The importance of the collection pattern for OneM2M architecture

Mahdi Ben Alaya
ben.alaya@laas.fr

Thierry Monteil
monteil@laas.fr

04/09/2014

Outline

- Introduction
- Collection is a pattern for RESTful architecture
- Collection enhances system performance
- Collection solves the resource type technical problem
- The Collection resource structure
- Impact on ARC
- Impact on PRO
- Conclusion

Introduction

- Collection is a pattern for RESTful architecture.

- Clarifications about collections:
  - Collections URIs are short and non hierarchal.
  - Collections resources do not cause a rigidness to the tree structure.
  - Collections do not overload the server.

- Collection provides essential features with a direct impact on the OneM2M architecture design, implementation, and performance.
  1. Design:
     - Collection simplifies the resource structure and make it easy to consume
     - Collection URIs are required for POST request to be compliant with REST/HATEOAS constraints.
  2. Performance:
     - Collection decreases resource representation size and response time.
     - Collection removes redundant resource type tags
     - Collection size is controlled by paging links.
  3. Implementation:
     - Collections provides separate URIs for each resource type, thus the server knows the resource type just by looking to the request URI.

Design (1/4)
Collection is a pattern for RESTful architecture

- REST/HATEOAS constraint decouples client and server in a way that allows the server functionality to evolve independently.

- Hypermedia and links should be used to find your way through the API. (Follow-Your-Nose strategy). Servers must instruct clients on how to construct appropriate URIs without using out-of-band information.

- Leonard Richardson, inventor of HATEOAS, and author of “RESTful Web APIs” book dedicated the chapter 6 of his book to the collection pattern: Collection is a well known pattern for RESTful API.

- Example of existing HATEOAS media type supporting the collection pattern: Collection+JSON, AtomPub, HAL, JSON-LD, Siren

Design(2/4)
Flexible resource tree with short and flat URIs

- **Example 1**: The client GET a Container resource starting from the CSEBase URI making use of Collections pattern.
- Resource representation are small so it is easy for the client to use provided links to change the application state.
- Collection URIs are short and flat. Resource tree is flexible.
- An optional resource type prefix is used to identify resources e.g. “**/a**/a1” and “**/c**/c1”.
  - We have a unique resource name per each resource type.
  - For All requests, the server knows the resource type just by looking to the URI.

N	Request	Response
1)	**GET /csebase**	<cse> … **<link rel=“applications" href=“/a" />** **<link rel=“containers" href=“/c" />** **<link rel=“subscriptions" href=“/s” />** </cse>
2)	GET /a	<collection> … <items rel=“application”> <link href=“/a/a1" /> <link href=“/a/a2" /> <link href=“/a/a3" /> <items> </collection>
3)	GET /a/a1	<application> … <link rel=“containers" href=“/a/a1/c" /> <link rel=“groups" href=“/a/a1/g" /> <link rel=“subscriptions" href=“/a/a1/s" /> </application>
4)	**GET /a/a1/c**	<collection> … <items rel=“container”> **<link href=“/c/c1" />** **<link href=“/c/c2" />** **<link href=“/c/c3" />** <items> </collection>
5)	**GET /c/c1**	<container> … <link rel=“instances" href=“/c/c1/i" /> <link rel=“subscriptions" href=“/c/c1/s" /> </container>

Design(3/4)
Collection URIs are required for POST request

- **Example 2**: The client POST a ContentInstance resource on the retrieved container making use of Collections pattern.
- The server use collections URIs to instruct the client on how to CREATE a new resource.
- The server return back a flat and short URI for the new application.

N	Request	Response
1)	GET /c/c1	<container> … **<link rel="instances" href="/c/c1/i"/>** <link rel="subscriptions" href="/c/c1/s"/> </container>
2)	**POST /c/c1/i** <instance> <content>xxx </content> </instance>	**Location: i/i5** <instance> <content>xxx</content> <link rel="subscriptions" href="/i/5/s"/> … </instance>
3)	**GET /i/i5**	<instance> <content>xxx</content> <link rel="subscriptions" href="/i/5/s"/> … </instance>

Design(4/4)
Flat URIs without resource type prefixes

- No prefix is used to identify resources e.g. “/a1” and “/c1”.
  - We must have a global unique resource name per CSE.
  - For GET, PUT, and DELETE requests, the server must check the resource ID before to find the resource type.
  - For POST requests, the server uses the Collection URI to find the resource type.

Example 1

N	Request	Response
1)	**GET /csebase**	<cse> … <link rel=“applications" **href=“/a"** /> <link rel=“containers" **href=“/c"** /> <link rel=“subscriptions" **href=“/s”** /> </cse>
2)	**GET /a**	<collection> … <items rel=“application”> <link **href=“/a1"** /> <link **href=“/a2"** /> <link **href=“/a3"** /> <items> </collection>
3)	**GET /a1**	<application> … <link rel=“containers" **href=“/a1/c**" /> <link rel=“groups" **href=“/a1/g"** /> <link rel=“subscriptions" **href=“/a1/s"** /> </application>
4)	**GET /a1/c**	<collection> … <items rel=“container”> <link **href=“/c1**" /> <link **href=“/c2"** /> <link **href=“/c3"** /> <items> </collection>
5)	**GET /c1**	<container> … <link rel=“instances" **href=“/c1/i"** /> <link rel=“subscriptions" **href=“/c1/s"** /> </container>

Example 2

N	Request	Response
1)	**GET /c1**	<container> … <link rel=“instances" **href=“/c1/i"**/> <link rel=“subscriptions" **href=“/c1/s"**/> </container>
2)	**POST /c1/i** <instance> <content>xxx </content> </instance>	**Location: /i5** <instance> <content>xxx</content> <link rel=“subscriptions" **href=“/i5/s"**/> … </instance>
3)	**GET /i5**	<instance> <content>xxx</content> <link rel=“subscriptions" **href=“/i5/s"**/> … </instance>

Performance (1/3)
Collections reduce the representation size

- In the current architecture, the resource representation contains big number of links to all child resources which considerably increases the size of the payload.

- Collections organize resources of the same type in separate representations. The resource representation become small because it contains only few collections links instead of big number of mixed resource links.

Without Collection	With Collection
<cse> … <link rel=“**application**" href=“/a1" /> <link rel=“**application**" href=“/a2" /> <link rel=“**application**" href=“/a3" /> … <link rel=“**container**" href=“/c1" /> <link rel=“**container**" href=“/c2" /> … <link rel=“**subscription**" href=“/s1” /> <link rel=“**subscription**" href=“/s2” /> </cse>	<cse> …. <link rel=“**applications**" href=“/a" /> <link rel=“**containers**" href=“/c" /> <link rel=“**subscriptions**" href=“/s” /> </cse>

Performance (2/3)
Collection remove redundant resource type tags

- In the current architecture, a resource contains mixed resources types, so each resource link must contain its own type tag, with increase the payload size.

- Collections contain resources of the same type, so all redundant type tags can be factorized in one tag: the collection type tag, with decrease the payload size.

Without Collection	With Collection
<cse> … <link **rel="application"** href="/a1" /> <link **rel="application"** href="/a2" /> <link **rel="application"** href="/a3" /> <link **rel="application"** href="/a4" /> … <link **rel="application"** href="/an" /> </cse>	<collection> … <items **rel="application"**> <link href="/a/a1" /> <link href="/a/a2" /> <link href="/a/a3" /> <link href="/a/a4" /> … <link href="/a/an" /> <items> </collection>

Performance (3/3)
Collection size can be controlled with paging links

- Collections are designed to contain a big number of items. To limit collections size, it is recommenced to use collection paging.

- Technically the server returns the representation including a limited number of items, with "next" and "previous“ links. Example:

```
<collection>
    …
    <items rel=“application”>
        <link href=“/a/a5" />
        <link href=“/a/a6" />
        <link href=“/a/a7" />
        <link href=“/a/a8" />
        <link href=“/a/a9" />
    <items>
    <link rel="next" href=“/a?start=10"/>
    <link rel="previous" href=“a?start=0"/>
</collection>
```

Implementation (1/2)
Resource type technical problem

- When the server receives a resource representation, it validates it using the specific XSD schema file, if succeeded, it parses to check access right, mandatory attributes, performing the request, notification, etc.

- In the current architecture, we define the same URI to create resources of different type. In the following examples, “/CSEBase” is the same URI for the 3 requests:
  - To create a **AE**, the client send POST to /CSEBase with AE representation.
  - To create a **Container** the client send POST to /CSEBase with AE representation.
  - To create a **Goup**, the client send POST to /CSEBase with AE representation.

- The server will not be able to determine the type of the received resource. There is two possible workarounds:
  1. The server will look inside the received representation to find the resource type tag, before validating the representation. (overload)
  2. The server will try the XSD schemas one by one till it finds the correct one, if validation is ok, then it parses the representation. (overload)

- With collections, we have separate links for each resource type. The server knows the resource type just by looking to the request URI.

Implementation (2/2)
Collection URIs solve the resource type problem

- With collections, we have separate links for each resource type. Let’s consider the CSEBase representation:

```
<cse>
    …
    <link rel=“applications" href=“/a" />
    <link rel=“containers" href=“/c" />
    <link rel=“subscriptions" href=“/s” />
</cse>
```

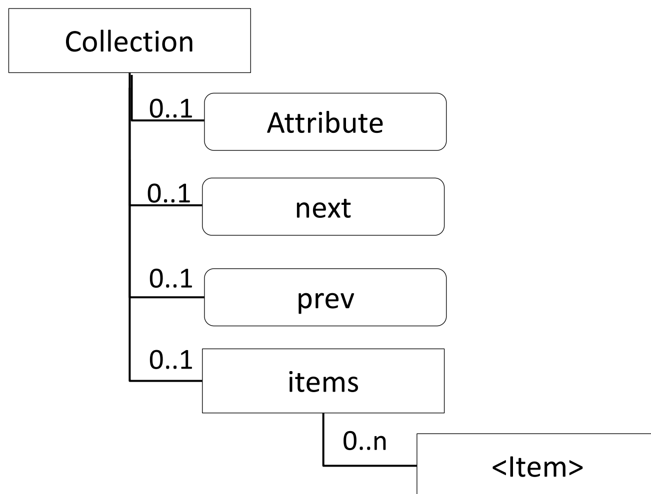
- We have now a different URI for each resource creation request:
  - To create a **AE**, the client send POST to **/a** with AE representation.
  - To create a **Container** the client send POST to **/c** with Container representation.
  - To create a **Goup**, the client send POST to **/g** with AE representation.

- The server knows the resource type just by looking to the request URI. This is the natural way to solve the resource type issue.

The Collection resource structure

- The Collection resource could have a specific structure depending on the items type. (Example ETSI M2M collections)

- The collection could also have a generic structure supporting all item types. Example:



Collection resource type structure

```
<collection>
        …
        <items rel=“resource-type”>
                <link href=“uri" />
                <link href=“uri " />
                …
         <items>
        <link rel="next" href=“uri"/>
        <link rel="previous" href=“uri"/>
 </collection>
```

Collection resource XML example

LAAS-CNRS

Impact on ARC

Child resource types must contain collections resources (e.g. AEs, Containers, etc.)

Resource Type	Short Description	Child Resource Types	Parent Resource Types	Clause
Content Instance	Represents a data instance in the container resource	*subscription*	*container*	9.6.7
AE	Stores information about the AE. It is created as a result of successful registration of an AE with the registrar CSE	*subscription, container, group, accessControlPolicy, mgmtObj, commCapabilities, pollingChannel*	*remoteCSE, CSEBase*	9.6.5
container	Shares data instances among entities. Used as a mediator that takes care of buffering the data to exchange "data" between AEs and/or CSEs.(…)	*container, contentInstance, subscription,*	*application, container, remoteCSE, CSEBase*	9.6.6
CSEBase	The structural root for all the resources that are residing on a CSE. It shall store information about the CSE itself	*remoteCSE, node, application, container, group, accessControlPolicy, subscription, mgmtObj, mgmtCmd, locationPolicy, statsConfig*	*None*	9.6.3

Table 9.6-1 Resource Summary

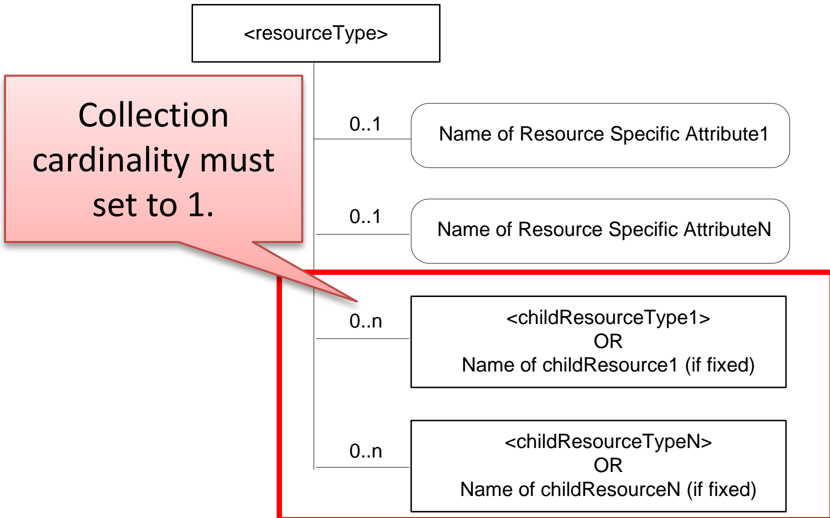


Figure 9.5-1: <resourceType> representation convention

CSEBase resource including collections example

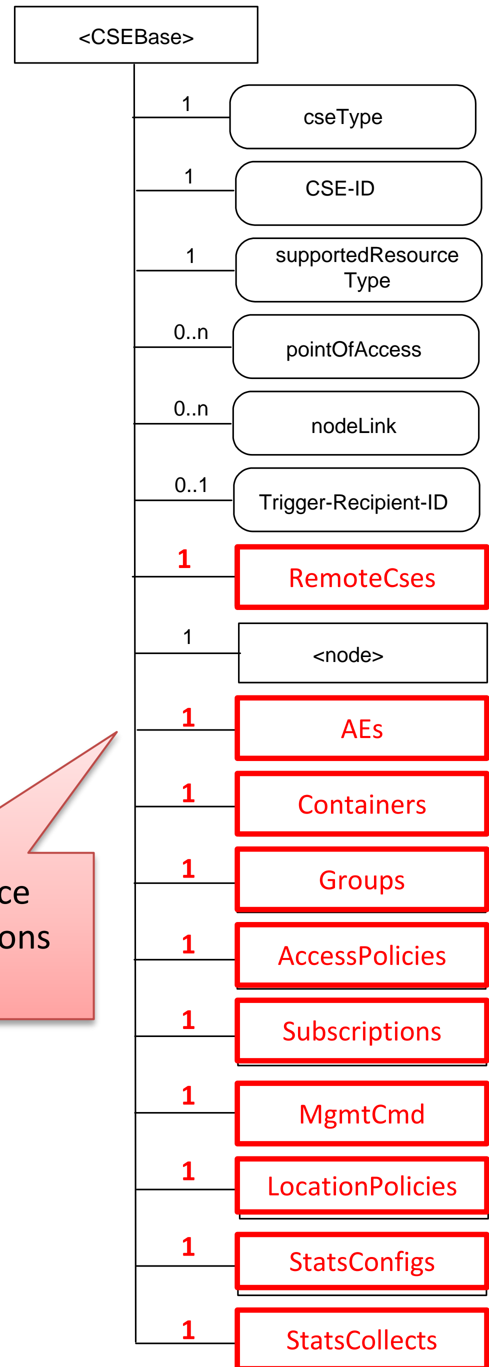


Figure 9.6.3-1: Structure of <CSEBase> resource

Impact on PRO

Child resource types must contain collections resources (e.g. AEs, Containers, etc.)

Child Resource Type Name	Multiplicity	Ref . to in Resource Type Definition
remoteCse(variable)	0..n	7.3.1
node(variable)	0..n	
AE (variable)	0..n	7.3.3
container(variable)	0..n	7.3.4
group(variable)	0..n	7.3.11
accessControlPolicy	0..n	
subscription(variable)	0..n	7.3.6
mgmtCmd(variable)	0..n	7.3.13
locationPolicy(variable)	0..n	7.3.8
statsConfig(variable)	0..n	7.3.29
statsCollect(variable)	0..n	7.3.31

Table 7.3.2.1-4: Reference of child resources

XSDs and XML file must be updated to have collection .

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.onem2m.org/xml/protocols"
  xmlns:m2m="http://www.onem2m.org/xml/protocols"
  elementFormDefault="unqualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="common_types-v1_0_0.xsd" />

  <xs:element name="container">
    <xs:complexType>
      <xs:complexContent>
        <!-- Inherit Common Attributes from regularResourceType -->
        <xs:extension base="m2m:regularResourceType">
          <!-- Resource Specific Attributes -->
          <xs:sequence>
            <xs:element name="maxNrOfInstances" type="xs:nonNegativeInteger"
              minOccurs="0" />
            <xs:element name="maxByteSize" type="xs:nonNegativeInteger"
              minOccurs="0" />
            <xs:element name="maxInstanceAge" type="xs:nonNegativeInteger"
              minOccurs="0" />
            <xs:element name="currentNrOfInstances" type="xs:nonNegativeInteger" />
            <xs:element name="currentByteSize" type="xs:nonNegativeInteger" />
            <xs:element name="latest" type="xs:anyURI" minOccurs="0" />
            <xs:element name="locationID" type="xs:anyURI"
              minOccurs="0" />
            <xs:element name="ontologyRef" type="xs:anyURI"
              minOccurs="0" />

            <!-- Child Resources -->
            <xs:element name="childResource" type="m2m:childResourceType"
              minOccurs="0" maxOccurs="unbounded" />
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

C.1.XML Schema for container resource type

```xml
<?xml version="1.0" encoding="UTF-8"?>
<m2m:container xmlns:m2m="http://www.onem2m.org/xml/protocols"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
   "http://www.onem2m.org/xml/protocols CDT-container-v1_0_0-20140609.xsd "
    name="12xx">
  <parentID>//IN-CSEID.m2m.myoperator.org/96734</parentID>
  <accessControlPolicyIDs>//IN-CSEID.m2m.myoperator.org/93405</accessControlPolicyIDs>
  <creationTime>2013-12-31T12:00:00</creationTime>
  <expirationTime>2013-12-31T12:30:00</expirationTime>
  <lastModifiedTime>2013-12-31T12:00:00</lastModifiedTime>
  <stateTag>0</stateTag>
  <labels>label1 label2</labels>

  <maxNrOfInstances>5</maxNrOfInstances>
  <maxByteSize>104857600</maxByteSize>
  <maxInstanceAge>3600</maxInstanceAge>
  <currentNrOfInstances>2</currentNrOfInstances>
  <currentByteSize>6</currentByteSize>
  <latest>//IN-CSEID.m2m.myoperator.org/96739</latest>
  <locationID>//IN-CSEID.m2m.myoperator.org/1112</locationID>
  <ontologyRef>http://tempuri.org/ontologies/xyz</ontologyRef>

  <childResource name="instance1234" type="instance">//IN-CSEID/1722</childResource>
  <childResource name="instance1235" type="instance">//IN-CSEID/34722</childResource>
  <childResource name="1923" type="subscription">//IN-CSEID/2323</childResource>

</m2m:container>
```

C.2. Container resource that conforms to the Schema in C.1

Conclusion

- The collection pattern should be integrated into the OneM2M platform to be compliant with REST architecture constraints, improve system performance and solve the resource type technical problem.
- To integrate collections to OneM2M, several changes should done on ARC and PRO specifications:
  - For each resource, same type child resources must be replaced with one collection link.
  - A new Collection resource must be defined. The collection structure can be generic.

References

- « RESTful Web APIs » book - Leonard Richardson, Mike Amundsen, Sam Ruby
- Leonard Richardson Maturity Model (http://martinfowler.com/articles/richardsonMaturityModel.html)
- REST APIs must be hypertext-driven – Roy T. Fielding (http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven)
- HATEOAS and the PayPal REST Payment API (https://developer.paypal.com/docs/integration/direct/paypal-rest-payment-hateoas-links)

Tank you for your attention

LAAS-CNRS

Does the discovery mechanism solve the problem ?!

- The discovery mechanism cannot solve this architectural problem:
  - Discovery enables only to find URIs of a specific resources quickly based on some filter criteria, like a search engine. But, starting from the discovered link, a client should be able to traverse the API by reading media type and following link.
  - Rely only on the discovery mechanism to use the API is not HATEOAS or even REST.
  - In addition, to be HATEOAS, servers must instruct clients on how to construct appropriate URIs to perform discovery request by he mean of URI template, et.