core
Internet-Draft
Intended status: Standards Track
Expires: April 18, 2016

P. van der Stok
consultant
A. Bierman
YumaWorks
J. Schoenwaelder
Jacobs University
A. Sehgal
consultant
October 16, 2015

CoAP Management Interface
draft-vanderstok-core-comi-08

Abstract

This document describes a network management interface for constrained devices, called CoMI. CoMI is an adaptation of the RESTCONF protocol for use in constrained devices and networks. It is designed to reduce the message sizes, server code size, and application development complexity. The Constrained Application Protocol (CoAP) is used to access management data resources specified in YANG, or SMIv2 converted to YANG. The payload of the CoMI message is encoded in Concise Binary Object Representation (CBOR).

Note

Discussion and suggestions for improvement are requested, and should be sent to core@ietf.org.

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 April 18, 2016.

Table of Contents

1. Introduction

The Constrained Application Protocol (CoAP) [RFC7252] is designed for Machine to Machine (M2M) applications such as smart energy and building control. Constrained devices need to be managed in an automatic fashion to handle the large quantities of devices that are expected in future installations. The messages between devices need to be as small and infrequent as possible. The implementation complexity and runtime resources need to be as small as possible.

The draft [I-D.ietf-netconf-restconf] describes a REST-like interface called RESTCONF, which uses HTTP methods to access structured data defined in YANG [RFC6020]. The use of standardized data sets, specified in a standardized language such as YANG, promotes interoperability between devices and applications from different manufacturers.

A large amount of Management Information Base (MIB) [RFC3418] [mibreg] specifications already exists for monitoring purposes. This data can be accessed in RESTCONF or CoMI if the server converts the SMIv2 modules to YANG, using the mapping rules defined in [RFC6643].

RESTCONF allows access to data resources contained in NETCONF [RFC6241] data-stores. RESTCONF messages can be encoded in XML [XML] or JSON [RFC7159]. The GET method is used to retrieve data resources and the POST, PUT, PATCH, and DELETE methods are used to create, replace, merge, and delete data resources.

The CoRE Management Interface (CoMI) is intended to work in a stateless client-server fashion. CoMI (and also RESTCONF) uses a single round-trip to complete a single editing transaction, where NETCONF needs up to 10 round trips.

RESTCONF relies on HTTP with TCP in contrast to CoMI which uses CoAP that is optimized for UDP with less overhead for small messages. RESTCONF uses the HTTP methods HEAD, and OPTIONS, which are not used by CoMI.

CoMI supports the methods GET, PUT, PATCH, POST and DELETE. The payload of CoMI is encoded in CBOR [RFC7049] which can be automatically generated from JSON [RFC7159]. CBOR has a binary format and hence has more coding efficiency than JSON. To promote small packets, CoMI uses an additional "data-identifier string-to-number conversion" to minime CBOR payloads and URI length. It is assumed that the managed device is the most constrained entity. The client might be more capable, however this is not necessarily the case.

Currently, small managed devices need to support at least two protocols: CoAP and SNMP [RFC3411]. When the MIB can be accessed with the CoAP protocol, the SNMP protocol can be replaced with the CoAP protocol. Although the SNMP server size is not huge (see Appendix A), the code for the security aspects of SMIv3 [RFC3414] is not negligible. Using CoAP to access secured management objects reduces the code complexity of the stack in the constrained device, and harmonizes applications development.

The objective of CoMI is to provide a Function Set that reads and sets values of managed objects in devices to (1) initialize parameter values at start-up, (2) acquire statistics during operation, and (3) maintain nodes by adjusting parameter values during operation.

The goal of CoMI is to provide information exchange in a uniform manner as a first step to the full management functionality as specified in [I-D.ersue-constrained-mgmt].

1.1. Design considerations

CoMI supports discovery of resources, accompanied by reading, writing and notification of resource values. As such it is close to the device management of the Open Mobile Alliance described in [OMA]. A comparison between CoMI and LWM2M management can be found in Appendix D. CoMI supports MIB modules which have been translated once from SMIv2 to YANG, using [RFC6643]. This mapping is read-only so writable SMIv2 objects need to be converted to YANG using an implementation-specific mapping.

The YANG data model contains a lot of information that can be exploited by automation tools and need not be transported in the request messages, ultimately leading to reduced message sizes.

1.2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Readers of this specification should be familiar with all the terms and concepts discussed in [RFC3410], [RFC3416], and [RFC2578].

The following terms are defined in the NETCONF protocol [RFC6241]: client, configuration data, data-store, and server.

The following terms are defined in the YANG data modelling language [RFC6020]: container, data node, key, key leaf, leaf, leaf-list, and list.

The following terms are defined in RESTCONF protocol [I-D.ietf-netconf-restconf]: data resource, data-store resource, edit operation, query parameter, target resource, and unified data-store.

The following terms are defined in this document:

YANG hash: CoMI object identifier, which is a 30-bit numeric hash of the YANG object identifier string for the object. When a YANG hash value is printed in the payload, error-path or other string, then the lowercase hexadecimal representation is used. Leading zeros are used so the value uses 8 hex characters.

Rehash bit: Bit 31. If a particular YANG hash value is a re-hash for an identifier, then the rehash bit will be set in the object identifier. This allows the server to return descendant nodes that have been rehashed, instead of returning an error for an entire GET request.

Data-node instance: An instance of a data-node specified in a YANG module present in the server. The instance is stored in the memory of the server.

Notification-node instance: An instance of a schema node of type notification, specified in a YANG module present in the server. The instance is generated in the server at the occurrence of the corresponding event and appended to a stream.

The following list contains the abbreviations used in this document.

XXXX: TODO, and others to follow.

1.2.1. Tree Diagrams

A simplified graphical representation of the data model is used in the YANG modules specified in this document. The meaning of the symbols in these diagrams is as follows:

Brackets "[" and "]" enclose list keys.

Abbreviations before data node names: "rw" means configuration data (read-write) and "ro" state data (read-only).

Symbols after data node names: "?" means an optional node, "!" means a presence container, and "*" denotes a list and leaf-list.

Parentheses enclose choice and case nodes, and case nodes are also marked with a colon (":").

Ellipsis ("...") stands for contents of subtrees that are not shown.

2. CoMI Architecture

This section describes the CoMI architecture to use CoAP for the reading and modifying of instrumentation variables used for the management of the instrumented node.

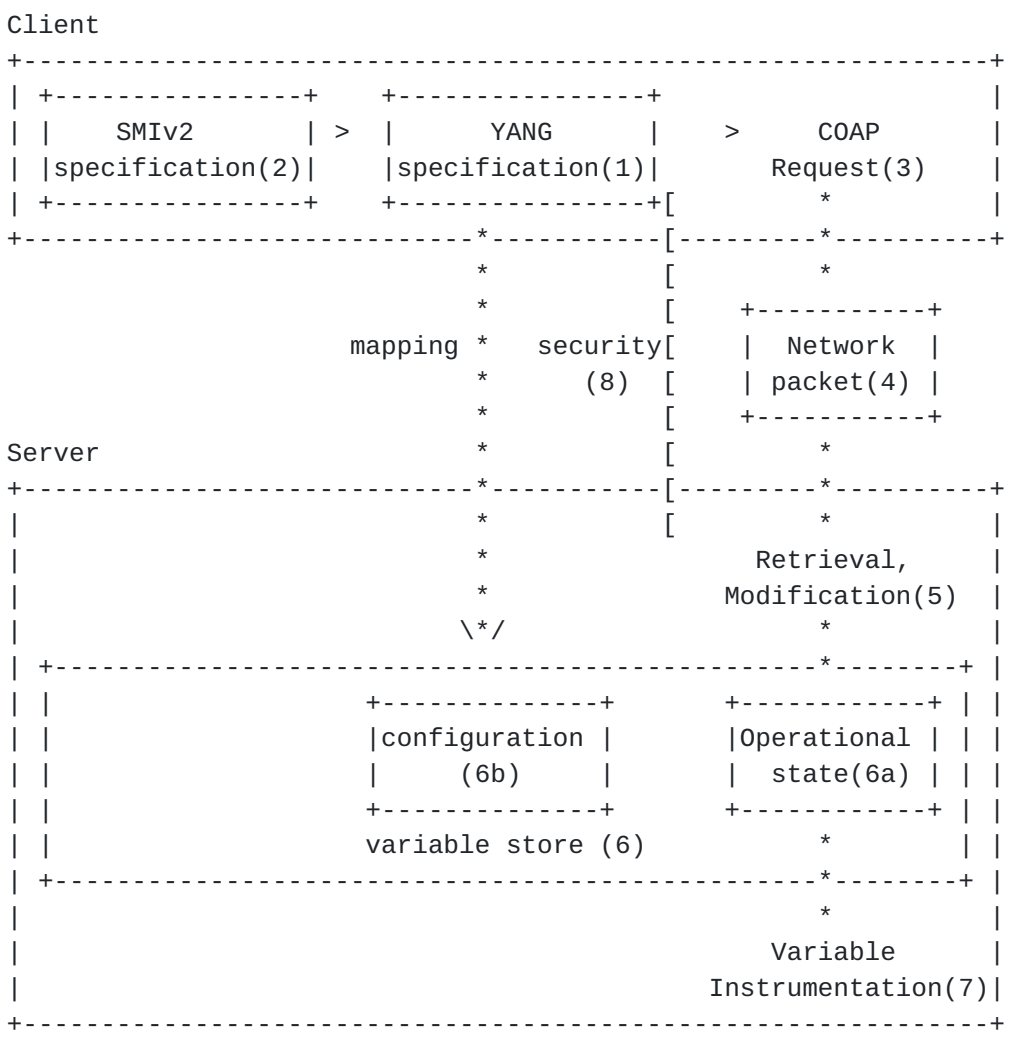

Figure 1: Abstract CoMI architecture

Figure 1 is a high level representation of the main elements of the CoAP management architecture. A client sends requests as payload in packets over the network to a managed constrained node.

Objectives are:

- Equip a constrained node with a management server that provides information about the operational characteristics of the code running in the constrained node.

- The server provides this information in a variable store that contains values describing the performance characteristics and the code parameter values.

- The client receives the performance characteristics on a regular basis or on request.
- The client sets the parameter values in the server at bootstrap and intermittently when operational conditions change.
- The constrained network requires the payload to be as small as possible, and the constrained server memory requirements should be as small as possible.

For interoperability it is required that in addition to using the Internet Protocol for data transport:

- The names, type, and semantics of the instrumentation variables are standardized.
- The instrumentation variables are described in a standard language.
- The URI of the CoAP request is standardized.
- The format of the packet payload is standardized.
- The notification from server to client is standardized.

The different numbered components of Figure 1 are discussed according to component number.

(1) YANG specification: contains a set of named and versioned modules. A module specifies a hierarchy of named and typed resources. A resource is uniquely identified by a sequence of its name and the names of the enveloping resources following the hierarchy order. The YANG specification serves as input to the writers of application and instrumentation code and the humans analying the returned values (arrow from YANG specification to Variable store). The specification can be used to check the correctness of the CoAP request and do the CBOR encoding.

(2) SMIv2 specification: A named module specifies a set of variables and "conceptual tables". Named variables have simple types. Conceptual tables are composed of typed named columns. The variable name and module name identify the variable uniquely. There is an algorithm to translate SMIv2 specifications to YANG specifications.

(3) CoAP request: The CoAP request needs a Universal Resource Identifier (URI) and the payload of the packet to send a request. The URI is composed of the schema, server, path and query and

looks like coap://entry.example.com/<path>?<query>. Fragments are not supported. Allowed operations are PUT, PATCH, GET, DELETE, and POST. New variables can be created with POST when they exist in the YANG specification. The Observe option is used to return variable values regularly or on event occurrence (notification).

(3.1) CoAP <path>: The path identifies the variable in the form "/mg/<hash-value>".

(3.2) CoAP <query>: The query parameter is used to specify additional (optional) aspects like the container name, list instance, and others. The idea is to keep the path simple and put variations on variable specification in the query.

(3.3) CoAP discovery: Discovery of the variables is done with standard CoAP resource discovery using /.well-known/core with ?rt=/core.mg.

(4) Network packet: The payload contains the CBOR encoding of JSON objects. This object corresponds with the converted RESTCONF message payload.

(5) Retrieval, modification: The server needs to parse the CBOR encoded message and identify the corresponding instances in the Variable store. In addition, this component includes the code for CoAP Observe and block options.

(6) Variable store: The store is composed of two parts: Operational state and Configuration data-store (see Section 2.1). CoMI does not differentiate between variable store types. The Variable store contains data-node instances. Values are stored in the appropriate instances, and or values are returned from the instances into the payload of the packet.

(7) Variable instrumentation: This code depends on implementation of drivers and other node specific aspects. The Variable instrumentation code stores the values of the parameters into the appropriate places in the operational code. The variable instrumentation code reads current execution values from the operational code and stores them in the appropriate instances.

(8) Security: The server MUST prevent unauthorized users from reading or writing any data resources. CoMI relies on DTLS [RFC6347] which is specified to secure CoAP communication.

2.1. RESTCONF/YANG Architecture

CoMI adapts the RESTCONF architecture so data exchange and implementation requirements are optimized for constrained devices.

The RESTCONF protocol uses a unified data-store to edit conceptual data structures supported by the server. The details of transaction preparation and non-volatile storage of the data are hidden from the RESTCONF client. CoMI also uses a unified data-store, to allow stateless editing of configuration variables and the notification of operational variables.

The child schema nodes of the unified data-store include all the top-level YANG data nodes in all the YANG modules supported by the server. The YANG data structures represent a hierarchy of data resources. The client discovers the list of YANG modules, and important conformance information such as the module revision dates, YANG features supported, and YANG deviations required. The individual data nodes are discovered indirectly by parsing the YANG modules supported by the server.

The YANG data definition statements contain a lot of information that can help automation tools, developers, and operators use the data model correctly and efficiently. The YANG definitions and server YANG module capability advertisements provide an "API contract" that allow a client to determine the detailed server management capabilities very quickly. CoMI allows access to the same data resources as a RESTCONF server, except that messages are optimized to reduce identifier and payload size.

RESTCONF uses a simple algorithmic mapping from YANG to URI syntax to identify the target resource of a retrieval or edit operation. A client can construct operations or scripts using a predictable syntax, based on the YANG data definitions. The target resource URI can reference a data resource instance, or the data-store itself (to retrieve the entire data-store or create a top-level data resource instance). CoMI uses a compression algorithm to reduce the size of the data-node instance identifier (see Section 2.2.

2.2. Compression of data-node instance identifier

The RESTCONF protocol uses the full path of the desired data resource in the target resource URI. The JSON encoding will include the module name string to specify the YANG module. If a representation of the target resource is included in the request or response message in RESTCONF messages, then the data definition name string is used to identify each node in the message. The module namespace (or name) may also be present in these identifiers.

In order to greatly reduce the size of identifiers used in CoMI, numeric object identifiers are used instead of these strings. The specific encoding of the object identifiers is not hard-wired in the protocol.

YANG Hash is the default encoding for object identifiers. This encoding in considered to be "unstructured" since the particular values for each object are determined by a hash algorithm. It is possible for 2 different objects to generate the same hash value. If this occurs, then the client and server will both need to rehash the colliding object identifiers to new unused hash values.

In order to eliminate the need for rehashing, CoMI allows for alternate "structured" object identifier encoding formats. Structured object identifier MUST be managed such that no object ID collisions are possible, and therefore no rehash procedures are needed. Structured object identifiers can also be selected to minimize the size of a subset of the object identifiers (e.g., the most requested objects).

3. CoAP Interface

In CoAP a group of links can constitute a Function Set. The format of the links is specified in [I-D.ietf-core-interfaces]. This note specifies a Management Function Set. CoMI end-points that implement the CoMI management protocol support at least one discoverable management resource of resource type (rt): core.mg, with path: /mg, where mg is short-hand for management. The name /mg is recommended but not compulsory (see Section 4.5).

The path prefix /mg has resources accessible with the following five paths:

/mg: YANG-based data with path "/mg" and using CBOR content encoding format. This path represents a data-store resource which contains YANG data resources as its descendant nodes. All identifiers referring to YANG data nodes within this path are encoded as YANG hash values (see Section 5.7).

/mg/mod.uri: URI identifying the location of the server module information, with path "/mg/mod.uri" and CBOR content format. This YANG data is encoded with plain identifier strings, not YANG hash values. An EntityTag MUST be maintained for this resource by the server, which MUST be changed to a new value when the set of YANG modules in use by the server changes.

/mg/num.typ: String identifying the object ID numbering scheme used by the CoMI server. The only value defined in this document is

'yanghash' to indicate that the YANG Hash numbering scheme defined in this document is used. It is possible for other object numbering schemes to be defined outside the scope of this document.

/mg/srv.typ: String identifying the CoMI server type. The value 'ro' indicates that the server is a read-only server and no editing operations are supported. A read-only server is not required to provide YANG deviation statements for any writable YANG data nodes. The value 'rw' indicates that the server is a read-write server and editing operations are supported. A read-write server is required to provide YANG deviation statements for any writable YANG data nodes that are not fully implemented.

/mg/yh.uri: URI indicating the location of the server YANG hash information if any objects needed to be re-hashed by the server. It has the path "/mg/yh.uri" and is encoded in CBOR format. The "yang-hash" container within the "ietf-yang-hash" module of Section 5.4 is used to define the syntax and semantics of this data structure. This YANG data is encoded with plain identifier strings, not YANG hash values. The server will only have this resource if there are any objects that needed to be re-hashed due to a hash collision. If a client requests a node that has been re-hashed, then a rehash error is returned, according to the procedure in Section 5.2.

/mg/stream: String identifying the default stream resource to which YANG notification instances are appended. Notification support is optional, so this resource will not exist if the server does not support any notifications.

The mapping of YANG data node instances to CoMI resources is as follows: A YANG module describes a set of data trees composed of YANG data nodes. Every root of a data tree in a YANG module loaded in the CoMI server represents a resource of the server. All data root descendants represent sub-resources.

The resource identifiers of the instances of the YANG specifications are YANG hash values, as described in Section 5.1. When multiple instances of a list node exist, the instance selection is described in Section 4.1.3.4

The profile of the management function set, with IF=core.mg, is shown in the table below, following the guidelines of [I-D.ietf-core-interfaces]:

name	path	rt	Data Type
Management	/mg	core.mg	n/a
Data	/mg	core.mg.data	application/cbor
Module Set URI	/mg/mod.uri	core.mg.moduri	application/cbor
Numbering Type	/mg/num.typ	core.mg.num-type	application/cbor
Server Type	/mg/srv.typ	core.mg.srv-type	application/cbor
YANG Hash Info	/mg/yh.uri	core.mg.yang-hash	application/cbor
Events	/mg/stream	core.mg.stream	application/cbor

4. MG Function Set

The MG Function Set provides a CoAP interface to perform a subset of the functions provided by RESTCONF.

A subset of the operations defined in RESTCONF are used in CoMI:

Operation	Description
GET	Retrieve the data-store resource or a data resource
POST	Create a data resource
PUT	Create or replace a data resource
PATCH	Replace a data resource partially
DELETE	Delete a data resource

4.1. Data Retrieval

4.1.1. GET

One or more instances of data resources are retrieved by the client with the GET method. The RESTCONF GET operation is supported in CoMI. The same constraints apply as defined in section 3.3 of [I-D.ietf-netconf-restconf]. The operation is mapped to the GET method defined in section 5.8.1 of [RFC7252].

It is possible that the size of the payload is too large to fit in a single message. In the case that management data is bigger than the maximum supported payload size, the Block mechanism from [I-D.ietf-core-block] is used, as explained in more detail in Section 4.4.

There are two query parameters for the GET method. A CoMI server MUST implement the keys parameter and MAY implement the select parameter to allow common data retrieval filtering functionality.

Query Parameter	Description
keys	Request to select instances of a YANG definition
select	Request selected sub-trees from the target resource

The "keys" parameter is used to specify a specific instance of the list resource. When keys is not specified, all instances are returned. When no or one instance of the resource exists, the keys parameter is ignored.

4.1.2. Mapping of the 'select' Parameter

RESTCONF uses the 'select' parameter to specify an expression which can represent a subset of all data nodes within the target resource [I-D.ietf-netconf-restconf]. This parameter is useful for filtering sub-trees and retrieving only a subset that a managing application is interested in.

However, filtering is a resource intensive task and not all constrained devices can be expected to have enough computing resources such that they will be able to successfully filter and return a subset of a sub-tree. This is especially likely to be true with Class 0 devices that have significantly lesser RAM than 10 KiB [RFC7228]. Since CoMI is targeted at constrained devices and

networks, only a limited subset of the 'select' parameter is used here.

Unlike the RESTCONF 'select' parameter, CoMI does not use object names in "XPath" or "path-expr" format to identify the subset that needs to be filtered. Parsing XML is resource intensive for constrained devices [management] and using object names can lead to large message sizes. Instead, CoMI utilizes the YANG hashes described in Section 5 to identify the sub-trees that should be filtered from a target resource. Using these hashes ensures that a constrained node can identify the target sub-tree without expending many resources and that the messages generated are also efficiently encoded.

The implementation of the 'select' parameter is already optional for constrained devices, however, even when implemented it is expected to be a best effort feature, rather than a service that nodes must provide. This implies that if a node receives the 'select' parameter specifying a set of sub-trees that should be returned, it will only return those that it is able to return.

4.1.3. Retrieval Examples

In all examples the path is expressed in readable names and as a hash value of the name (where the hash value in the payload is expressed as a hexadecimal number, and the hash value in the URL as a base64 number). CoMI payloads use the CBOR format. The CBOR syntax of the YANG payloads is specified in Section 5. The examples in this section use a JSON payload with extensions to approach the permissible CBOR payload. Appendix C shows the CBOR format of some of the examples.

4.1.3.1. Single instance retrieval

A request to read the values of instances of a management object or the leaf of an object is sent with a confirmable CoAP GET message. A single object is specified in the URI path prefixed with /mg.

Using for example the clock container from [RFC7317], a request is sent to retrieve the value of clock/current-datetime specified in module system-state. The answer to the request returns a (identifier, value) pair, transported as a CBOR map with a single item.

```
REQ: GET example.com/mg/system-state/clock/current-datetime

RES: 2.05 Content (Content-Format: application/cbor)
{
   "current-datetime" : "2014-10-26T12:16:31Z"
}
```

The YANG hash value for 'current-datetime' is calculated by constructing the schema node identifier for the object:

```
/ietf-system:system-state/clock/current-datetime
```

The 30 bit murmur3 hash value (see Section 5.1) is calculated on this string with hash: 0x047c468b and EfEaM. The request using this hash value is shown below:

```
REQ: GET example.com/mg/EfEaM

RES: 2.05 Content (Content-Format: application/cbor)
{
    0x047c468b : "2014-10-26T12:16:31Z"
}
```

The specified object can be an entire object. Accordingly, the returned payload is composed of all the leaves associated with the object. The payload is a CBOR map where each leaf is returned as a (YANG hash, value) pair. For example, the GET of the clock object, sent by the client, results in the following returned payload sent by the managed entity, transported as A CBOR map with two items:

```
REQ: GET example.com/mg/system-state/clock
   (Content-Format: application/cbor)

RES: 2.05 Content (Content-Format: application/cbor)
{
  "clock" : {
      "current-datetime" : "2014-10-26T12:16:51Z",
      "boot-datetime" : "2014-10-21T03:00:00Z"
  }
}
```

The YANG hash values for 'clock', 'current-datetime', and 'boot-datetime' are calculated by constructing the schema node identifier for the objects, and then calculating the 30 bit murmur3 hash values (shown in parenthesis):

```
/ietf-system:system-state/clock (0x021ca491 and CDKSQ)
/ietf-system:system-state/clock/current-datetime (0x047c468b)
/ietf-system:system-state/clock/boot-datetime (0x1fb5f4f8)
```

The request using the hash values is shown below:

```
REQ: GET example.com/mg/CDKSQ
   (Content-Format: application/cbor)

RES: 2.05 Content (Content-Format: application/cbor)
{
  0x021ca491 : {

      0x047c468b : "2014-10-26T12:16:51Z",
      0x1fb5f4f8 : "2014-10-21T03:00:00Z"
   }
}
```

The corresponding CBOR code can be found in Appendix C.

4.1.3.2. Multiple instance retrieval

A "list" node can have multiple instances. Accordingly, the returned payload is composed of all the instances associated with the list node. Each instance is returned as a (key, object) pair, where key and object are composed of one or more (identifier, value) pairs.

For example, the GET of the /interfaces/interface/ipv6/neighbor results in the following returned payload sent by the managed entity, transported as a CBOR map of 3 (key : object) pairs, where key and value are CBOR maps with one entry each. In this case the key is the "ip" attribute and the value is the "link-layer-address" attribute.

```
   REQ: GET example.com/mg/interfaces/interface/ipv6/neighbor
      (Content-Format: application/cbor)

   RES: 2.05 Content (Content-Format: application/cbor)
   {
      "neighbor" :{
        {"ip" : "fe80::200:f8ff:fe21:67cf"}:
           {"link-layer-address" : "00:00::10:01:23:45"}
        ,
        {"ip" : "fe80::200:f8ff:fe21:6708"}:
           {"link-layer-address" : "00:00::10:54:32:10"}
        ,
        {"ip" : "fe80::200:f8ff:fe21:88ee"}:
           {"link-layer-address" : "00:00::10:98:76:54"}
      }
   }
```

The YANG hash values for 'neighbor', 'ip', and 'link-layer-address' are calculated by constructing the schema node identifier for the objects, and then calculating the 30 bit murmur3 hash values (shown in parenthesis):

```
/ietf-interfaces:interfaces/interface/ietf-ip:ipv6/neighbor
    (0x2445e478 and kReR4)
/ietf-interfaces:interfaces/interface/ietf-ip:ipv6/neighbor/ip
    (0x2283ed40 and ig-la)
/ietf-interfaces:interfaces/interface/ietf-ip:ipv6/neighbor/
    link-layer-address  (0x3d6915c7)
```

The request using the hash values is shown below:

```
   REQ: GET example.com/mg/kReR4
      (Content-Format: application/cbor)

   RES: 2.05 Content (Content-Format: application/cbor)
   {
      0x2445e478 : {
        {0x2283ed40 : "fe80::200:f8ff:fe21:67cf"}:
           {0x3d6915c7 : "00:00::10:01:23:45"}
        ,
        {0x2283ed40 : "fe80::200:f8ff:fe21:6708"}:
           {0x3d6915c7 : "00:00::10:54:32:10"}
        ,
        {0x2283ed40 : "fe80::200:f8ff:fe21:88ee"}:
           {0x3d6915c7 : "00:00::10:98:76:54"}
      }
   }
```

4.1.3.3. Access to MIB Data

The YANG translation of the SMI specifying the ipNetToMediaTable [RFC4293] yields:

```
container IP-MIB {
  container ipNetToPhysicalTable {
    list ipNetToPhysicalEntry {
       key "ipNetToPhysicalIfIndex
            ipNetToPhysicalNetAddressType
            ipNetToPhysicalNetAddress";
       leaf ipNetToMediaIfIndex {
          type: int32;
       }
       leaf ipNetToPhysicalIfIndex {
         type if-mib:InterfaceIndex;
       }
       leaf ipNetToPhysicalNetAddressType {
         type inet-address:InetAddressType;
       }
       leaf ipNetToPhysicalNetAddress {
         type inet-address:InetAddress;
       }
       leaf ipNetToPhysicalPhysAddress {
         type yang:phys-address {
            length "0..65535";
         }
       }
       leaf ipNetToPhysicalLastUpdated {
         type yang:timestamp;
       }
       leaf ipNetToPhysicalType {
         type enumeration { ... }
       }
       leaf ipNetToPhysicalState {
         type enumeration { ... }
       }
       leaf ipNetToPhysicalRowStatus {
         type snmpv2-tc:RowStatus;
       }
    }
 }
```

The following example shows an "ipNetToPhysicalTable" with 2 instances, using JSON encoding as defined in [I-D.ietf-netmod-yang-json]:

```
{
  "IP-MIB/ipNetToPhysicalTable/ipNetToPhysicalEntry" : [
        {
          "ipNetToPhysicalIfIndex" : 1,
          "ipNetToPhysicalNetAddressType" : "ipv4",
          "ipNetToPhysicalNetAddress" : "10.0.0.51",
          "ipNetToPhysicalPhysAddress" : "00:00:10:01:23:45",
          "ipNetToPhysicalLastUpdated" : "2333943",
          "ipNetToPhysicalType" : "static",
          "ipNetToPhysicalState" : "reachable",
          "ipNetToPhysicalRowStatus" : "active"
        },
        {
          "ipNetToPhysicalIfIndex" : 1,
          "ipNetToPhysicalNetAddressType" : "ipv4",
          "ipNetToPhysicalNetAddress" : "9.2.3.4",
          "ipNetToPhysicalPhysAddress" : "00:00:10:54:32:10",
          "ipNetToPhysicalLastUpdated" : "2329836",
          "ipNetToPhysicalType" : "dynamic",
          "ipNetToPhysicalState" : "unknown",
          "ipNetToPhysicalRowStatus" : "active"
        }
      ]
    }
  }
}
```

The YANG hash values for 'ipNetToPhysicalEntry' and its child nodes are calculated by constructing the schema node identifier for the objects, and then calculating the 30 bit murmur3 hash values (shown in parenthesis):

```
/IP-MIB:IP-MIB/ipNetToPhysicalTable (0x0aba15cc and kuhXM)
/IP-MIB:IP-MIB/ipNetToPhysicalTable/ipNetToPhysicalEntry
   (0xo6aaddbc and Gqt28)
/IP-MIB:IP-MIB/ipNetToPhysicalTable/ipNetToPhysicalEntry/
   ipNetToPhysicalIfIndex (0x346b3071)
/IP-MIB:IP-MIB/ipNetToPhysicalTable/ipNetToPhysicalEntry/
   ipNetToPhysicalNetAddressType (0x3650bb64)
/IP-MIB:IP-MIB/ipNetToPhysicalTable/ipNetToPhysicalEntry/
   ipNetToPhysicalNetAddress (0x06fd4d91)
/IP-MIB/ipNetToPhysicalTable/ipNetToPhysicalEntry/
   ipNetToPhysicalPhysAddress (0x26180bcb)
/IP-MIB:IP-MIB/ipNetToPhysicalTable/ipNetToPhysicalEntry/
   ipNetToPhysicalLastUpdated (0x3d6bbe90)
/IP-MIB:IP-MIB/ipNetToPhysicalTable/ipNetToPhysicalEntry/
   ipNetToPhysicalType (0x35ecbb3d)
/IP-MIB:IP-MIB/ipNetToPhysicalTable/ipNetToPhysicalEntry/
   ipNetToPhysicalState (0x13038bb5)
/IP-MIB:IP-MIB/ipNetToPhysicalTable/ipNetToPhysicalEntry/
   ipNetToPhysicalRowStatus (0x09e1fa37)
```

The following example shows a request for the entire ipNetToPhysicalTable. The payload is a CBOR map composed of two (key , object} pairs, where key and object are CBOR maps, composed of 3 and 5 (identifier, value) pairs respectively.

```
REQ: GET example.com/mg/Gqt28

RES: 2.05 Content (Content-Format: application/cbor)
{
    0x06aaddbc: {
        {
          0x346b3071 : 1,
          0x3650bb64 : "ipv4",
          0x06fd4d91 : "10.0.0.51"}:
        {
          0x26180bcb : "00:00:10:01:23:45",
          0x3d6bbe90 : "2333943",
          0x35ecbb3d : "static",
          0x13038bb5 : "reachable",
          0x09e1fa37 : "active"
        },
        {
          0x346b3071 : 1,
          0x3650bb64 : "ipv4",
          0x06fd4d91 : "9.2.3.4"}:
        {
          0x26180bcb : "00:00:10:54:32:10",
          0x3d6bbe90 : "2329836",
          0x35ecbb3d : "dynamic",
          0x13038bb5 : "unknown",
          0x09e1fa37 : "active"
        }
     }
}
```

The corresponding CBOR code can be found in Appendix C.

4.1.3.4. The 'keys' Query Parameter

There is a query parameter that MUST be supported by servers called "keys". This parameter is used to specify the key values for an instance of an object identified by a YANG hash value. All key leaf values of the instance are passed in order. The first key leaf in the top-most list is the first key encoded in the 'keys' parameter.

The key leafs from top to bottom and left to right are encoded as a comma-delimited list. If a key leaf value is missing then all values for that key leaf are returned.

Example: In this example exactly 1 instance is requested from the ipNetToPhysicalEntry (from a previous example). The CBOR payload is constructed as before.

```
REQ: GET example.com/mg/Gqt28?keys=1,ipv4,10.0.0.51

RES: 2.05 Content (Content-Format: application/cbor)
{
   0x06aaddbc: {
      {
          0x346b3071 : 1,
          0x3650bb64 : "ipv4",
          0x06fd4d91 : "9.2.3.4"}:
        {
          0x26180bcb : "00:00:10:54:32:10",
          0x3d6bbe90 : "2329836",
          0x35ecbb3d : "dynamic",
          0x13038bb5 : "unknown",
          0x09e1fa37 : "active"
        }
    }
}
```

An example illustrates the syntax of keys query parameter. In this example the following YANG module is used:

```
  module foo-mod {
    namespace foo-mod-ns;
    prefix foo;

    list A {
      key "key1 key2";
      leaf key1 { type string; }
      leaf key2 { type int32; }
      list B {
        key "key3";
        leaf key3 { type string; }
        leaf col1 { type uint32; }
      }
    }
  }
```

The path identifier for the leaf "col1" is the following string:

```
   /foo-mod:A/B/col1
```

The YANG hash for this identifier string has values: 0x189295aa and YkpWq).

The following string represents the CoMI target resource identifier for the instance of the "col1" leaf with key values "top", 17, "group":

```
/mg/YkpWq?keys="top",17,"group1"
```

4.1.3.5. The 'select' Query Parameter

The select parameter is used along with the GET method to provide a sub-tree filter mechanism. A list of YANG hashes that should be filtered is provided along with a list of keys identifying the instances that should be returned. When the keys parameter is used together with the select, the key values are added in brackets without using the "keys=" text.

The following example shows an "ipNetToPhysicalTable" (from a previous example) with 4 instances, using JSON encoding following [I-D.ietf-netmod-yang-json]:

```
{
  "IP-MIB/ipNetToPhysicalTable/ipNetToPhysicalEntry" : [
        {
          "ipNetToPhysicalIfIndex" : 1,
          "ipNetToPhysicalNetAddressType" : "ipv4",
          "ipNetToPhysicalNetAddress" : "10.0.0.51",
          "ipNetToPhysicalPhysAddress" : "00:00:10:01:23:45",
          "ipNetToPhysicalLastUpdated" : "2333943",
          "ipNetToPhysicalType" : "static",
          "ipNetToPhysicalState" : "reachable",
          "ipNetToPhysicalRowStatus" : "active"
        },
        {
          "ipNetToPhysicalIfIndex" : 3,
          "ipNetToPhysicalNetAddressType" : "ipv4",
          "ipNetToPhysicalNetAddress" : "9.2.3.4",
          "ipNetToPhysicalPhysAddress" : "00:00:10:54:32:10",
          "ipNetToPhysicalLastUpdated" : "2329836",
          "ipNetToPhysicalType" : "dynamic",
          "ipNetToPhysicalState" : "unknown",
          "ipNetToPhysicalRowStatus" : "active"
        },
        {
          "ipNetToPhysicalIfIndex" : 2,
          "ipNetToPhysicalNetAddressType" : "ipv4",
          "ipNetToPhysicalNetAddress" : "10.24.2.53",
          "ipNetToPhysicalPhysAddress" : "00:00:10:28:19:CA",
          "ipNetToPhysicalLastUpdated" : "2124368",
          "ipNetToPhysicalType" : "static",
          "ipNetToPhysicalState" : "unknown",
          "ipNetToPhysicalRowStatus" : "active"
        },
        {
          "ipNetToPhysicalIfIndex" : 1,
          "ipNetToPhysicalNetAddressType" : "ipv4",
          "ipNetToPhysicalNetAddress" : "192.168.2.12",
          "ipNetToPhysicalPhysAddress" : "00:00:10:29:11:32",
          "ipNetToPhysicalLastUpdated" : "1925384",
          "ipNetToPhysicalType" : "dynamic",
          "ipNetToPhysicalState" : "reachable",
          "ipNetToPhysicalRowStatus" : "active"
        }
      ]
    }
  }
}
```

Data may be retrieved using the select query parameter in the following way, transported as a CBOR maps of maps:

REQ: GET example.com/mg/?select=Gqt28(1,ipv4)

```
RES: 2.05 Content (Content-Format: application/cbor)
{
    0x06aaddbc: {
      {
          0x346b3071 : 1,
          0x3650bb64 : "ipv4",
          0x06fd4d91 : "10.0.0.51"}:
        {
          0x26180bcb : "00:00:10:01:23:45",
          0x3d6bbe90 : "2333943",
          0x35ecbb3d : "static",
          0x13038bb5 : "reachable",
          0x09e1fa37 : "active"
        },
        {
          0x346b3071 : 1,
          0x3650bb64 : "ipv4",
          0x06fd4d91 : "192.168.2.12"}:
        {
          0x26180bcb : "00:00:10:29:11:32",
          0x3d6bbe90 : "1925384",
          0x35ecbb3d : "dynamic",
          0x13038bb5 : "reachable",
          0x09e1fa37 : "active"
        }
   }
}
```

In this example exactly 2 instances are returned as response from the ipNetToPhysicalTable because both those instances match the provided keys.

Supposing there were multiple YANG hashes with their own sets of keys that were to be filtered, the select query parameter can be used to retrieve results from these in one go as well. The following string represents the CoMI target resource identifier when multiple YANG hashes, with their own sets of keys are queried:

```
    /mg/?select=hash1(hash1-key1,hash1-key2,...),hash2(hash2-key1)...
```

4.2. Data Editing

CoMI allows data-store contents to be created, modified and deleted using CoAP methods.

Data-editing is an optional feature. The server will indicate its editing capability with the "/core.mg.srv-type resource type. If the value is 'rw' then the server supports editing operations. If the value is 'ro' then the server does not support editing operations.

4.2.1. Data Ordering

A CoMI server is not required to support entry insertion of lists and leaf-lists that are ordered by the user (i.e., YANG statement "ordered-by user"). The 'insert' and 'point' query parameters from RESTCONF are not used in CoMI.

A CoMI server SHOULD preserve the relative order of all user-ordered list and leaf-list entries that are received in a single edit request. These YANG data node types are encoded as arrays so messages will preserve their order.

4.2.2. POST

Data resource instances are created with the POST method. The RESTCONF POST operation is supported in CoMI, however it is only allowed for creation of data resources. The same constraints apply as defined in section 3.4.1 of [I-D.ietf-netconf-restconf]. The operation is mapped to the POST method defined in section 5.8.2 of [RFC7252].

There are no query parameters for the POST method.

4.2.3. PUT

Data resource instances are created or replaced with the PUT method. The PUT operation is supported in CoMI. A request to set the values of instances of an object/leaf is sent with a confirmable CoAP PUT message. The Response is piggybacked to the CoAP ACK message corresponding with the Request. The same constraints apply as defined in section 3.5 of [I-D.ietf-netconf-restconf]. The operation is mapped to the PUT method defined in section 5.8.3 of [RFC7252].

There are no query parameters for the PUT method.

4.2.4. PATCH

Data resource instances are partially replaced with the PATCH method [I-D.vanderstok-core-patch]. The PATCH operation is supported in CoMI. A request to set the values of instances of a subset of the values of the resource is sent with a confirmable CoAP PATCH message. The Response is piggybacked to the CoAP ACK message corresponding with the Request. The same constraints apply as defined in section 3.5 of [I-D.ietf-netconf-restconf]. The operation is mapped to the PATCH method defined in [I-D.vanderstok-core-patch].

The processing of the PATCH command is specified by the CBOR payload. The CBOR patch payload describes the changes to be made to target YANG data nodes. It follows closely the rules described in [RFC7396]. If the CBOR patch payload contains objects that are not present in the target, these objects are added. If the target contains the specified object, the contents of the objects are replaced with the values of the payload. Null values indicate the removal of existing values. The CBOR patch extends [RFC7396] by specifying rules for list elements.

For example consider the following YANG specification:

```
module foo {
  namespace "http://example.com/book";
  prefix "bo";
  revision 2015-06-07;

  list B {
      key key1;
      key key2;
      leaf key1 { type string; }
      leaf key2 {type string; }
      leaf col1 { type int32; }
      leaf counter1 { type uint32; }
  }

  container book {
    leaf title { type string; }
    container author {
      leaf  givenName {type string; }
      leaf  familyName {type string; }
    }
    leaf-list tags {type string; }
    leaf content{type string;}
    leaf phoneNumber {type string;}
  }
```

Consider the following target data nodes described with the JSON encoding of [I-D.ietf-netmod-yang-json].

```
    "B": [
        {
         "key1" : "author1",
         "key2" : "book2",
         "col1" : 25,
         "counter1" : 4321
        },
       {
         "key1" : "author5",
         "key2" : "book6",
         "col1" : 2,
         "counter1" : 1234
        }
     ]


    "book": {
     "title" : "mytitle",
     "author": {
       "givenName" : "John",
       "familyName" : "Doe"
       }
     "tags" : [ "example", "sample"],
     "content" : "This will be unchanged"
   }
```

The following changes are requested for the document (following the example from [RFC7396]: the title changes from "mytitle" to "favoured", the phoneNumber is added to the book container, the familyName is deleted, and "sample" is removed from the tags leaf-list. In addition author1, book1 item is removed, author5 counter1 is upgraded, and a new author is added in B list. The following CBOR Patch payload, represented in JSON is sent: (for the CBOR contents see Appendix C.

```
 {
    "B":  {
     { "key1" : "author1",
      "key2" : "book2"}:
      { null : null},
     { "key1" : "author5"} :
       {"counter1" : 4444},
     { "key1" : "newauthor",
      "key2" : "newbook"}:
     { "col1" : 1,
       "counter1" : 1}
    },
  "book" : {
     "title" : "favoured",
     "author": {"familyName" : null},
     "tags" : [ "example"],
     "phoneNumber" : "+01-123-456-7890"
   }
}
```

In his example, the value "author5" specifies the entry uniquely. However, when several entries exist with the "author5" value for "key1", the outcome of the example Patch is undefined.

The processing of the Patch payload results in the following new target data nodes.

```
    "B": [
        {
         "key1" : "newauthor",
         "key2" : "newbook",
         "col1" : 1,
         "counter1" : 1
        },
       {
         "key1" : "author5",
         "key2" : "book6",
         "col1" : 2,
         "counter1" : 4444
        }
     ]

   "book": {
     "title" : "favoured",
     "author": {
       "givenName" : "John"
       }
     "tags" : [ "example"],
     "content" : "This will be unchanged",
     "phoneNumber" : +01-123-456-7890"
   }
```

There are no query parameters for the PATCH method.

4.2.5. DELETE

Data resource instances are deleted with the DELETE method. The RESTCONF DELETE operation is supported in CoMI. The same constraints apply as defined in section 3.7 of [I-D.ietf-netconf-restconf]. The operation is mapped to the DELETE method defined in section 5.8.4 of [RFC7252].

There are no optional query parameters for the DELETE method.

4.2.6. Editing Multiple Resources

Editing multiple data resources at once can allow a client to use fewer messages to make a configuration change. It also allows multiple edits to all be applied or none applied, which is not possible if the data resources are edited one at a time.

It is easy to add multiple entries at once. The "PATCH" method can be used to simply patch the parent node(s) of the data resources to be added. If multiple top-level data resources need to be added, then the data-store itself ('/mg') can be patched.

If other operations need to be performed, or multiple operations need to be performed at once, then the YANG Patch [I-D.ietf-netconf-yang-patch] media type can be used with the PATCH method. A YANG patch is an ordered list of edits on the target resource, which can be a specific data node instance, or the data-store itself. The resource type used by YANG Patch is 'application/yang.patch'. A status message is returned in the response, using resource type 'application/yang.patch.status'.

The following YANG tree diagram describes the YANG Patch structure, Each 'edit' list entry has its own operation, sub-resource target, and new value (if needed).

```
  +--rw yang-patch
      +--rw patch-id?   string
      +--rw comment?    string
      +--rw edit* [edit-id]
         +--rw edit-id      string
         +--rw operation    enumeration
         +--rw target       target-resource-offset
         +--rw point?       target-resource-offset
         +--rw where?       enumeration
         +--rw value
```

The YANG Hash values for the YANG Patch request objects are calculated as follows:

```
0x2c3f93c7: /ietf-yang-patch:yang-patch
0x2fb8873e: /ietf-yang-patch:yang-patch/patch-id
0x011640f0: /ietf-yang-patch:yang-patch/comment
0x16804b72: /ietf-yang-patch:yang-patch/edit
0x2bd93228: /ietf-yang-patch:yang-patch/edit/edit-id
0x1959d8c9: /ietf-yang-patch:yang-patch/edit/operation
0x1346e0aa: /ietf-yang-patch:yang-patch/edit/target
0x0750e196: /ietf-yang-patch:yang-patch/edit/point
0x0b45277e: /ietf-yang-patch:yang-patch/edit/where
0x2822c407: /ietf-yang-patch:yang-patch/edit/value
```

Refer to [I-D.ietf-netconf-yang-patch] for more details on the YANG Patch request and response contents.

4.3. Notify functions

Notification by the server to a selection of clients when an event occurs in the server is an essential function for the management of servers. CoMI allows events specified in YANG [RFC5277] to be notified to a selection of requesting clients. The server appends newly generated events to a stream. There is one, so-called "default", stream in a CoMI server. The /mg/stream resource identifies the default stream. The server MAY create additional streams. When a CoMI server generates an internal event, it is appended to the chosen stream, and the contents of a notification instance is ready to be sent to all CoMI clients which observe the chosen stream resource.

Reception of generated notification instances is enabled with the CoAP Observe [I-D.ietf-core-observe] function. The client subscribes to the notifications by sending a GET request with an "Observe" option, specifying the /mg/stream resource when the default stream is selected.

Every time an event is generated, the chosen stream is cleared, and the generated notification instance is appended to the chosen stream. After appending the instance, the contents of the instance is sent to all clients observing the modified stream.

Suppose the server generates the event specified with:

```
module example-port {
  ...
  prefix ep;
  ...
  notification example-port-fault {
    description
      "Event generated if a hardware fault on a
       line card port is detected";
    leaf port-name {
      type string;
      description "Port name";
    }
    leaf port-fault {
      type string;
      description "Error condition detected";
    }
  }
}

}
```

The YANG Hash values for this notification are assigned as follows:

```
0x3fe84d89: /example-port:example-port-fault
0x2921ba9e: /example-port:example-port-fault/port-name
0x2d452885: /example-port:example-port-fault/port-fault
0x11287619 (RKHUZ) : /stream


}
```

By executing a GET on the /mg/stream resource the client receives the following response:

```
REQ: GET example.com/mg/stream
    (observe option register)

RES: 2.05 Content (Content-Format: application/cbor)
{
      "example-port-fault":{
            "port-name" : "0/4/21",
            "port-fault" : "Open pin 2"
      }
}
```

Replacing the names by the hash values leads to:

```
REQ: GET example.com/mg/RKHUZ
    (observe option register)

RES: 2.05 Content (Content-Format: application/cbor)
{
    0x3fe84d89 : {
      0x2921ba9e : "0/4/21",
      0x2d452885 : "Open pin 2"
    }
}
```

In the example, the request returns a success response with the contents of the last generated event. Consecutively the server will regularly notify the client when a new event is generated.

To check that the client is still alive, the server MUST send confirmable notifications once in a while. When the client does not confirm the notification from the server, the server will remove the client from the list of observers [I-D.ietf-core-observe].

In the registration request, the client MAY include a "Response-To-Uri-Host" and optionally "Response-To-Uri-Port" option as defined in [I-D.becker-core-coap-sms-gprs]. In this case, the observations SHOULD be sent to the address and port indicated in these options. This can be useful when the client wants the managed device to send the trap information to a multicast address.

4.4. Use of Block

The CoAP protocol provides reliability by acknowledging the UDP datagrams. However, when large pieces of text need to be transported the datagrams get fragmented, thus creating constraints on the resources in the client, server and intermediate routers. The block option [I-D.ietf-core-block] allows the transport of the total payload in individual blocks of which the size can be adapted to the underlying fragment sizes such as: (UDP datagram size ~64KiB, IPv6 MTU of 1280, IEEE 802.15.4 payload of 60-80 bytes). Each block is individually acknowledged to guarantee reliability.

The block size is specified as exponents of the power 2. The SZX exponent value can have 7 values ranging from 0 to 6 with associated block sizes given by 2**(SZX+4); for example SZX=0 specifies block size 16, and SZX=3 specifies block size 128.

The block number of the block to transmit can be specified. There are two block options: Block1 option for the request payload transported with PUT, POST or PATCH, and the block2 option for the response payload with GET. Block1 and block2 can be combined. Examples showing the use of block option in conjunction with observer options are provided in [I-D.ietf-core-block].

Notice that the Block mechanism splits the data at fixed positions, such that individual data fields may become fragmented. Therefore, assembly of multiple blocks may be required to process the complete data field.

Beware of race conditions. Blocks are filled one at a time and care should be taken that the whole data representation is sent in multiple blocks sequentially without interruption. In the server, values are changed, lists are re-ordered, extended or reduced. When these actions happen during the serialization of the contents of the variables, the transported results do not correspond with a state having occurred in the server; or worse the returned values are inconsistent. For example: array length does not correspond with actual number of items. It may be advisable to use CBOR maps or CBOR arrays of undefined length are foreseen for data streaming purposes.

4.5. Resource Discovery

The presence and location of (path to) the management data are discovered by sending a GET request to "/.well-known/core" including a resource type (RT) parameter with the value "core.mg" [RFC6690]. Upon success, the return payload will contain the root resource of the management data. It is up to the implementation to choose its root resource, but it is recommended that the value "/mg" is used, where possible. The example below shows the discovery of the presence and location of management data.

```
REQ: GET /.well-known/core?rt=core.mg

RES: 2.05 Content </mg>; rt="core.mg"
```

Management objects MAY be discovered with the standard CoAP resource discovery. The implementation can add the hash values of the object identifiers to /.well-known/core with rt="core.mg.data". The available objects identified by the hash values can be discovered by sending a GET request to "/.well-known/core" including a resource type (RT) parameter with the value "core.mg.data". Upon success, the return payload will contain the registered hash values and their location. The example below shows the discovery of the presence and location of management data.

```
REQ: GET /.well-known/core?rt=core.mg.data

RES: 2.05 Content </mg/BaAiN>; rt="core.mg.data",
</mg/CF_fA>; rt="core.mg.data"
```

Lists of hash values may become prohibitively long. It is discouraged to provide long lists of objects on discovery. Therefore, it is recommended that details about management objects are discovered by reading the YANG module information stored in the "ietf-yang-library" module [I-D.ietf-netconf-restconf]. The resource "/mg/mod.uri" is used to retrieve the location of the YANG module library.

The module list can be stored locally on each server, or remotely on a different server. The latter is advised when the deployment of many servers are identical.

```
Local in example.com server:

REQ: GET example.com/mg/mod.uri

RES: 2.05 Content (Content-Format: application/cbor)
{
  "mod.uri" : "example.com/mg/modules"
}


Remote in example-remote-server:

REQ: GET example.com/mg/mod.uri

RES: 2.05 Content (Content-Format: application/cbor)
{
  "moduri" : "example-remote-server.com/mg/group17/modules"
}
```

Within the YANG module library all information about the module is stored such as: module identifier, identifier hierarchy, grouping, features and revision numbers.

The hash identifier is obtained as specified in Section 5.1. When a collision occurred in the name space of the target server, a rehash is executed as explained in Section 5.2.

4.6. Error Return Codes

The RESTCONF return status codes defined in section 6 of the RESTCONF draft are used in CoMI error responses, except they are converted to CoAP error codes.

TODO: assign an error code for a rehash-error.

RESTCONF Status Line	CoAP Status Code
100 Continue	none?
200 OK	2.05
201 Created	2.01
202 Accepted	none?
204 No Content	?
304 Not Modified	2.03
400 Bad Request	4.00
403 Forbidden	4.03
404 Not Found	4.04
405 Method Not Allowed	4.05
409 Conflict	none?
412 Precondition Failed	4.12
413 Request Entity Too Large	4.13
414 Request-URI Too Large	4.00
415 Unsupported Media Type	4.15
500 Internal Server Error	5.00
501 Not Implemented	5.01
503 Service Unavailable	5.03

5. Mapping YANG to CoMI payload

A mapping for the encoding of YANG data in CBOR is necessary for the efficient transport of management data in the CoAP payload. Since object names may be rather long and may occur repeatedly, CoMI allows for association of a given object path identifier string value with an integer, called a "YANG hash".

5.1. YANG Hash Generation

The association between string value and string number is done through a hash algorithm. The 30 least significant bits of the "murmur3" 32-bit hash algorithm are used. This hash algorithm is described online at [murmur3]. Implementations are available online [murmur-imp]. When converting 4 input bytes to a 32-bit integer in the hash algorithm, the Little-Endian convention MUST be used.

The hash is generated for the string representing the object path identifier. A canonical representation of the path identifier is used.

The module name is used to identify the namespace of the object node. The prefix cannot be used because it is allowed to change over time. The module name is never allowed to change.

The module name MUST be present in the identifier for the first node in the object path identifier.

If a child node in the object path identifier is from the same module namespace as its parent, then the module-name MUST NOT be used in the identifier.

If a child node in the object path identifier is not from the same module namespace as its parent, then the module-name MUST be used in the identifier.

Choice and case node names are not included in the path expression. Only 'container', 'list', 'leaf', 'leaf-list', and 'anyxml' nodes are listed in the path expression.

The YANG Hash value is calculated for all data nodes in the module, even if the server only implements a subset of these objects. This includes all "data-def", "rpc", "notification", and external data nodes derived from "augment" statements.

The "murmur3_32" hash function is executed for the entire path string. The value '42' is used as the seed for the hash function. The YANG hash is subsequently calculated by taking the 30 least significant bits.

The resulting 30-bit number is used by the server, unless the value is already being used for a different object by the server. In this case, the re-hash procedure in the following section is executed.

Example: the following identifier is for the 'mtu' leaf in the ietf-interfaces module:

/ietf-interfaces:interfaces/interface/mtu

Example: the following identifier is for the 'ipv4' container in the ietf-ip module, which augments the 'interface' list in the ietf-interfaces module:

/ietf-interfaces:interfaces/interface/ietf-ip:ipv4

5.2. Re-Hash Error Procedure

In most cases, the hash function is expected to produce unique values for all the node names supported by a constrained device. Given a known set of YANG modules, both server and client can calculate the YANG hashes independently, and offline.

Even though collisions are expected to happen rather rarely, they need to be considered. Collisions can be detected before deployment, if the vendor knows which modules are supported by the server, and hence all YANG hashes can be calculated. Collisions are only an issue when they occur at the same server. The client needs to discover any re-hash mappings on a per server basis.

If the server needs to re-hash any object identifiers, then it MUST create a "rehash" entry for all its rehashed node names, as described in the following YANG module.

A re-hashed object identifier has the rehash bit set in the identifier, every time it is sent from the server to the client. This allows the client to identify nodes for which a "reverse rehash" entry needs to be retrieved. A client does not need to retrieve the rehash map before retrieving or altering data nodes.

If any node identifier provided by the client is not available because it has been rehashed, the server MUST return a rehash error, containing the 'rehash' entries for all the invalid nodes which were specified by the client.

It is possible that none of the node identifiers provided by the client in a GET method are invalid and rehashed, but rather one or more descendant nodes within the selected subtree(s) has been rehashed. In this case, a rehash error is not returned. Instead the requested subtree(s) are returned, and the rehash bit is set for any descendant node(s) that have been rehashed. The client will strip off the rehash bit and retrieve the 'revhash' entry for these nodes (if not already done).

5.3. Reverse Re-Hash Error Procedure

A hash collision occurs if two different path identifier strings have the same hash value. If the server has over 30,000 node names in its YANG modules, then the probability of a collision is 10% or higher, see Appendix E. If a hash collision occurs on the server, then the node name that is causing the conflict has to be altered, such that the new hash value does not conflict with any value already in use by the server.

5.4. ietf-yang-hash YANG Module

The "ietf-yang-hash" YANG module is used by the server to report any objects that have been mapped to produce a new hash value that does not conflict with any other YANG hash values used by the server.

YANG tree diagram for "ietf-yang-hash" module:

```
  +--ro yang-hash
      +--ro rehash* [hash]
         +--ro hash      uint32
         +--ro object*
            +--ro module     string
            +--ro newhash    uint32
            +--ro path?      string
```

<CODE BEGINS> file "ietf-yang-hash@2015-06-06.yang"

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

  organization
    "IETF CORE (Constrained RESTful Environments) Working Group";

  contact
    "WG Web:   <http://tools.ietf.org/wg/core/>
     WG List:  <mailto:core@ietf.org>

     WG Chair: Carsten Bormann
               <mailto:cabo@tzi.org>

     WG Chair: Andrew McGregor
               <mailto:andrewmcgr@google.com>
```

```
        Editor:   Peter van der Stok
                  <mailto:consultancy@vanderstok.org>

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

        Editor:   Juergen Schoenwaelder
                  <mailto:j.schoenwaelder@jacobs-university.de>

        Editor:   Anuj Sehgal
                  <mailto:s.anuj@jacobs-university.de>";

    description
      "This module contains re-hash information for the CoMI protocol.

       Copyright (c) 2015 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-vanderstok-core-comi-08.txt

    // RFC Ed.: update the date below with the date of RFC publication
    // and remove this note.
    revision 2015-09-24 {
      description
        "Initial revision.";
      reference
        "RFC XXXX: CoMI Protocol.";
    }

    container yang-hash {
      config false;
      description
        "Contains information on the YANG Hash values used by
         the server.";
```

```
      list rehash {
        key hash;
        description
          "Each entry describes an re-hash mapping in use by
           the server.";

        leaf hash {
          type uint32;
          description
            "The hash value that has a collision.  This hash value
             cannot be used on the server.  The rehashed
             value for each affected object must be used instead.";
        }

        list object {
          min-elements 2;

          description
            "Each entry identifies one of the objects involved in the
             hash collision and contains the rehash information for
             that object.";

          leaf module {
            type string;
            mandatory true;
            description
              "The module name identifying the module namespace
               for this object.";
          }

          leaf newhash {
            type uint32;
            mandatory true;
            description
              "The new hash value for this object. The rehash bit is
               not set in this value.";
          }

          leaf path {
            type string;
            description
              "The object path identifier string used in the original
               YANG hash calculation. This object MUST be included for
               any objects in the rehash entry with the same 'module'
               value.";
          }
        }
      }
```

```
    }

  }

  <CODE ENDS>
```

5.5. YANG Re-Hash Examples

In this example there are two YANG modules, "foo" and "bar".

```
  module foo {
    namespace "http://example.com/ns/foo";
    prefix "f";
    revision 2015-06-07;

    container A {
      list B {
        key name;
        leaf name { type string; }
        leaf counter1 { type uint32; }
      }
    }
  }

  module bar {
    namespace "http://example.com/ns/bar";
    prefix "b";
    import foo { prefix f; }
    revision 2015-06-07;

    augment /f:A/f:B {
      leaf counter2 { type uint32; }
    }
  }
```

This set of 3 YANG modules containing a total of 7 objects produces the following object list. Note that actual hash values are not shown, since these modules do not actually cause the YANG Hash clashes described in the examples.

```
      Object    Path                    Hash

   foo:

      container /foo:A                  h1
      list      /foo:A/B                h2
      leaf      /foo:A/B/name           h3
      leaf      /foo:A/B/counter1       h4

   bar:

     leaf      /foo:A/B/bar1:counter2   h5
```

5.5.1. Multiple Modules

In this example, assume that the 'B' and 'counter2' objects produce the same hash value, so 'h2' and 'h5' both have the same value (e.g. '1234'):

The client might retrieve an entry from the list "/foo:A/B", which would cause this subtree to be returned. Instead, the server will return a message with the resource type "core.mg.yang-hash", representing the "yang-hash" data structure. Only the entry for the requested identifier is returned, even if multiple 'rehash' list entries exist.

```
REQ: GET example.com/mg/h2?keys="entry2"

RES: 4.00 "Bad Request" (Content-Format: application/cbor)
{
   "ietf-yang-hash:yang-hash" : {
     "rehash" : [
        {
          "hash" : 1234,
           "object" : [
             {
               "module" : "foo",
               "newhash" : 5678
             },
             {
               "module" : "bar",
               "newhash" : 8182
             }
           ]
        }
     ]
  }
}
```

5.5.2. Same Module

In this example, assume that the 'B', 'counter1', and 'counter2' objects produce the same hash value, so 'h2', 'h4', and 'h5' objects all have the same value (e.g. '1234'):

The client might retrieve an entry from the list "/foo:A/B", which would cause this subtree to be returned. Instead, the server will return a message with the resource type "core.mg.yang-hash", representing the "yang-hash" data structure. Only the entry for the requested identifier is returned, even if multiple 'rehash' list entries exist.

```
REQ: GET example.com/mg/h2?keys="entry2"

RES: 4.00 "Bad Request" (Content-Format: application/cbor)
{
   "ietf-yang-hash:yang-hash" : {
     "rehash" : [
        {
          "hash" : 1234,
           "object" : [
             {
               "module" : "foo",
               "newhash" : 5678,
               "path" : "/foo:A/B"

             },
             {
               "module" : "foo",
               "newhash" : 2134,
               "path" : "/foo:A/B/counter1"
             },
             {
               "module" : "bar",
               "newhash" : 8182,
               "path" : "/foo:A/B/bar:counter2"
             }
           ]
        }
     ]
  }
}
```

5.6. Retrieval of Rehashed Data

In this example, assume that the 'B', 'counter1', and 'counter2' objects produce the same hash value, so 'h2', 'h4', and 'h5' objects all have the same value (e.g. '1234'):

The client might retrieve the top-level container "/foo:A", which would cause this subtree to be returned. Since the identifier (h1) has not been re-hashed, the server will return the requested data. The new hashes for 'h2', 'h4',and 'h5' will be returned, except the rehash bit will be set for these identifiers.

The notation "R+" indicates that the rehash bit is set.

```
REQ: GET example.com/mg/h1

RES: 2.05 Content (Content-Format: application/cbor)
{
     h1 : {
        R+5678 : {
           { h3 : "entry1"}:
             {R+2134: 615,
              R+8182: 7},
           { h3 : "entry2"}:
             {R+2134: 491,
              R+8182: 26}
         }
      }
}
```

The client will notice that the rehash bit is set for 3 nodes. The client will need to retrieve the full "yang-hash" container at this point, if that has not already been done. The rehashed identifiers will be in "rehash" list, contained in the "newhash" leaf for the "object" list.

5.7. YANG Hash in URL

When a URL contains a YANG hash, it is encoded using base64url "URL and Filename safe" encoding as specified in [RFC4648].

The hash H is represented as a 30-bit integer, divided into five 6-bit integers as follows:

```
B1 = (H & 0x3f000000) >> 24
B2 = (H & 0xfc0000) >> 18
B3 = (H & 0x03f000) >> 12
B4 = (H & 0x000fc0) >> 6
B5 =  H & 0x00003f
```

Subsequently, each 6-bit integer Bx is translated into a character Cx using Table 2 from [RFC4648], and a string is formed by concatenating the characters in the order C1, C2, C3, C4, C5.

For example, the YANG hash 0x29abdcca is encoded as "pq9zK".

6. Mapping YANG to CBOR

6.1. High level encoding

When encoding YANG variables in CBOR, the CBOR encodings entry is a map, composed of (key, value) pairs. The key is the YANG hash of entry variable, whereas the value contains its value.

For encoding of the variable values, a CBOR datatype is used. Section 6.2 provides the mapping between YANG datatypes and CBOR datatypes.

6.2. Conversion from YANG datatypes to CBOR datatypes

Table 1 defines the mapping between YANG datatypes and CBOR datatypes.

Elements of types not in this table, and of which the type cannot be inferred from a type in this table, are ignored in the CBOR encoding by default. Examples include the "description" and "key" elements. However, conversion rules for some elements to CBOR MAY be defined elsewhere.

YANG type	CBOR type	Specification
int8, int16, int32, int64, uint16, uint32, uint64, decimal64	unsigned int (major type 0) or negative int (mayor type 1)	The CBOR integer type depends on the sign of the actual value.
boolean	either "true" (major type 7, simple value 21) or "false" (major type 7, simple value 20)	
string	text string (major type 3)	
enumeration	unsigned int (major type 0)	
bits	array of text	Each text string contains the

	strings	name of a bit value that is set.
binary	byte string (major type 2)	
empty	null (major type 7, simple value 22)	TBD: This MAY not be applicable to true MIBs, as SNMP may not support empty variables...
union		Similar to the JSON transcription from [I-D.ietf-netmod-yang-json], the elements in a union MUST be determined using the procedure specified in section 9.12 of [RFC6020].
leaf-list	array (major type 4)	The array is encapsulated in the map associated with the YANG variable.
list	map (major type 5) of maps (major type 5)	Map of array element pairs (index, value). Each array element contains a map of associated YANG hash - value pairs.
container	map (major type 5)	The map contains YANG hash - value pairs corresponding to the elements in the container.
smiv2:oid	array of integers	Each integer contains an element of the OID, the first integer in the array corresponds to the most left element in the OID.

Table 1: Conversion of YANG datatypes to CBOR

7. Error Handling

In case a request is received which cannot be processed properly, the managed entity MUST return an error message. This error message MUST contain a CoAP 4.xx or 5.xx response code, and SHOULD include additional information in the payload.

Such an error message payload is encoded in CBOR, using the following structure:

TODO: Adapt RESTCONF <errors> data structure for use in CoMI. Need to select the most important fields like <error-path>.

```
errorMsg     : ErrorMsg;

*ErrorMsg {
  errorCode  : uint;
  ?errorText : tstr;
}
```

The variable "errorCode" has one of the values from the table below, and the OPTIONAL "errorText" field contains a human readable explanation of the error.

CoMI Error Code	CoAP Error Code	Description
0	4.00	General error
1	4.00	Malformed CBOR data
2	4.00	Incorrect CBOR datatype
3	4.00	Unknown MIB variable
4	4.00	Unknown conversion table
5	4.05	Attempt to write read-only variable
0..2	5.01	Access exceptions
0..18	5.00	SMI error status

The CoAP error code 5.01 is associated with the exceptions defined in [RFC3416] and CoAP error code 5.00 is associated with the error-status defined in [RFC3416].

8. Security Considerations

For secure network management, it is important to restrict access to MIB variables only to authorized parties. This requires integrity protection of both requests and responses, and depending on the application encryption.

CoMI re-uses the security mechanisms already available to CoAP as much as possible. This includes DTLS [RFC6347] for protected access to resources, as well suitable authentication and authorization mechanisms.

Among the security decisions that need to be made are selecting security modes and encryption mechanisms (see [RFC7252]). This requires a trade-off, as the NoKey mode gives no protection at all, but is easy to implement, whereas the X.509 mode is quite secure, but may be too complex for constrained devices.

In addition, mechanisms for authentication and authorization may need to be selected.

CoMI avoids defining new security mechanisms as much as possible. However some adaptations may still be required, to cater for CoMI's specific requirements.

9. IANA Considerations

'rt="core.mg.data"' needs registration with IANA.

'rt="core.mg.moduri"' needs registration with IANA.

'rt="core.mg.modset"' needs registration with IANA.

'rt="core.mg.yang-hash"' needs registration with IANA.

'rt="core.mg.yang-stream"' needs registration with IANA.

Content types to be registered:

- application/comi+cbor

10. Acknowledgements

We are very grateful to Bert Greevenbosch who was one of the original authors of the CoMI specification and specified CBOR encoding and use of hashes. Mehmet Ersue and Bert Wijnen explained the encoding aspects of PDUs transported under SNMP. Carsten Bormann has given feedback on the use of CBOR. The draft has benefited from comments

(alphabetical order) by Somaraju Abhinav, Rodney Cummings, Dee Denteneer, Esko Dijk, Michael van Hartskamp, Alexander Pelov, Zach Shelby, Hannes Tschofenig, Michel Veillette, Michael Verschoor, and Thomas Watteyne. The CBOR encoding borrows extensively from Ladislav Lhotka's description on conversion from YANG to JSON.

This material is based upon work supported by Philips Research, Huawei, and The Space & Terrestrial Communications Directorate (S&TCD); the latter under Contract No. W15P7T-13-C-A616. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Philips Research, Huawei, or The Space & Terrestrial Communications Directorate (S&TCD).

Juergen Schoenwaelder and Anuj Sehgal were partly funded by Flamingo, a Network of Excellence project (ICT-318488) supported by the European Commission under its Seventh Framework Programme.

11. Changelog

Changes from version 00 to version 01

- Focus on MIB only
- Introduced CBOR, JSON, removed BER
- defined mappings from SMI to xx
- Introduced the concept of addressable table rows

Changes from version 01 to version 02

- Focus on CBOR, used JSON for examples, removed XML and EXI
- added uri-query attributes mod and con to specify modules and contexts
- Definition of CBOR string conversion tables for data reduction
- use of Block for multiple fragments
- Error returns generalized
- SMI - YANG - CBOR conversion

Changes from version 02 to version 03

- Added security considerations

Changes from version 03 to version 04

- Added design considerations section
- Extended comparison of management protocols in introduction
- Added automatic generation of CBOR tables
- Moved lowpan table to Appendix

Changes from version 04 to version 05

- Merged SNMP access with RESTCONF access to management objects in small devices
- Added CoMI architecture section
- Added RESTCONf NETMOD description
- Rewrote section 5 with YANG examples
- Added server and payload size appendix
- Removed Appendix C for now. It will be replaced with a YANG example.

Changes from version 04 to version 05

- Extended examples with hash representation
- Added keys query parameter text
- Added select query parameter text
- Better separation between specification and instance
- Section on discovery updated
- Text on rehashing introduced
- Elaborated SMI MIB example
- Yang library use described
- use of BigEndian/LittleEndian in Hash generation specified

Changes from version 05 to version 06

- Hash values in payload as hexadecimal and in URL in base64 numbers
- Streamlined CoMI architecture text
- Added select query parameter text
- Data editing optional
- Text on Notify added
- Text on rehashing improved with example

Changes from version 06 to version 07

- reduced payload size by removing JSON hierarchy
- changed rehash handling to support small clients
- added LWM2M comparison
- Notification handling as specified in YANG
- Added Patch function
- Rehashing completely reviewed
- Discover type of YANG name encoding
- Added new resource types
- Read-only servers introduced
- Multiple updates explained

Changes from version 07 to version 08

- Changed YANG Hash algorithm to use module name instead of prefix
- Added rehash bit to allow return values to identify rehashed nodes in the response
- Removed /mg/mod.set resource since this is not needed
- Clarified that YANG Hash is done even for unimplemented objects
- YANG lists transported as CBOR maps of maps
- Adapted examples with more CBOR explanation

- Added CBOR code examples in new appendix
- Possibility to use other than default stream
- Added text and examples for Patch payload
- Repaired some examples
- Added appendices on hash clash probability and hash clash storage overhead

12. References

12.1. Normative References

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

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <http://www.rfc-editor.org/info/rfc4648>.

[RFC5277] Chisholm, S. and H. Trevino, "NETCONF Event Notifications", RFC 5277, DOI 10.17487/RFC5277, July 2008, <http://www.rfc-editor.org/info/rfc5277>.

[RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, DOI 10.17487/RFC6020, October 2010, <http://www.rfc-editor.org/info/rfc6020>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <http://www.rfc-editor.org/info/rfc7049>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://www.rfc-editor.org/info/rfc7252>.

[RFC7396] Hoffman, P. and J. Snell, "JSON Merge Patch", RFC 7396, DOI 10.17487/RFC7396, October 2014, <http://www.rfc-editor.org/info/rfc7396>.

[I-D.becker-core-coap-sms-gprs]
Becker, M., Li, K., Kuladinithi, K., and T. Poetsch, "Transport of CoAP over SMS", draft-becker-core-coap-sms-gprs-05 (work in progress), August 2014.

[I-D.ietf-core-block]
Bormann, C. and Z. Shelby, "Block-wise transfers in CoAP", draft-ietf-core-block-18 (work in progress), September 2015.

[I-D.ietf-core-observe]
Hartke, K., "Observing Resources in CoAP", draft-ietf-core-observe-16 (work in progress), December 2014.

[I-D.ietf-netmod-yang-json]
Lhotka, L., "JSON Encoding of Data Modeled with YANG", draft-ietf-netmod-yang-json-06 (work in progress), October 2015.

[I-D.ietf-netconf-restconf]
Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", draft-ietf-netconf-restconf-07 (work in progress), July 2015.

[I-D.vanderstok-core-patch]
Stok, P. and A. Sehgal, "Patch Method for Constrained Application Protocol (CoAP)", draft-vanderstok-core-patch-02 (work in progress), October 2015.

[murmur3] "murmurhash family", Web http://en.wikipedia.org/wiki/MurmurHash.

[murmur-imp]
"murmurhash implementation", Web https://code.google.com/p/smhasher/.

12.2. Informative References

[RFC2578] McCloghrie, K., Ed., Perkins, D., Ed., and J. Schoenwaelder, Ed., "Structure of Management Information Version 2 (SMIv2)", STD 58, RFC 2578, DOI 10.17487/RFC2578, April 1999, <http://www.rfc-editor.org/info/rfc2578>.

[RFC3410] Case, J., Mundy, R., Partain, D., and B. Stewart, "Introduction and Applicability Statements for Internet-Standard Management Framework", RFC 3410, DOI 10.17487/RFC3410, December 2002, <http://www.rfc-editor.org/info/rfc3410>.

[RFC3411] Harrington, D., Presuhn, R., and B. Wijnen, "An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks", STD 62, RFC 3411, DOI 10.17487/RFC3411, December 2002, <http://www.rfc-editor.org/info/rfc3411>.

[RFC3414] Blumenthal, U. and B. Wijnen, "User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)", STD 62, RFC 3414, DOI 10.17487/RFC3414, December 2002, <http://www.rfc-editor.org/info/rfc3414>.

[RFC3416] Presuhn, R., Ed., "Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP)", STD 62, RFC 3416, DOI 10.17487/RFC3416, December 2002, <http://www.rfc-editor.org/info/rfc3416>.

[RFC3418] Presuhn, R., Ed., "Management Information Base (MIB) for the Simple Network Management Protocol (SNMP)", STD 62, RFC 3418, DOI 10.17487/RFC3418, December 2002, <http://www.rfc-editor.org/info/rfc3418>.

[RFC4293] Routhier, S., Ed., "Management Information Base for the Internet Protocol (IP)", RFC 4293, DOI 10.17487/RFC4293, April 2006, <http://www.rfc-editor.org/info/rfc4293>.

[RFC4944] Montenegro, G., Kushalnagar, N., Hui, J., and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", RFC 4944, DOI 10.17487/RFC4944, September 2007, <http://www.rfc-editor.org/info/rfc4944>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <http://www.rfc-editor.org/info/rfc6241>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[RFC6643] Schoenwaelder, J., "Translation of Structure of Management Information Version 2 (SMIv2) MIB Modules to YANG Modules", RFC 6643, DOI 10.17487/RFC6643, July 2012, <http://www.rfc-editor.org/info/rfc6643>.

[RFC6650] Falk, J. and M. Kucherawy, Ed., "Creation and Use of Email Feedback Reports: An Applicability Statement for the Abuse Reporting Format (ARF)", RFC 6650, DOI 10.17487/RFC6650, June 2012, <http://www.rfc-editor.org/info/rfc6650>.

[RFC6690] Shelby, Z., "Constrained RESTful Environments (CoRE) Link Format", RFC 6690, DOI 10.17487/RFC6690, August 2012, <http://www.rfc-editor.org/info/rfc6690>.

[RFC6775] Shelby, Z., Ed., Chakrabarti, S., Nordmark, E., and C. Bormann, "Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)", RFC 6775, DOI 10.17487/RFC6775, November 2012, <http://www.rfc-editor.org/info/rfc6775>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <http://www.rfc-editor.org/info/rfc7228>.

[RFC7277] Bjorklund, M., "A YANG Data Model for IP Management", RFC 7277, DOI 10.17487/RFC7277, June 2014, <http://www.rfc-editor.org/info/rfc7277>.

[RFC7317] Bierman, A. and M. Bjorklund, "A YANG Data Model for System Management", RFC 7317, DOI 10.17487/RFC7317, August 2014, <http://www.rfc-editor.org/info/rfc7317>.

[I-D.ietf-core-interfaces] Shelby, Z., Vial, M., and M. Koster, "CoRE Interfaces", draft-ietf-core-interfaces-03 (work in progress), July 2015.

[I-D.ersue-constrained-mgmt] Ersue, M., Romascanu, D., and J. Schoenwaelder, "Management of Networks with Constrained Devices: Problem Statement, Use Cases and Requirements", draft-ersue-constrained-mgmt-03 (work in progress), February 2013.

[I-D.ietf-lwig-coap] Kovatsch, M., Bergmann, O., and C. Bormann, "CoAP Implementation Guidance", draft-ietf-lwig-coap-03 (work in progress), July 2015.

[XML] "Extensible Markup Language (XML)", Web http://www.w3.org/xml.

[OMA] "OMA-TS-LightweightM2M-V1_0-20131210-C", Web http://technical.openmobilealliance.org/Technical/current_releases.aspx.

[DTLS-size] Hummen, R., Shafagh, H., Raza, S., Voigt, T., and K. Wehrle, "Delegation-based Authentication and Authorization for the IP-based Internet of Things", Web http://www.vs.inf.ethz.ch/publ/papers/mshafagh_secon14.pdf.

[mibreg] "Structure of Management Information (SMI) Numbers (MIB Module Registrations)", Web http://www.iana.org/assignments/smi-numbers/smi-numbers.xhtml/.

[dcaf] Bormann, C., Bergmann, O., and S. Gerdes, "Delegated Authenticated Authorization for Constrained Environments", Private Information .

[openwsn] Watteijne, T., "Coap size in Openwsn", Web http://builder.openwsn.org/.

[coll-prob] Preshing, j., "Hash collision probablilities", Web http://preshing.com/20110504/hash-collision-probabilities, May 2011.

[birthday] Wikipedia, , "Birthday problem", Web https://en.wikipedia.org/wiki/Birthday_problem.

[Erbium] Kovatsch, M., "Erbium Memory footprint for coap-18", Private Communication .

[management] Schoenwalder, J. and A. Sehgal, "Management of the Internet of Things", Web http://cnds.eecs.jacobs-university.de/slides/2013-im-iot-management.pdf, 2013.

[I-D.ietf-netconf-yang-patch] Bierman, A., Bjorklund, M., and K. Watsen, "YANG Patch Media Type", draft-ietf-netconf-yang-patch-04 (work in progress), June 2015.

Appendix A. Payload and Server sizes

This section provides information on code sizes and payload sizes for a set of management servers. Approximate code sizes are:

Code	processor	Text	Data	reference
Observe agent	erbium	800	n/a	[Erbium]
CoAP server	MSP430	1K	6	[openwsn]
SNMP server	ATmega128	9K	700	[management]
Secure SNMP	ATmega128	30K	1.5K	[management]
DTLS server	ATmega128	37K	2K	[management]
NETCONF	ATmega128	23K	627	[management]
JSON parser	CC2538	4.6K	8	[dcaf]
CBOR parser	CC2538	1.5K	2.6K	[dcaf]
DTLS server	ARM7	15K	4	[I-D.ietf-lwig-coap]
DTLS server	MSP430	15K	4	[DTLS-size]
Certificate	MSP430	23K		[DTLS-size]
Crypto	MSP430	2-8K		[DTLS-size]

Thomas says that the size of the CoAP server is rather arbitrary, as its size depends mostly on the implementation of the underlying library modules and interfaces.

Payload sizes are compared for the following request payloads, where each attribute value is null (N.B. these sizes are educated guesses, will be replaced with generated data). The identifier are assumed to be a string representation of the OID. Sizes for SysUpTime differ due to preambles of payload. "CBOR opt" stands for CBOR payload where the strings are replaced by table numbers.

Request	BERR SNMP	JSON	CBOR	CBOR opt
IPnetTOMediaTable	205	327	~327	~51
lowpanIfStatsTable		710	614	121
sysUpTime	29	13	~13	20
RESTCONF example				

Appendix B. Notational Convention for CBOR data

To express CBOR structures [RFC7049], this document uses the following conventions:

A declaration of a CBOR variable has the form:

```
name : datatype;
```

where "name" is the name of the variable, and "datatype" its CBOR datatype.

The name of the variable has no encoding in the CBOR data.

"datatype" can be a CBOR primitive such as:

tstr: A text string (major type 3)

uint: An unsigned integer (major type 0)

map(x,y): A map (major type 5), where each first element of a pair is of datatype x, and each second element of datatype y. A '.' character for either x or y means that all datatypes for that element are valid.

A datatype can also be a CBOR structure, in which case the variable's "datatype" field contains the name of the CBOR structure. Such CBOR structure is defined by a character sequence consisting of first its name, then a '{' character, then its subfields and finally a '}' character.

A CBOR structure can be encapsulated in an array, in which case its name in its definition is preceded by a '*' character. Otherwise the structure is just a grouping of fields, but without actual encoding of such grouping.

The name of an optional field is preceded by a '?' character. This means, that the field may be omitted if not required.

Appendix C. CBOR examples

TODO: inconsistent with examples in text

The two examples in this appendix show the use of the CBOR map. In each example, the JSON code of the text is shown followed by the corresponding CBOR code that MUST be transported.

The first example show the transport of two leaves of a simple container

```
--------------------------JSON example ----------------------
{
  0x021ca491 : {
      0x047c468b : "2014-10-26T12:16:51Z",
      0x1fb5f4f8 : "2014-10-21T03:00:00Z"
   }
}
-----------------------------CBOR code ----------------------
a1                                      # map(1)
   1a 021ca491                          # unsigned(35431569)
   a2                                   # map(2)
      1a 047c468b                       # unsigned(75253387)
      74                                # text(20)
         323031342d31302d32365431323a31363a35315a # "2014-10-26T12:16:51Z"
      1a 1fb5f4f8                       # unsigned(532018424)
      74                                # text(20)
         323031342d31302d32315430333a30303a30305a # "2014-10-21T03:00:00Z"
```

The following figure shows the transport of two YANG list items. A map of length 2 (the 2 list items) is composed of two maps: the key of length 3 and the value of length 5.

```
   --------------------------JSON example ----------------------
   {
       0x06aaddbc: {
           {
             0x346b3071 : 1,
             0x3650bb64 : "ipv4",
             0x06fd4d91 : "10.0.0.51"}:
           {
             0x26180bcb : "00:00:10:01:23:45",
             0x3d6bbe90 : "2333943",
             0x35ecbb3d : "static",
             0x13038bb5 : "reachable",
```

```
            0x09e1fa37 : "active"
          },
          {
            0x346b3071 : 1,
            0x3650bb64 : "ipv4",
            0x06fd4d91 : "9.2.3.4"}:
          {
            0x26180bcb : "00:00:10:54:32:10",
            0x3d6bbe90 : "2329836",
            0x35ecbb3d : "dynamic",
            0x13038bb5 : "unknown",
            0x09e1fa37 : "active"
          }
       }
   }
   ----------------------------CBOR code ----------------------
   a1                                    # map(1)
      1a 06aaddbc                        # unsigned(111861180)
      a2                                 # map(2)
         a3                              # map(3)
            1a 346b3071                  # unsigned(879439985)
            01                           # unsigned(1)
            1a 3650bb64                  # unsigned(911260516)
            64                           # text(4)
               69707634                  # "ipv4"
            1a 06fd4d91                  # unsigned(117263761)
            69                           # text(9)
               31302e302e302e3531        # "10.0.0.51"
         a5                              # map(5)
            1a 26180bcb                  # unsigned(639110091)
            71                           # text(17)
               30303a30303a31303a30313a32333a3435 # "00:00:10:01:23:45"
            1a 3d6bbe90                  # unsigned(1030471312)
            67                           # text(7)
               32333333393433            # "2333943"
            1a 35ecbb3d                  # unsigned(904706877)
            66                           # text(6)
               737461746963              # "static"
            1a 13038bb5                  # unsigned(318999477)
            69                           # text(9)
               726561636861626c65        # "reachable"
            1a 09e1fa37                  # unsigned(165804599)
            66                           # text(6)
               616374697665              # "active"
         a3                              # map(3)
            1a 346b3071                  # unsigned(879439985)
            01                           # unsigned(1)
            1a 3650bb64                  # unsigned(911260516)
```

```
            64                              # text(4)
               69707634                     # "ipv4"
            1a 06fd4d91                     # unsigned(117263761)
            67                              # text(7)
               392e322e332e34               # "9.2.3.4"
         a5                                 # map(5)
            1a 26180bcb                     # unsigned(639110091)
            71                              # text(17)
               30303a30303a31303a35343a33323a3130 # "00:00:10:54:32:10"
            1a 3d6bbe90                     # unsigned(1030471312)
            67                              # text(7)
               32333239383336               # "2329836"
            1a 35ecbb3d                     # unsigned(904706877)
            67                              # text(7)
               64796e616d6963               # "dynamic"
            1a 13038bb5                     # unsigned(318999477)
            67                              # text(7)
               756e6b6e6f776e               # "unknown"
            1a 09e1fa37                     # unsigned(165804599)
            66                              # text(6)
               616374697665                 # "active"
```

The following figure shows the example Patch payload:

```
--------------------------JSON example ----------------------
 {
     {
      0x200a297b : "author1",
      0x35e1c1da : "book2"}:
      { null : null},
     {
      0x200a297b : "author5"} :
       {0x26c3ca7e : 4444},
     {
      0x200a297b : "newauthor",
      0x35e1c1da : "newbook"}:
     { 0x0119ddec : 1,
       0x26c3ca7e : 1},

  0x5cc7979 : "favoured",
  0x2935c912 : null,
  0x3f83c748 : [ "example"],
  0x324d9526 : "+01-123-456-7890"
}
----------------------------CBOR code ----------------------
a7                                      # map(7)
   a2                                   # map(2)
      1a 200a297b                       # unsigned(537536891)
```

```
      67                                  # text(7)
         617574686f7231                   # "author1"
      1a 35e1c1da                         # unsigned(903987674)
      65                                  # text(5)
         626f6f6b32                       # "book2"
   a1                                     # map(1)
      00                                  # unsigned(0)
      f6                                  # primitive(22)
   a1                                     # map(1)
      1a 200a297b                         # unsigned(537536891)
      67                                  # text(7)
         617574686f7235                   # "author5"
   a1                                     # map(1)
      1a 26c3ca7e                         # unsigned(650365566)
      19 115c                             # unsigned(4444)
   a2                                     # map(2)
      1a 200a297b                         # unsigned(537536891)
      69                                  # text(9)
         6e6577617574686f72               # "newauthor"
      1a 35e1c1da                         # unsigned(903987674)
      67                                  # text(7)
         6e6577626f6f6b                   # "newbook"
   a2                                     # map(2)
      1a 0119ddec                         # unsigned(18472428)
      01                                  # unsigned(1)
      1a 26c3ca7e                         # unsigned(650365566)
      01                                  # unsigned(1)
   1a 05cc7979                            # unsigned(97286521)
   68                                     # text(8)
      6661766f75726564                    # "favoured"
   1a 2935c912                            # unsigned(691390738)
   f6                                     # primitive(22)
   1a 3f83c748                            # unsigned(1065600840)
   81                                     # array(1)
      67                                  # text(7)
         6578616d706c65                   # "example"
   1a 324d9526                            # unsigned(843945254)
   70                                     # text(16)
      2b30312d3132332d3435362d37383930    # "+01-123-456-7890"
```

Appendix D. Comparison with LWM2M

CoMI and LWM2M, both, provide RESTful device management services over CoAP. Differences between the designs are highlighted in this section.

Unlike CoMI, which enables the use of SMIv2 and YANG data models for device management, LWM2M defines a new object resource model. This

means that data models need to be redefined in order to use LWM2M. In contrast, CoMI provides access to a large variety of SMIv2 and YANG data modules that can be used immediately.

Objects and resources within CoMI are identified with a YANG hash value, however, each object is described as a link in the CoRE Link Format by LWM2M. This approach by LWM2M can lead to larger complex URIs and more importantly payloads can grow large in size. Using a hash value to represent the objects and resources allows URIs and payloads to be smaller in size, which is important for constrained devices that may not have enough resources to process large messages.

LWM2M encodes payload data in Type-length-value (TLV), JSON or plain text formats. While the TLV encoding is binary and can result in reduced message sizes, JSON and plain text are likely to result in large message sizes when lots of resources are being monitored or configured. Furthermore, CoMI's use of CBOR gives it an advantage over the LWM2M's TLV encoding as well since this too is more efficient [citation needed].

CoMI is aligned with RESTCONF for constrained devices and uses YANG data models that have objects containing resources organized in a tree-like structure. On the other hand, LWM2M uses a very flat data model that follows the "object/instance/resource" format, with no possibility to have subresources. Complex data models are, as such, harder to model with LWM2M.

In situations where resources need to be modified, CoMI uses the CoAP PATCH operation when resources are modified partially. However, LWM2M uses the CoAP PUT and POST operations, even when a subset of the resource needs modifications.

Appendix E. Hash clash probability

Number of names	28 bits	29 bits	30 bits	31 bits	32 bits	33 bits
10	1,7E-07	8,4E-08	4,2E-08	2,1E-08	1,1E-08	5,2E-09
100	1,8E-05	9,2E-06	4,6E-06	2,3E-06	1,2E-06	5,8E-07
200	7,4E-05	3,7E-05	1,9E-05	9,3E-06	4,6E-06	2,3E-06
10^3	1,9E-03	9,3E-04	4,7E-04	2,3E-04	1,2E-04	5,8E-05
4000	3,0E-02	1,5E-02	7,5E-03	3,7E-03	1,9E-03	9,3E-04
10^4	1,9E-01	9,3E-02	4,6E-02	2,3E-02	1,2E-02	5,8E-03

Table 2: Probability of one or more clashes

In CoMI the YANG node names are hashed with the murmur3 hash algorithm. The consequence is that a clash may occur, and the YANG node name needs to be rehashed with a given prefix character. This appendix calculates the probability of a hash clash as function of the hash size and the number of YANG names. The standard way to calculate the probability of a clash is to calculate the probability that no clashes occur [birthday], [coll-prob].

The probability of no clashes when generating k numbers with a hash size of N=2^bits is given by:

((N-1)/N)*((N-2)/N)*....(N-(k-1))/N

which can be approximated with:

exp(-k(k-1)/2N)

The probability that one or more clashes occur is given by:

1 - exp(-k(k-1)/2N) ~ k(k-1)/2N

Table 2 shows the probabilities for a given set of values of N=2^bits and number of YANG node names k. Probabilities which are larger than 0.5 are not shown because the used approximations are not accurate any more.

The overhead in servers and clients depends on the number of clashes. Therefore it is interesting to know the probability that more than

one clash occurs. The probability that one pair of hashes clashes out of k hashes is given by:

k*(k-1)/2N

The probability that the remaining k-2 hashes do no clash is given by:

exp(-(k-3)(k-2)/2N)

The probability that exactly one pair of hashes clashes is given by:

(k*(k-1)/2N) * exp(-(k-3)(k-2)/2N)

The probability that more than one pair clashes is given by the probability that a clash occurs minus the probability that only one pair clashes. This leads to:

k(k-1)/2N - (k*(k-1)/2N) * exp(-(k-3)(k-2)/2N) =

(k(k-1)/2N) (1-exp(-(k-3)(k-2)/2N)) =

(k(k-1)(k-3)(k-2)/4N^2)

Number of names	28 bits	29 bits	30 bits	31 bits	32 bits	33 bits
10	1,8E-14	4,4E-15	1,1E-15	2,7E-16	6,8E-17	1,7E-17
100	3,3E-10	8,3E-11	2,1E-11	5,2E-12	1,3E-12	3,3E-13
200	5,4E-09	1,4E-09	3,4E-10	8,5E-1	2,1E-11	5,3E-12
10^3	3,4E-06	8,6E-07	2,2E-07	5,4E-08	1,4E-08	3,4E-09
4000	8,9E-04	2,2E-04	5,6E-05	1,4E-05	3,5E-06	8,7E-07
10^4	3,5E-02	8,7E-03	2,2E-03	5,4E-04	1,4E-04	3,4E-05

Table 3: Probability of more than 2 entries equal clashes

The corresponding probabilities are shown in Table 3. The probabilities of Table 3 include the probability that more than 2 hashes share one clashing value. Assuming a hash size of 2^30, and about 1000 YANG nodes in a server, the probability of one clashing

pair is 0.5*10^-3, and the probability that more clashes occur is 2*10^-7.

Appendix F. Hash clash storage overhead

Clashes may occur in servers dynamically during the operation of their clients, and clashes must be handled on a per server basis in the client. When rehashing is possible, clashing names on a given server are prefixed with a character (for example "~") and are rehashed, thus leading to hash values which uniquely identify the data nodes in the server. This appendix calculates the storage space needed when a clash occurs in a set of servers running the same server code. Appendix E shows that more than one clash in a server set is exceptional, which suggests at most two clashing object names in a given server.

The sizes of server and client tables needed to handle the clashes in client and server are calculated separately, because they differ significantly.

F.1. Server tables

When a request arrives at the server, the server must relate the incoming hash value to the memory locations where the related values are stored. In the server a translation table must be provided that relates a hash value to a memory address where either the raw data or a description of the data (as prescribed by the YANG compiler) are stored. The required storage space is a sequence of (32 bit yang hash, 64 bit memory address) for every YANG data node. The translation table size in a server is 12 bytes times the number of YANG data nodes in the server.

For every clashing hash value the following server clash table entries are needed: Clashed hash value, module name, and new hash. To reduce table size in the client, module name can be replaced with a 1 byte module identifier. The module identifier represents the index value of an array of module names. Server clash table size is: 2 hashes (8 bytes) + 1 module identifier (1 byte)

F.2. Client tables

In the client, the compiled code must refer to a hash value. To cope with on-the-fly rehashing, the compiled code needs to invoke a procedure that returns the possibly rehashed value as function of the original hash value, module name, and server address. The client needs to store a client clash table containing: the clashed hash value, module name, server IPv6 address (or name), and rehash value for as many rehashes occurring in a given server. Many servers

contain an identical set of YANG modules. The servers containing the same module set belong to the same server type. The server type is used to administrate the hash clash occurrence. To reduce client clash table size, module name can be replaced with a 1 byte module identifier. The module identifier represents the index value of an array of module names. A table of IPv6 server addresses must already exist in the client. To reduce client clash table size further, the server IPv6 address can be replaced with a 1 byte server type identifier. The server table can be ordered according to server type. A table with server type and pointer to sub-table start suffices to find all IPv6 addresses belonging to a server type.

The client clash table reduces to clashed hash value (4 bytes), module identifier (1 byte), server type identifier (1 byte) and rehash value (4 bytes).

F.3. Table summary

Sizes of all the tables are:

Server clash table: 9 bytes per clashing object name.

Client clash table: 10 bytes per server type, per clashing object name.

Array of module names: Sum of module name sizes.

Server identifier table: 1 byte server type + 4 bytes pointer per server type.

The existence of the translation table in a server is required independent of rehashing. The table sizes calculated to estimate the storage requirements coming from CoMI clashes. Assume the following numbers:

- 500 data nodes per server
- 10 server types
- 30 modules
- Module name is on average 20 bytes
- Maximum of 2 clashing object names occurring in 2 server types

This yields the following overhead estimates:

Server tables size:

* Server clash table: 2*9 bytes represents 18 bytes

* Module name array: 30*20 represents 600 bytes

Client table sizes:

* Client clash table: 10*2*2 represents 40 bytes for 2 object names in 2 server types.

* Module name array: 30*20 represents 600 bytes.

* Server identifier table: 10*5 = 50 bytes

In conclusion:

1. Storage space size in client is independent of number of servers but depends on number of server types.

2. There is a common storage size for the module array of 600 bytes.

3. Assuming 2 clashing object names in 2 server types, additional storage space in client is 40 bytes and in server 18 bytes.

4. When the module array is suppressed (removing 600 bytes storage space), the server clash table and the client clash table increase with 40 bytes and 80 bytes respectively.

Authors' Addresses

Peter van der Stok
consultant

Phone: +31-492474673 (Netherlands), +33-966015248 (France)
Email: consultancy@vanderstok.org
URI: www.vanderstok.org

Andy Bierman
YumaWorks
685 Cochran St.
Suite #160
Simi Valley, CA 93065
USA

Email: andy@yumaworks.com

Juergen Schoenwaelder
Jacobs University
Campus Ring 1
Bremen 28759
Germany

Email: j.schoenwaelder@jacobs-university.de

Anuj Sehgal
consultant
Campus Ring 1
Bremen 28759
Germany

Email: anuj@iurs.org