LC2 Specification

Artem Chirkin, Lukas Treyer, Michael Franzen

February 21, 2017

Contents

Changelog

- 2016-08-05: First draft

1 Introduction

1.1 Notes

The words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are used in accordance to RFC2119.

1.2 Understanding the document

1.3 Editing the document

The document contains a number of JSON or GeoJSON listings representing content of Lightweight Urban Computation Interchange (Luci) actions. In the listings we use the following coloring scheme:

- Key names are shown in black (e.g. `action`);
- Reserved keywords, such as value types are shown in blue (e.g. `string`);
- Fixed strings constants are shown in red (e.g. `"create_scenario"`);
- Additional structural keywords are shown in grey (e.g. `OPT` means the key-value pair is optional, `XOR` before several key-value pairs means exactly one alternative).
- Comments are in purple, separated by double slash (e.g. `// comment`)

If there is a missing reserved keyword, you can add it into tex file annotation (`keywords` or `ndkeywords` lists in `lstdefinelanguage` command).

2 Service API

Luci is primarily intended to connect machines in a LAN. Luci works as a TCP server, and all remote services and clients are TCP clients. In the beginning, a service registers itself in Luci, and then Luci sends run requests to the service from time to time. The service on the other hand keeps Luci updated of its current status.

2.1 Registering a Service

For registering a service in Luci, a set of three parameters **must** be implemented into the service and connect to Luci using the built-in service `RemoteRegister` (Appendix A.7).

The three required fields are

- `serviceName`: The service name as a string
- `description`: The service description as a string
- `exampleCall`: An example call to the service as json encoded string

If a service accepts input arguments or outputs data, two additional parameters **can** be provided:

- `input`: A json-schema that specifies the service input.
- `output`: A json-schema that specifies the service output

2.1.1 JSON-Schema

The schema for service inputs and outputs is a dictionary assigning each argument its type. A basic example schema is given in listing 1. A json-dictionary is valid for this schema if it contains the keys `foo`, `bar` and `baz` and assigns a number, string and boolean to them, respectively.

```
1  {
2    foo : number,
3    bar : string,
4    baz : boolean
5  }
```

Listing 1: JSON Schema

Arguments can be made optional and mutually exclusive using the prefixes OPT and XOR. In listing 2, only the field `foo` is obligatory. A valid JSON dictionary **must** contain the key `foo` and **can** contain the key `bar`.

```
1  {
2    foo : number,
3    OPT bar : string
4  }
```

Listing 2: JSON Schema with optional arguments

Finally, consecutive arguments can be mutually exclusive. In listing 3 the arguments `baz1`, `baz2`, `baz3` are mutually exclusive, indicating that exactly one of the three must be included.

```
1 {
2   XOR baz1 : boolean,
3   XOR baz2 : boolean,
4   XOR baz3 : boolean
5 }
```

Listing 3: JSON Schema with mutually exclusive arguments

2.1.2 Example Register-message

Listing 4 shows an example message how to register a service named *ExampleService*. The schema in the section `inputs` indicates, that the service takes up to 2 input values: an obligatory number in the field `obligatory_input` and a string in the field `optional_input`. The OPT-keyword indicates that the latter field is not a required one for the service to run.

After its execution, the service then outputs a dictionary containing the following keys:

- `obligatory_output` which is assigned a string value
- `output_value1` **or** `output_value2`. Only one of the keys is allowed to be included in the outputs.
- `optional_output` which is a boolean and **optional**. In some cases, the service might output this value and in other cases it might omit it.

```
{
  run : "RemoteRegister",
  description : "I am an exemplary service",
  serviceName : "ExampleService",
  callID : 1,
  inputs : {
    obligatory_input : number,
    OPT optional_input : string
  },
  outputs : {
    obligatory_output : string,
    XOR output_value1 : number,
    XOR output_value2 : number.
    OPT optional_output : boolean
  },
  exampleCall : {
    {
      run : "ExampleService",
      optional_input : "foo",
      obligatory_input : 12
    }
  }
}
```

Listing 4: Registering a service

The whole message is sent as a *run*-message (see section 2.2) to the pseudo-service *RemoteRegister*. Afterwards, Luci is aware of the service and associates the service *ExampleService* with the IP from which this registration message originated.

2.2 Run and Cancel

When a client sends a message in the form

```
{
  run: string, // the service name.
  callID: integer // a unique random number identifying the session
}
```

to Luci, Luci simply forwards this message to the service. The service **MAY** returns with a message containing in its run id:

```
{
  newCallID: number, // the newly generated callID
```

```
  callID: integer // the same callID, it has priority over newCallID
}
```

If a client requests to stop the execution of a certain service call, he sends the following message which is subsequently forwarded to and handled by the service.

```
{
  cancel: number, // the callID
  callID: integer // the same callID, it has priority over newCallID
}
```

2.3 Result, Error and Progress

Without any notification of Luci, the service is responsible of keeping Luci updated of the service's current status. The result is returned when the service is finished and is formatted in the following way:

```
{
  result: json, // the result as specified in the output field upon service
      registration
  callID: integer
}
```

From time to time, the service might want to keep Luci updated of it's progress. If the progress equals 0, the service has been ordered to start.

```
{
  progress: integer,
  callID: integer,
  OPT intermediateResult: json // some result, it is optional.
}
```

Finally, if an error occured, the service will provide an error message to Luci:

```
{
  error: string,
  callID: integer
}
```

2.4 Attachments

Luci messages can contain binary data. It is attached to messages as byte arrays. Every message must contain a JSON "header", attachments are optional. The UTF8 encoded JSON header is human-readable; a complete luci message consists of a json string and a thin binary wrapper around the json header plus a binary attachements part.

Usually binary attachments should be referenced in a header using a special attachment description. Here is a convention for JSON format of such description:

```
{
  format: string, // e.g. "binary" or "4-byte float array"
  attachment: {
    length: number, // length of attachments in bytes
    checksum: string, // MD5 checksum
    position: integer // starting at 1; 0 = undefined position
  }
  OPT name: string,
  ANY key: string
}
```

Attachments can be referenced multiple times in a JSON header. Attachments - if they need to be forwarded to only one service - are forwarded directly to remote services, i.e. Luci will not wait until the whole attachment is being transferred to Luci before sending it to a remote service.

2.5 Geometry

Similar to attachments a JSON header can also contain JSON encoded geometry like GeoJSON. Since there are several JSON based geometry formats (e.g. TopoJSON) a JSONGeometry object must follow this structure:

```
{
  format: string, // e.g. "binary" or "4-byte float array"
  geometry: json, // format specific
  OPT name: string,
  OPT crs: string, // reference system
  OPT attributeMap: json,
}
```

3 Low-Level Networking

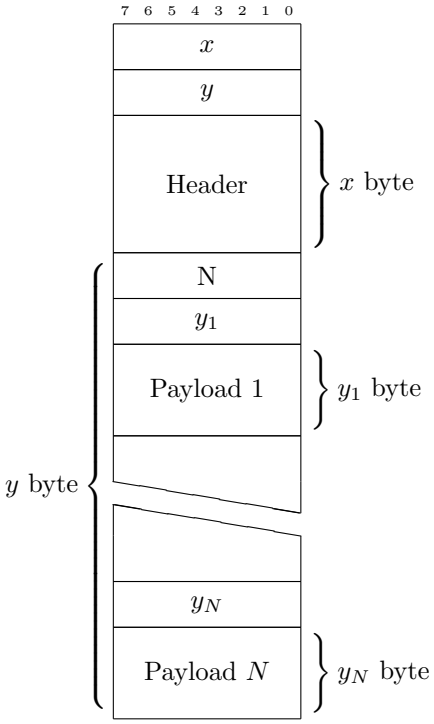

Figure 1: LC2 Message

4 Scenario Services

Luci uses GeoJSON format to represent scenario geometry. The format declares geometry information syntax, but does not declare consistent naming and geometry type mappings for Luci scenario entities, such as buildings, roads, etc. This document aims at providing guidelines on usage of Luci scenarios in Luci clients and services.

A note on jsonGeometry data type The word `geometry` in Luci specification has two different meanings. On the one hand it is a name of the key that occurs from time to time in Luci actions. On the other hand it is a name of a pre-defined data type that represents scenario geometry in JSON format. To resolve this ambiguity, in current document we use word `geometry` to represent the name of the key, and `jsonGeometry` to represent the data type. This differentiation does not introduce any changes to the existing JSON messages.

4.1 Luci scenario actions

To send geometry to Luci, we wrap it in the structure that is shown in listing 5 Special type `jsonGeometry` wraps various types of geometry processed by Luci. It allows to enclose arbitrary number of custom-named geometries (keys `KEY_NAME_N`) inline (GeoJSON) or in separate files (as binary attachments).

```
{
  /* key name is an arbitrary string.
     The convention is to name it by filename of a file,
     or arbitraryname.geojson in case of GeoJSON geometry */
  KEY_NAME_1 :
    XOR { // "in-line" geometry
      format   : string // "GeoJSON", later add also TopoJSON, CurveJSON, ...
      geometry : object // GeoJSON FeatureCollection
      OPT crs  : string // name of a crs
      OPT attributeMap : { // mapping between Luci and foreign types
        LUCI_ATTRIBUTE_NAME : FOREIGN_ATTRIBUTE_NAME,
        ...
      }
    }
    XOR { // "streaminfo" - attachment description
      format     : string // shp | shx | dbf | any other file format?
      streaminfo : {
        checksum : string // MD5 sum of an attached binary data
        length : long // length of the attached binary data
        order : long // number of attachment (starts with 1)
      }
      OPT crs    : string // name of a crs
      OPT attributeMap : { // mapping between Luci and foreign types
        LUCI_ATTRIBUTE_NAME : FOREIGN_ATTRIBUTE_NAME,
        ...
      }
    },
  KEY_NAME_2 : ...,
  ...
}
```

Listing 5: structure of `jsonGeometry` data type

The type of object `geometry` is GeoJSON `FeatureCollection` – the format that is described in GeoJSON specification[1].

[1] `http://geojson.org/geojson-spec.html`

According to `spec/LuciSpecification.pdf`, Luci provides three operations to work with scenarios: `create_scenario`, `update_scenario`, and `get_scenario`. This document covers only GeoJSON geometry manipulation; in this format individual entities are represented as `Feature` objects in `FeatureCollection`. Each `Feature` has a property `geomID:long` that is given either by a client, or by Luci (in case if client application does not specify property `geomID`).

Creating a scenario is done via `create_scenario` action shown in listing 6. The action allows to specify a location (`projection`) and a geometry to put inside the new scenario.

```
{
  action           : "create_scenario", // constant, represents the action
  name             : string, // name of the scenario
  OPT projection   : {
    XOR crs    : string, // name of a crs
    XOR bbox   : [ [number, number]   // top-left coords [lat, long]
                 , [number, number]], // bottom-right coords [lat, long]
  },
  OPT geometry     : jsonGeometry, // wrapper around various geometry types
  OPT switchLatLon : boolean // switch lat-long to long-lat in geometry
}
```

Listing 6: JSON action structure for creating a scenario in Luci

Listing 7 shows the action to update scenario geometry. The action allows to change the name and the bounding box, as well as the geometry inside.

```
{
  action       : "update_scenario", // constant, represents the action
  ScID         : long, // ID of the scenario in Luci
  OPT name     : string, // set a new name for the scenario
  OPT bbox     : [ [number, number]   // top-left coords [lat, long]
                 , [number, number]], // bottom-right coords [lat, long]
  OPT geometry : jsonGeometry // wrapper around various geometry types
  OPT switchLatLon : boolean // switch lat-long to long-lat in geometry
}
```

Listing 7: JSON action structure for updating a scenario in Luci

- In order to add an entity to the scenario, one adds `Feature` into `FeatureCollection` (`geometry` object).
- In order to modify an existing entity, one must specify its property `geomID` (if given `geomID` does not exist, the entity is added to the scenario, otherwise it is edited).
- In order to delete a number of entities from the scenario, one adds an empty `Feature` into `FeatureCollection` that has a property `deleted_geomIDs:[long]` – array of `geomID` for deletion.

Listing 8 shows the action to get scenario geometry. The action allows to specify the format of the data to (Luci does transformation), and get the geometry from the scenario at given time.

```
{
  action             : "get_scenario", // constant, represents the action
  XOR scenarioname   : string // name of the scenario in Luci
  XOR ScID           : long, // ID of the scenario in Luci
  OPT format_request : string, // maybe we will change this later to "format"
  OPT crs            : string,
  OPT geomIDs        : [long], // select a subset of scenario objects
  OPT timerange      : { // time is a number - timestamp in unix format
    XOR until   : long,
    XOR from    : long,
    XOR between : [long,long],
    XOR exactly : long,
```

```
13      OPT all      : boolean // include all versions (not only the last)
14    }
15  }
```

Listing 8: JSON action structure for getting a scenario from Luci

4.2 Scenario GeoJSON geometry

Although GeoJSON specification provides all necessary geometry primitives, we need a more structured convention to define one-to-one mapping between the geometry and scenario entities. Luci does not generate an error for an input that does not follow it – the convention only describes what kind of data structures services and clients should expect.

Section 4.2.1 describes the rules assumed by Luci when communicating with all services and clients. Section 4.2.2 describes the rules assumed by most applications, but not checked in Luci. Section 4.2.3 describes the application-specific rules. Any service or client provider (Luci user) may introduce a rule that is used in their application. The providers are encouraged to add these rules into section 4.2.3. By an agreement in our team some of the rules go up from section 4.2.3 to section 4.2.2. In case of wide usage they might be enforced in Luci, thus moving one step up to section 4.2.1.

4.2.1 Standardized Rules

Object `geometry` in listing 5 is assumed to be of type `FeatureCollection`. Every entity is represented as a `Feature` inside that collection, and has a property `geomID:long` that is given by a client or Luci (if the client omits the property).

4.2.2 Conventional Rules

Some services require 3D objects, others use only 2D footprints. To distinguish these two types of geometry, we agreed on using `Feature` property `layer`.

- Object (e.g. Building) – a 3D geometry, represented as `Feature` with `geometry` field of type `Polygon` or `MultiPolygon`. To be processed correctly by Luci services, the object requires property `layer:"buildings"`.
- Footprint – a 2D geometry, represented as `Feature` with `geometry` field of type `Polygon` or `MultiPolygon`. To be processed correctly by Luci services, the footprint requires property `layer:"footprints"`.

4.2.3 Per-application Rules

Web geometry modeler The application distinguishes dynamic and static geometry: dynamic geometry can be edited, static geometry is only used for evaluation and visualization. Thus, I propose an optional property `static:boolean`. Absence of a property implies `static:false`.

4.2.4 Example for Scenario Usage

The following example illustrates a service registering to Luci and being subsequently called by a client. The service queries a scenario from the scenario service and returns some analysis result to the client.

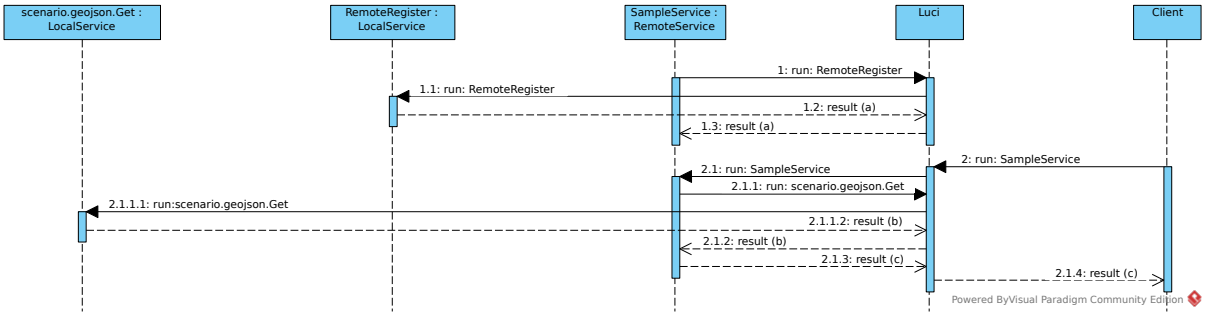

Figure 2: A SampleService that analyzes a scenario registers itself to Luci and is called by client as UML sequence diagram.

5 QUA-Compliance

To make a service compliant for use in the qua-kit, a service is subject to additional constraints that are described here. A qua-view-compliant service must support one or more of following working modes:

- `scenario` – a service gets a scenario as input and returns a single result for a whole scenario;
- `objects` – a service gets a scenario and ids of individual objects and returns a result per every object;
- `points` – a service gets a scenario and a grid of points, and returns a result per each point on the grid;
- `new` – a service updates or creates a new geometry.

5.1 QUA-View-Compliance

Defining a service being `qua-view-compliant` means promising to follow a strict predefined format of service input, output, and parameters. To declare a service `qua-view-compliant`, one needs to add `qua-view-compliant:true` flag in the root of `RemoteRegister` JSON message sent to luci/helen at the start of a service execution. To test if a service follows strictly the specification is a responsibility of a service developer.

5.1.1 Optional parameters

A `qua-view-compliant` service may have arbitrary number of optional parameters. All optional parameters must be defined in an `inputs` object of `RemoteRegister` JSON message using `paramName:paramType` syntax. Valid parameter types are:

1. `boolean` – either `true` or `false`;
2. `number` – any number type;
3. `string` – a fixed string, must have a corresponding field in `constraints` object.

5.1.2 Constraints

Parameters may have constraints defined on them. These constraints are used by qua-view control panel to configure user input fields. To define a constraint, one needs to use `constraints` object in the root of `RemoteRegister` JSON message (see listing 9).

```
1  {
2    run : "RemoteRegister",
3    ...
4    qua-view-compliant : true,
5    inputs : {
6      ...
7      mode : "string",
8      constrainedInput1 : "number",
9      constrainedInput2 : "string"
10   },
11   outputs : {
12     ...
13   },
14   constraints : {
```

```
15        mode: ["points", "scenario"],
16        constrainedInput1 : {
17            min : 42,
18            integer : true
19        },
20        constrainedInput2 : ["hello", "world", "3rd option"]
21    },
22    ...
23  }
```

Listing 9: qua-compliance – parameter constraints

Number To describe constraints on a number type, one needs to create a JSON object with following (all optional) records:

- `min:number` – minimum allowed value;
- `max:number` – maximum allowed value;
- `def:number` – default value visible in a viewer;
  - `(min = undef, max = undef, def = undef):  def = 0;`
  - `(min = exist, max = undef, def = undef):  def = min;`
  - `(min = undef, max = exist, def = undef):  def = max;`
  - `(min = exist, max = exist, def = undef):  def = (max - min) / 2;`
- `integer:boolean` – whether to treat a number as an integer; default `integer:false`.

String To describe contraints on a string, one needs to create a JSON array of strings – an enumeration of possible value. Without a constraint object, a parameter accepts any string value; with a constraint object, a parameter is displayed as a drop-down list of possible options (see listing 9 constraints on a `constrainedInput2` input key). The first option in a list is the default one.

Boolean One possible constraint for a boolean is a default value in qua-view. Use {`def:true`} or {`def:false`} constructs.

5.2 Execution modes

The execution modes define how a service is visualized in qua-view. Each mode dictates obligatory input and output parameters. A service may support more than one execution modes. At runtime, an execution mode is defined by `inputs` record `mode:string`. A qua-view-compliant service must define `inputs/mode` field in `RemoteRegister` JSON message and a corresponding constraint (the same way as optional parameters).

5.2.1 points

In this mode, qua-view generate a grid of points in 3D and sends them to a service together with scenario reference. A service sends back floating point values for each point. Then, qua-view visualizes the values as a coloured heat map. The grid of points is sent in a binary attachment in form `XYZXYZ...XYZ`, 4 byte floating point value per coordinate in Little Endian

format (`float32`) – $3 \cdot 4n$ bytes in total, where n is the number of points. A service determines a number of points by the size of incoming data.

```
{
  run : "RandomQUA",
  callID : 283,
  ScID : 123,
  mode : "points",
  points : {
    format : "float32 array",
    attachment {
      length : 3072,
      position : 1,
      checksum : "2929bead3ee3cf55113ec9aade2b6add"
    }
  }
}
```

Listing 10: A qua-compliant service run request for mode `points`

```
{
  callID : 283,
  result : {
    units : "metre",
    values : {
      format : "float32 array",
      attachment : {
        length: 1024,
        checksum: "50752c7cb358a5fffd913d9aa3605433",
        position: 1
      }
    }
  }
}
```

Listing 11: A qua-compliant service output for mode `points`

A result of a service execution must contain following fields:

1. `units` – any string, e.g. "metre", "kg", "mm/s", etc.;
2. `values` – attachment object containing a reference to a blob of `float32` data; its size must be exactly $4n$, where n is the number of points received.

5.2.2 objects

This mode is similar to `points` mode except instead of point coordinates a client (qua-view) sends ids of geometric objects within a scenario. Qua-view visualizes it by colouring objects according to the values received for each object. A user also can click on a building and see exact numeric value received from a service.

A service in this mode has two obligatory inputs:

1. `ScID` – id of a current scenario;
2. `objIds` – an attachment reference to a binary blob of size $4n$, where n is the number of objects to process by a service; the blob consists of unsigned 4-byte integers in Little Endian format (`uint32`) – `geomID` of each object.

```
{
  run : "RandomQUA",
  callID : 172,
```

```
  ScID : 123,
  mode : "objects",
  geomIDs : {
    format : "uint32 array",
    attachment {
      length : 1024,
      position : 1,
      checksum : "50752c7cb358a5fffd913d9aa3605433"
    }
  }
}
```

Listing 12: A qua-compliant service run request for mode `objects`

A result of a service execution must contain following fields:

1. `units` – any string, e.g. "metre", "kg", "mm/s", etc.;
2. `values` – attachment object containing a reference to a blob of `float32` data; its size must be exactly $2n$, where n is the number of objects' `geomID` received.

```
{
  callID : 172,
  result : {
    units : "metre",
    values : {
      format : "float32 array",
      attachment : {
        length: 1024,
        checksum: "2929bead3ee3cf55113ec9aade2b6add",
        position: 1
      }
    }
  }
}
```

Listing 13: A qua-compliant service output for mode `objects`

5.2.3 scenario

A scenario execution mode returns only one value for a whole scenario. The value can be an arbitrary string or a `.png` image (in attachment). In either way, the result is displayed on a side panel in qua-view.

The only obligatory field in this mode is a scenario id `ScID`.

```
{
  run : "RandomQUA",
  callID : 213,
  ScID : 125,
  mode : "scenario"
}
```

Listing 14: A qua-compliant service run request for mode `scenario`

One possible result of a scenario-mode-service is a text string.

```
{
  callID : 213,
  result : {
    answer : "This scenario looks good! The goodness level is 9.125."
  }
```

```
}
```

Listing 15: A qua-compliant service output for mode `scenario` (string output)

Another possible result type is a `.png` image attached to a message. Recommended size of an image is 512x512px. An image may contain a graph or an arbitrary figure.

```
{
  callID : 213,
  result : {
    image : {
      format : "image/png",
      attachment : {
        length: 12623,
        checksum: "8329beex3ee3cf55113ec9aade2b6a02a",
        position: 1
      }
    }
  }
}
```

Listing 16: A qua-compliant service output for mode `scenario` (image output)

5.2.4 new

In this mode, a service updates or creates a new geometry. It can be used by various geometry-generating services. A `ScID` key is optional. This mode does not have obligatory parameters. A service should write its output directly into scenario via luci/helen. The output of a service must have following fields:

```
result: {
  ScID : long, // the id of the newly created scenario
  timestamp_accessed : long, // optional - the timestamp at which the scenario
      was accessed if the ScID-field was set in the input
  timestamp_modified : long // the timestamp at which the new scenario was
      stored
}
```

```
{
  run : "RandomQUA",
  callID : 244,
  mode : "new" // Note that the call does not contain a ScID (though it can)
}
```

Listing 17: A qua-compliant service run request for mode `new`

```
{
  callID : 244,
  result : {
    ScID : 1337,
    timestamp_accessed : 1472137755,
    timestamp_modified : 1472137758
  }
}
```

Listing 18: A qua-compliant service output for mode `new`

5.3 Registering a service

When a service connects to luci/helen, it must register itself, so clients know what does the service do and how to call it. This is done via `RemoteRegister` service call. Content of this message is the only source of information for a client (such as qua-view) to determine how to call the service. Especially important in this sense is the `constraints` field of a message, because it defines in which modes a service can be called and how to visualize parameters selection in GUI. Listing 19 shows a registration message of a qua-compliant service named *RandomQUA*. In the following, we will briefly reiterate what each field indicates.

- Lines 1-5: A service of name *RandomQUA* is registered. The fields are not subject to any change by qua-compliance. See Section 2, for how to register a generic service.
- Line 6: The field `qua-view-compliant` indicates that the service is qua-view-compliant.
- Lines 7-15 specify the service's inputs
  - The field `ScID` is optional, it is used to specify the scenario on which the service is asked to operate. Since not every execution mode requires such a scenario, the field is optional.
  - The execution mode `mode` is obligatory and, as can be noted from the constraints in line 26, takes the four execution mode names *points, objects, scenario* and *new*.
  - The field `points` is an optional binary attachment that is required when the service is executed in the *points*-execution-mode
  - The field `geomIDs` is an optional binary attachment specifying the object ids when the service is executed in the *objects*-execution-mode
  - The other fields are service-specific fields. Generally, a service can add an arbitrary amount of additional inputs for the service.
- Lines 16-24 specify the service's outputs
  - The field `units` gives the units of the service result
  - The `values` field contains the service's result values in the modes `points` or `objects`. The `values` field refers to a binary attachment in which the result values are encoded as floating point numbers (32bit, LE). On the other hand, if the service operates in the *scenario*-mode, it contains an `image` or a string `answer` as a single result.
  - The fields `ScID`, `timestamp_accessed` and `timestamp_modified` describe the outputs if the service runs in the mode *new*. They describe the scenario id, its version number (timestamp) when it was accessed and its version number after it has been modified, respectively.
- Lines 25-41 specify the service's input constraints
  - Line 26: The mode must be a subset of {`points, objects, scenario, new`}
  - Line 27-40: all other constraints are optional.
- Lines 42-60 give an example of a valid call of the service. In this case, the service is asked to operate on scenario no. 123. The service is executed in mode *points*; length of the *points*-attachment is 36 bytes which corresponds to 3 three-dimensional points. Finally, the call specified all service parameters.

```
{
  run : "RemoteRegister",
  description : "I am a random service",
  serviceName : "RandomQUA",
  callID : 1,
  qua-view-compliant : true,
  inputs : {
    OPT ScID : number,  // obligatory for any mode except "new"
    mode : string,
    OPT points : attachment, // obligatory for mode "points"
    OPT geomIDs : attachment, // obligatory for mode "objects"
    ANY numberKeyX : number,
    ANY stringKeyX : number,
    ANY booleanKeyX : number
  },
  outputs : {
    OPT units  : string,     // only in mode "points" or "objects"
    OPT values : attachment, // only in mode "points" or "objects"
    OPT answer : string,     // only in mode "scenario"
    OPT image  : attachment, // only in mode "scenario"
    OPT ScID   : long,             // only in mode "new"
    OPT timestamp_accessed : long, // only in mode "new"
    OPT timestamp_modified : long  // only in mode "new"
  }
  constraints : {
      mode : ["points", "objects", "scenario", "new"], // obligatory
      OPT numberKey1 : {
          min : 42,
          max : 100,
          def : 50,
          integer : true
      },
      OPT numberKey2 : {
          min : 0.001,
          def : 2
      },
      OPT stringKey1 : ["hello", "world"],
      OPT booleanKey1 : {
          def : true
      }
  },
  exampleCall : {
    {
      run : "RandomQUA",
      ScID : 123,
      mode : "points",
      points : {
        format : "float32 array",
        attachment {
          length : 36,
          position : 1,
          checksum : "2929bead3ee3cf55113ec9aade2b6add"
        }
      }
      numberKey1 : 42,
      numberKey2 : 13.37,
      stringKey1 : "hello",
      booleanKey1 : false
    }
  }
}
```

Listing 19: Registering a QUA-compliant service

A Built-In Services

A.1 AttachmentJSON

Returns the json specification of attachments and json geometry objects.

Inputs:

- **run**:

```
AttachmentJSON
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  attachment: attachment,
  jsongeometry: jsongeometry
}
```

A.2 Download

Downloads a file from a given URL and stores is in Luci's attachment folder as <md5 checksum>.<format>

Inputs:

- **run**:

```
Download
```

- **url**: the URL to be used

```
string
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  checksum: string// the checksum in hexadecimal representation,
  format: string// equals the string typically put as a file ending,
  length: number// the length of the attachments as number of bytes
}
```

A.3 Exists

Tests whether a service with the given serviceName or a task with given taskID exists.

Inputs:

- **run**:

```
Exists
```

- **XOR instanceID**:

```
number
```

- **XOR serviceName**: the name of the service to be tested

```
string
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  exists: boolean
}
```

A.4 FilterServices

Filters services according to a) its keys at a given recursion level, b) its types at a given recursion level or c) a given json template with values equal to 'null' meaning only keys are compared.

Inputs:

- **run**:

```
FilterServices
```

- **XOR jsonMatch**:

```
json
```

- **XOR keys**:

```
[
  string
]
```

- **XOR types**:

```
[
  string
]
```

- **OPT rcrLevel**:

```
number
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  serviceList: [
    string
  ]
}
```

A.5 GetStartupTime

Returns the startup of time of Luci.

Inputs:

- **run**:

```
GetStartupTime
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  startupTime: number// in milliseconds since 1.1.1970
}
```

A.6 RemoteDeregister

Deregisters a client from a service registration. This will cancel any running service call, cancel all subscriptions of this client and, if the service to be deregistered is the last remaining of its kind, its serviceName will be removedfrom the list of available services.

Inputs:

- **run**:

```
RemoteDeregister
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  deregisteredName: string// Intended to inform listeners about the service
      name that was deregistered,
  id: number// int, the id that was deregistered,
  nodeIP: string// Intended to inform listeners about the IP of the machine
      that deregistered the service,
  remainsAvailable: boolean// boolean to indicate whether other services
     with the samename remain  available (true) or whether the service is
     being deleted from the listof  available services (false)
}
```

A.7 RemoteRegister

Registers a client as a service.

Inputs:

- **run**:

```
RemoteRegister
```

- **description**:

```
string
```

- **exampleCall**: Mandatory json object that shows an example call. Since OPT/XOR/ANY modifiers are only allowed in in & output specifications and not in calls, an example call shows very clearly how a call could look like.

```
json
```

- **serviceName**: mandatory string that identifies the service

```
string
```

- **OPT id**: Optional int to be used to identify a remote service. E.g. for viewer services it might be important to be able to identify remote service instances. If the id is taken already an error will be thrown / registration aborted.

```
number
```

- **OPT inputs**: Optional json object holding the input description to be included by 'ServiceInfo' / the API

```
json
```

- **OPT outputs**: Optional json object holding the output description to be included by 'ServiceInfo' / the API

```
json
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  id: number// The int given as input or an int generated by Luci,
  nodeIP: string// The IP of the client that was registered as a service (
      intended to inform listeners),
  registeredName: string// Intended to inform listeners about the service
      name that was registered
}
```

A.8 ServiceControl

Used to remotely administrate installed services on a selected machine or on allmachines having a 'ServiceControl' remote service running.

Inputs:

- **run**:

```
ServiceControl
```

- **XOR info**: get filtered information about either selected machines (nodes) or about selectedserviceNames on all machines or full information about all machines in case this value is 'null'

```
{
  \null\: error,
  {XOR nodes: {
    OPT threadPool: boolean,
    XOR nodes: [
      string
    ],
    XOR serviceNames: [
      string
    ]
  }
}
```

- **XOR install**: install files on either selected machines (nodes) or on all machines (services)

```
{
  XOR nodes: {
    ANY IP: [
      attachment
    ]
  },
  XOR services: [
    attachment
  ]
}
```

- **XOR remove**: remove services identified by their name (serviceName) either on selected machines (nodes) or on all machines (services)

```
{
  XOR nodes: {
    ANY IP: [
      string
    ]
  },
  XOR services: [
    string
  ]
}
```

- **XOR start**: startup a given number of services either globally or on selected machines (nodes)

```
{
  XOR nodes: {
    ANY IP: {
      ANY serviceName: {
        \number\,: error,
        {ANY id: {
          ANY id: string
        }
      }
    }
  },
  XOR services: {
    ANY serviceName: number
  }
}
```

- **XOR stop**: stop a given amount of services either globally or on specified machines (nodes).

```
{
  XOR nodes: {
    ANY IP: {
      ANY serviceName: number
    }
  },
  XOR services: {
    ANY serviceName: number
  }
}
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  OPT errors: list// 'TODO' in case at least one of the items to  install/
      remove/start/stop produced an error this list contains one error
      string per item,
  XOR info: {
    OPT quickPool: {
      avgWaitingTime: number,
      maxPoolSize: number,
      numCalls: number,
      poolSize: number,
      waitingInQueue: number
    },
    OPT slowPool: {
      avgWaitingTime: number,
      maxPoolSize: number,
      numCalls: number,
      poolSize: number,
      waitingInQueue: number
    },
    XOR nodes: {
      ANY ip: {
        OPT installedServices: {
          ANY serviceName: {
            exec: string// to be run in console,
            numRequestedCPUCores: number// 0 = using all available cores
          }
        },
        OPT system: {
          CPU: {
            archNumBits: number// int; 32 or 64,
            endian: string// 'big' or 'little',
            numCores: number
          },
          name: string,
          osname: string,
          osversion: string
        },
        services: {
          busy: {
            ANY serviceName: number
          },
          idle: {
            ANY serviceName: number
          },
          running: {
            ANY serviceName: number
          }
        }
      }
    },
    XOR services: {
      ANY serviceName: {
        OPT available: number// not busy,
        OPT nodes: list// list of IPv4,
        OPT running: number,
        avgDuration: number// in milliseconds,
        numCalls: number// how many times this service has been called
            already
      }
    },
    remoteSummary: {
      busy: number,
      idle: number
```

```
    }
  },
  XOR installed: {
    ANY IP: [
      string
    ]
  },
  XOR removed: {
    ANY IP: [
      string
    ]
  },
  XOR started: {
    ANY IP: {
      ANY serviceName: number
    }
  },
  XOR stopped: {
    ANY IP: {
      ANY serviceName: number
    }
  }
}
```

A.9 ServiceHistory

Returns the history of a requested service as in number of calls and availableworkers either since startup time or during the requested period

Inputs:

- **run**:

```
ServiceHistory
```

- **serviceName**:

```
string
```

- **OPT period**: in seconds

```
number
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  calls: {
    { callID: {
      callID: number,
      idleTime: number,
```

```
      readingTime: number,
      runningTime: number,
      timestamp: number,
      writingTime: number
    }
  },
  workers: {
    { allWorkers: {
      allWorkers: number,
      idleWorkers: number,
      timestamp: number
    }
  }
}
```

A.10 ServiceInfo

Returns all known information on selected services such as a short description like this one, in- and output description and an example call.

Inputs:

- **run**:

```
ServiceInfo
```

- **OPT inclDescr**: indicates whether descriptions such as this one should be included in the result

```
boolean
```

- **OPT serviceNames**: Optional list of selected service names for which information should be returned. If no such list is given, the result will show the structure of general json structures such as 'attachment', 'jsongeometry'

```
list
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  ANY serviceName: {
    OPT numNodes: number,
    example: json,
    inputs: json,
    outputs: json
  }
}
```

A.11 ServiceList

Get a list of all loaded and registered services as a list of service names.

Inputs:

- **run**:

```
ServiceList
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  installed: list,
  serviceNames: list
}
```

A.12 task.Create

Create a task. A task corresponds to a node in the workflow diagram.

Inputs:

- **run**:

```
task.Create
```

- **parentID**: the taskID of the workflow in which this task is being created

```
number
```

- **position**: x/y from top left

```
{
  x: number,
  y: number
}
```

- **serviceName**:

```
string
```

- **OPT inputs**: if omitted example values will be inserted

```
json
```

- **OPT listensToDone**: a list of taskIDs to which this task generally listens (=not listens to specific outputs)

```
list
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  name: string,
  taskID: number
}
```

A.13 task.Get

Returns infos about a task (requesting a taskID) or returns a list if ids representingthe same service

Inputs:

- **run**:

```
task.Get
```

- **XOR serviceName**: requests a list of taskID representing the same service

```
string
```

- **XOR taskID**: requests infos about a specific task

```
number
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  XOR task: {
    OPT elements: list// if the task is a workflow this list contains all children,
    OPT services: list// if the task is a workflow this list contains all services that need to started up (if any are set),
    inputSchema: json// the current input values,
    inputs: json// the inputs with subscriptions being represented as {'taskID':'number','key':'string (the name of the output key of the task referenced by taskID)'},
    listensToDone: list// a list of taskIDs the task is listening to generally (=not to specific outputs),
    name: string// can be different from service name,
    parentID: number// the taskID of a workflow to which this task belongs; 0 = no parent ID / root workflow,
    position: json// {'x':'number','y':'number'},
    taskID: number
  },
  XOR taskIDs: list
}
```

A.14 task.Remove

Remove tasks.

Inputs:

- **run**:

```
task.Remove
```

- **XOR serviceName**: removes all tasks representing the service with this name

```
string
```

- **XOR taskIDs**: a list of taskIDs to be removed

```
list
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  taskIDs: list
}
```

A.15 task.Revert

Reverts a task to the last saved state.

Inputs:

- **run**:

```
task.Revert
```

- **taskID**:

```
number
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**: refer to ¡a href=’#task.get’¿task.Get¡/a¿ for descriptions.

```
{
  OPT elements: list,
  OPT services: list,
  inputSchema: json,
  inputs: json,
  listensToDone: list,
  name: string,
  parentID: number,
  position: json,
  taskID: number
}
```

A.16 task.SubscribeTo

Subscribes the current client to a taskID or any call to a service indicated byserviceName

Inputs:

- **run**:

```
task.SubscribeTo
```

- **XOR serviceName**: if a client subscribes to a service, it gets a notification upon any output created by a service with that name (technically: the client subscribes to the service factory rather than to a specific service)

```
string
```

- **XOR taskIDs**: a list of taskIDs to which the client should be subscribed

```
list
```

- **OPT inclResults**: specifies whether the client that subscribes only get a notification (false) or all the results produced by a service (true)

```
boolean
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  success: boolean
}
```

A.17 task.UnsubscribeFrom

Unsubscribe from a service or a list of taskIDs.

Inputs:

- **run**:

```
task.UnsubscribeFrom
```

- **XOR serviceName**:

```
string
```

- **XOR taskIDs**:

```
list
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  success: boolean
}
```

A.18 task.Update

Update a the input values / subscriptions of a task.

Inputs:

- **run**:

```
task.Update
```

- **taskID**:

```
number
```

- **OPT inputs**: inputs to be changed; subscriptions are represented by 'property':{'taskID':'number','key':'string (output key)'}

```
json
```

- **OPT listensToDone**: a list of taskIDs to which the task should listen generally (=not to a specific output)

```
list
```

- **OPT name**: the name of the task (can be different to service name)

```
string
```

- **OPT outputs**: if the task to be updated is a workflow that is contained by another workflow (e.g. a group) the workflow can have outputs that might be linked to outputs of some task it contains

```
json
```

- **OPT position**:

```
{
  x: number,
  y: number
}
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  OPT unknownKeys: list,
  taskID: number
}
```

A.19 test.Delay

Used to simulate a service with a long calculation time.

Inputs:

- **run**:

```
test.Delay
```

- **seconds**:

```
number
```

- **ANY inputs**: any inputs are being forwarded

```
any
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  ANY outputs: any
}
```

A.20 test.Error

A service that simulates an error being thrown.

Inputs:

- **run**:

```
test.Error
```

- **OPT message**: optional error message

```
string
```

Outputs:

- **error**:

```
string
```

A.21 test.Fibonacci

A service that serves as an (implementation) example.

Inputs:

- **run**:

```
test.Fibonacci
```

- **amount**: how many numbers should be calculated

```
number
```

- **OPT onlyLast**: return only the last number

```
boolean
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  fibonacci_sequence: list
}
```

A.22 test.FileEcho

A service to test sending and receiving attachments. Mainly used by connectionlibrary developers.

Inputs:

- **run**:

```
test.FileEcho
```

- **ANY filenames**:

```
attachment
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  ANY sameFilenames: attachment
}
```

A.23 test.Randomly

Get either an amount of random numbers or create key-value-pairs from a key list.

Inputs:

- **run**:

```
test.Randomly
```

- **XOR amount**:

```
number
```

- **XOR keys**:

```
list
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  XOR keyValuePairs: json,
  XOR randomNumbers: list
}
```

A.24 test.Validation

A service to test how type and value constraint validation works.@inputs.key1 normal string@inputs.key2 string values constraint to 'string1' and 'string2'@inputs.key3 number values constraint to 4 ranges@inputs.key4 optional attachment that must be of type 'jpg'@inputs.key5 geojson geometry@inputs.key6 optional attachment that requires types to be one of 'jpg' or 'png'@inputs.key7 optional attachment with predefied required keys

Inputs:

- **run**:

```
test.Validation
```

- **key1**:

```
string
```

- **key2**:

```
{
  constraints: [
    string1,
    string2
  ],
  type: string
}
```

- **key3**:

```
{
  constraints: error,
  type: number
}
```

- **key5**:

```
jsongeometry/geojson
```

- **OPT key4**:

```
attachment/jpg
```

- **OPT key6**:

```
{
  constraints: [
    jpg,
    png
  ],
  type: attachment
}
```

- **OPT key7**:

```
attachment/jpg
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  errorTestStrings: list,
  passed: boolean
}
```

A.25 user.Authenticate

Authenticates a user

Inputs:

- **run**:

```
user.Authenticate
```

- **email**:

```
string
```

- **password**:

```
string
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  success: boolean
}
```

A.26 user.Create

Create a user by giving username, email, and sha1 encrypted password representedwith hexadecimals

Inputs:

- **run**:

```
user.Create
```

- **email**:

```
string
```

- **password**:

```
string
```

- **OPT city**:

```
string
```

- **OPT country**:

```
string
```

- **OPT name**:

```
string
```

- **OPT organization**:

```
string
```

- **OPT street**:

```
string
```

- **OPT surname**:

```
string
```

- **OPT zip**:

```
number
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  id: number
}
```

A.27 user.Delete

Deletes a user if it does not own a group or [TODO] own a scenario

Inputs:

- **run**:

```
user.Delete
```

- **id**:

```
number
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  success: boolean
}
```

A.28 user.List

Get a list of users

Inputs:

- **run**:

```
user.List
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  groups: {
    { id: {
      id: number,
      owner: number
    }
  },
  users: {
    { email: {
      email: string,
      groupIDs: list,
```

```
      id: number
    }
  }
}
```

A.29 user.Logout

resets the connection to an unauthenticated state

Inputs:

- **run**:

```
user.Logout
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  success: boolean
}
```

A.30 user.Permission

Allows to restrict given services only to given userIDs/groupIDs.

Inputs:

- **run**:

```
user.Permission
```

- **OPT add**:

```
{
  ANY serviceName: list
}
```

- **OPT remove**:

```
{
  ANY serviceName: list
}
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  ANY serviceName: list
}
```

A.31 user.Update

Udpate user information and does sanity checks on some of the properties.

Inputs:

- **run**:

```
user.Update
```

- **id**:

```
number
```

- **OPT city**:

```
string
```

- **OPT country**:

```
string
```

- **OPT email**:

```
string
```

- **OPT name**:

```
string
```

- **OPT organization**:

```
string
```

- **OPT password**:

```
string
```

- **OPT street**:

```
string
```

- **OPT surname**:

```
string
```

- **OPT zip**:

```
number
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  success: boolean
}
```

A.32 workflow.Create

Create a workflow.

Inputs:

- **run**:

```
workflow.Create
```

- **OPT group**: a list of taskIDs that should be grouped together into a workflow

```
list
```

- **OPT name**:

```
string
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  elements: list,
  inputSchema: json,
  inputs: json,
  listensToDone: list,
  name: string,
  parentID: number,
  position: json,
  services: list,
  taskID: number
}
```

A.33 workflow.List

Returns a list of all workflows (not only root workflows).

Inputs:

- **run**:

```
workflow.List
```

- **OPT name**: restrict the list to only one entry, e.g. {'1':'New Workflow'}

```
string
```

Outputs:

- **result**: {'taskID':'name'}, not a list but a json object; since keys in json cannot be numbers, the taskIDs are converted to string

```
{
  ANY numberAsString: string
}
```

A.34 workflow.Safe

Saves a workflow to disk (in Luci). Luci uses H2's mvstore to serialize workflows to disk. In the future this would also allow to restore earlier versions of the workflow (no version tree, but linear versions).

Inputs:

- **run**:

```
workflow.Safe
```

- **taskID**:

```
number
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  change: boolean
}
```

A.35 workflow.ServiceGet

Gets the services, the instance ids and corresponding startup args associated with this workflow.

Inputs:

- **run**:

```
workflow.ServiceGet
```

- **taskID**: the taskID to indentify the workflow

```
number
```

- **OPT ids**: filter the instances to show only the ones with the given ids

```
list
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  ANY IP: {
    ANY id: {
      args: string,
      isAutoID: boolean,
      serviceName: string
    }
  }
}
```

A.36 workflow.ServiceRemove

Remove a service instance from a workflow configuration.

Inputs:

- **run**:

```
workflow.ServiceRemove
```

- **id**: the instance id

```
number
```

- **taskID**:

```
number
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  success: boolean
}
```

A.37 workflow.ServiceStart

Start up all the services that are configured to run with the given workflow (taskID).

Inputs:

- **run**:

```
workflow.ServiceStart
```

- **taskID**:

```
number
```

- **OPT serviceIDs**: optionally restrict the instances to be started only to the given IDs.

```
list
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  started: {
    ANY IP: {
      ANY serviceName: number
    }
  }
}
```

A.38 workflow.ServiceUpdate

Update a service instance that is coupled to a workflow configuration.

Inputs:

- **run**:

```
workflow.ServiceUpdate
```

- **args**: arguments to be used to start the service with

```
string
```

- **id**: the instance id

```
number
```

- **ip**: the IPv4 of the machine on which the service should run

```
string
```

- **serviceName**: the name of the service to start (using exec parameter from the service installation)

```
string
```

- **taskID**:

```
number
```

- **OPT isAutoID**:

```
boolean
```

Outputs:

- **XOR error**:

```
string
```

- **XOR progress**:

```
json
```

- **XOR result**:

```
{
  success: boolean
}
```

A.39 workflow.UpdateIO

Updates in & outputs of a workflow; difference to task.Update: it does not set the values for in & outputs, but actually creates in & outputs dynamically for a workflow, its key, type and default value

Inputs:

- **run**:

```
workflow.UpdateIO
```

- **taskID**:

```
number
```

- **OPT inputs**:

```
{
  ANY keyname: {
    default: error,
    type: string
  }
}
```

- **OPT outputs**:

```
{
  ANY keyname: {
    default: error,
    type: string
  }
}
```

Outputs:

- **XOR error**:

```
string
```

- **XOR result**:

```
{
  elements: list,
  inputSchema: json,
  inputs: json,
  listensToDone: list,
  name: string,
  parentID: number,
  position: json,
  services: list,
  taskID: number
}
```