is a YANG extension needed to allow data modeler control of resource boundaries?

- Encoding of leafrefs? Is there some additional meta-data needed? Do leafref nodes need to be identified in responses (RFC 5988) or is the YANG module definition sufficient to provide this meta-data?

- What should the default algorithm be for defining data resources? Should the default for an augment from another namespace be to start a new resource? Top-level data node defaults as a resource OK?

- Is the token "entries" legal in the YANG-API usage of Range? What units should be used? "bytes" is the only token defined by HTTP.

- How should private transaction conflicts be handled? Currently up to the server to decide how to handle conflicts. What happens if there are transactions A and B. A commits. Next, B commits w/o updating. Will A's changes be lost? Maybe. Detecting conflicts may require a very resource-intensive implementation on the server - may force the server to create a copy of the entire datastore for each transaction. Want to allow a transaction to be just a diff-set towards the datastore, so transactions are cheap.

- Does the shared transaction work like the candidate wrt to locks? I.e. will an exclusive transaction start fail if there are uncommitted changes?

- Need to specify the update/commit procedure in more detail so that there is some server flexibility and client can tell what the server will do? E.g., what causes a conflict? When is update required before commit?

- Are all header lines used by YANG-API supported by common application frameworks, such as FastCGI and WSGI? If not, then should query parameters be used instead, since the QUERY_STRING is widely available to WEB applications?

- Should the <errors> element returned in error responses be a separate media type?

- Locks tied to sessions, but if don't have sessions, then how do locks work?

- Should locks be modeled as resources as operations. I.e., remove lock-datastore and unlock-datastore operations. and transactions

will be required (exclusive mode) to write more than one operation at a time with exclusive access.

- Should the writable-running (direct mode) be removed and just have transaction resources, which will hide writes to running config?

- Should POST to create a new transaction for a shared candidate be needed? Could get the same transaction ID back each ime? Predictable resource needed instead?

- Do changes to the shared transaction show up in all copies when the change is made?

- How can private transactions be shared securely? Are any new access control mechanisms needed?

12. Example YANG Module

```
   module example-jukebox {

      namespace "http://example.com/ns/example-jukebox";
      prefix "jbox";

      import ietf-yang-api { prefix api; }

      organization "Example, Inc.";
      description "Example Jukebox Data Model Module";
      revision "2012-05-30";


      identity genre {
        description "Base for all genre types";
      }

      // abbreviated list of genre classifications
      identity Alternative {
        base genre;
      }
      identity Blues {
        base genre;
      }
      identity Country {
        base genre;
      }
      identity Jazz {
        base genre;
      }
      identity Pop {
        base genre;
      }
      identity Rock {
        base genre;
      }

      container jukebox {
        presence
          "An empty container indicates that the jukebox
           service is available";

        container library {
          list artist {
            key index;
            unique name;
```

```
            leaf index {
              api:optional-key;
              type uint32;
              description
                "Optional key used instead of natural key for
                 example. Also rare but possible artists with
                 the same name are really different entities.";
            }
            leaf name {
              type string;
            }

            list album {
              key name;
              leaf name {
                type string {
                  length "1 .. max";
                }
              }
              leaf genre {
                type identityref { base genre; }
              }
              leaf year {
                type uint16 {
                  range "1900 .. max";
                }
              }
              list song {
                api:optional-key;
                key index;
                ordered-by user;
                leaf index {
                  type uint32;
                }
                leaf name {
                  mandatory true;
                  type string;
                }
                leaf location {
                  mandatory true;
                  type string;
                }
                leaf format {
                  type string;
                }
                leaf length {
                  units "seconds";
                  type uint32;
```

```
            }
          }
        }
      }
      leaf artist-count {
         config false;
         type uint32;
         units "songs";
         description "Number of artists in the library";
      }
      leaf album-count {
         config false;
         type uint32;
         units "albums";
         description "Number of albums in the library";
      }
      leaf song-count {
         type uint32;
         units "songs";
         description "Number of songs in the library";
      }
    }

    list playlist {
      description
        "Example configuration data resource";
      key name;
      leaf name {
        type string;
      }
      leaf description {
        type string;
      }
      list song {
        description
          "Example nested configuration data resource";
        ordered-by user;
        key index;
        leaf index {
          api:optional-key;
          type uint32;
        }
        leaf id {
          mandatory true;
          type instance-identifier;
          description
            "Song identifier. Must identify an instance of
             /jukebox/library/artist/album/song.
```

```
                The id is not the key to allow duplicates
                in a playlist";
            }
          }
        }

        container player {
          leaf gap {
            description "Time gap between each song";
            units "tenths of seconds";
            type decimal64 {
              fraction-digits 1;
              range "0.0 .. 2.0";
            }
          }
        }
      }

      rpc play {
        description "Control function for the jukebox player";
        input {
          leaf playlist {
            type string;
            mandatory true;
            description "playlist name";
          }
          leaf song-number {
            type uint32;
            mandatory true;
            description "Song number in playlist to play";
          }
        }
      }
    }
```

13. Normative References

[I-D.lhotka-yang-json]
Lhotka, L., "Modeling JSON Text with YANG", draft-lhotka-yang-json-00 (work in progress), April 2012.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, March 2010.

[RFC6020] Bjorklund, M., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, October 2010.

[RFC6021] Schoenwaelder, J., "Common YANG Data Types", RFC 6021, October 2010.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, June 2011.

[RFC6536] Bierman, A. and M. Bjorklund, "Network Configuration Protocol (NETCONF) Access Control Model", RFC 6536, March 2012.

Authors' Addresses

Andy Bierman
YumaWorks

Email: andy@yumaworks.com

Martin Bjorklund
Tail-f Systems

Email: mbj@tail-f.com