

Remote Interoperability Protocol for online laboratories

Prepared by Luis de la Torre, Jesus Chacon and Dictino Chaos

Universidad Nacional de Educacion a Distancia (UNED)
ETS. Ingenieria Informatica
Madrid
Spain

30th October 2019

Contents

Revision History	1
1 Introduction	2
1.1 Purpose	2
1.2 Document Conventions	2
1.3 Intended Audience and Reading Suggestions	2
1.4 Product Scope	3
2 Overall Description	4
2.1 Protocol Perspective	4
2.1.1 Server Implementation Perspective	5
2.1.2 Client Implementation Perspective	6
2.2 Protocol Functions (High-Level API)	6
2.2.1 Internal Functions	6
2.2.2 External Functions	7
2.3 Protocol Communication Methods (Low-Level API)	7
2.4 Operating Environment	8
2.5 Design and Implementation Constraints	8
2.6 Assumptions and Dependencies	8
2.7 User Documentation	9
2.7.1 Method info	9
2.7.2 Method connect	9
2.7.3 Method set	10
2.7.4 Method get	10
2.8 Developer Documentation	10
2.8.1 Internal Functions of the High-Level API	10
2.8.1.1 Method basicInfo	11
2.8.1.2 Method experienceInfo	11
2.8.1.3 Method start	11
2.8.1.4 Method open	11

2.8.1.5	Method close	12
2.8.1.6	Method run	12
2.8.1.7	Method stop	12
2.8.1.8	Method get	12
2.8.1.9	Method set	12
2.8.2	Low-Level API	12
2.8.2.1	Method info - GET	13
2.8.2.2	Method connect - SSE	22
2.8.2.3	Method set - POST	23
2.8.2.4	Method get - POST	24
3	Protocol Features	26
3.1	Defining Experiences and Meta-data	26
3.2	Defining Readable and Writable Variables	26
3.3	Obtaining Meta-data	27
3.4	Reading Readable Variables and Writing Writable Variables	27
3.5	Concurrent Multi-user Connection	27
3.6	Defining and Subscribing to Server Events	28
3.6.1	Defining Server Events	29
3.6.2	Subscribing to Server Events	31
3.6.3	Built-in Event Triggering Conditions	32
	Appendices	33
	A Glossary	34
	References	35

Revision History

Revision	Date	Author(s)	Description
0.1	18.11.2018	L. de la Torre	Chapter 1 - Introduction and document structure
0.21	24.11.2018	L. de la Torre	Chapter 2 - Overall Description (Sections 2.1-2.5)
0.22	25.11.2018	L. de la Torre	Chapter 2 - Overall Description (Updated Sections 2.1-2.5 and added Sections 2.6 and 2.8)
0.23	29.11.2018	L. de la Torre	Chapter 2 - Overall Description (Updated Section 2.8.2)
0.24	30.11.2018	L. de la Torre	Chapter 2 - Overall Description (Updated Sections 2.2, 2.3 and 2.8 and finished Section 2.8.2)
0.25	02.12.2018	L. de la Torre	Chapter 2 - Overall Description (Updated Section 2.8 and added Section 2.7)
0.26	03.12.2018	L. de la Torre	Chapter 2 - Overall Description (Updated Section 2.8.1)
0.27	14.12.2018	L. de la Torre	Chapter 2 - Overall Description (Updated Section 2.7)
0.3	15.12.2018	L. de la Torre	Chapter 3 - Protocol Features (Updated Section 3.1)
0.31	04.04.2019	L. de la Torre	Chapter 3 - Protocol Features (Updated Section 3.1)
0.35	17.04.2019	L. de la Torre	Chapter 3 - Protocol Features (Updated Section 3.1 and added Sections 3.2-3.5)
0.36	29.10.2019	L. de la Torre	Chapter 3 - Protocol Features (Added Section 3.6)
0.361	30.10.2019	L. de la Torre	Chapter 3 - Protocol Features (Updated Section 3.6)

Chapter 1

Introduction

1.1 Purpose

This document is a specification of the Remote Interoperability Protocol (RIP), which was conceived at UNED for the remote operation of online laboratories (OLs). Instructions on how to correctly implement both a server and a client that talk RIP are also given.

1.2 Document Conventions

For the purpose of this document, we consider that an OL can either be a virtual laboratory (VL) or a remote laboratory (RL).

VLS are simulations and offer experimentation possibilities based on mathematical models.

RLs use lab equipment and perform the experiments in real life, just remotely.

A *simulation model* is understood as software that includes mathematical models that simulates a system for virtual experimentation purposes.

Control program is a term used in this document to refer to the software in charge of controlling and monitoring lab equipment.

In this sense, we consider that a VL always has an associated *simulation model* that is hosted and run in some computer, while a RL always has an associated *control program* that is also hosted and run in some computer.

Finally, we define an *experience* as each of the lab activities that can be carried out with an OL implementation, either through a RL or through a VL.

1.3 Intended Audience and Reading Suggestions

Audiences that may be interested in this document are educators, researchers and industry stakeholders that want or need to remotely communicate either with hardware devices or mathematical models from a web application.

More specifically, this document aims at anyone who is interested in one or more of the following points:

1. Implementing a RIP server and/or a RIP client to use RIP as the communication protocol for operating OLs.
2. Using or modifying an existing RIP server and/or RIP client implementation.

3. Making modifications on the RIP protocol itself.

In any of the above cases, it is advised to read the present document in order. Before reading this document, it is recommended to have some notions about TCP [1], HTTP [2], SSE [3] and JSON-RPC [4].

1.4 Product Scope

The objective of RIP is to offer a simple, yet powerful, communication solution usable from web clients. As such, RIP only uses pure HTTP standard protocols, supported by all major web browsers.

RIP is designed to communicate web clients with OLs; either VLs or RLs. When used to communicate with a VL, RIP exposes meta-data and input and output methods and variables related to a simulation model that is hosted and runs on a computer (usually, a remote server). When used to communicate with a RL, RIP does the same thing with a *control program* defined in a computer (usually, a remote server) to monitor and manipulate the lab equipment.

Figure 1.1 shows the usage of the RIP protocol implemented in a RIP client and a RIP server to communicate a web client with an OL. The figure represents how an OL can implement either a VL, a RL, or any combination of both, each one defined as an independent *experience*, referenced through a certain *expId* parameter.

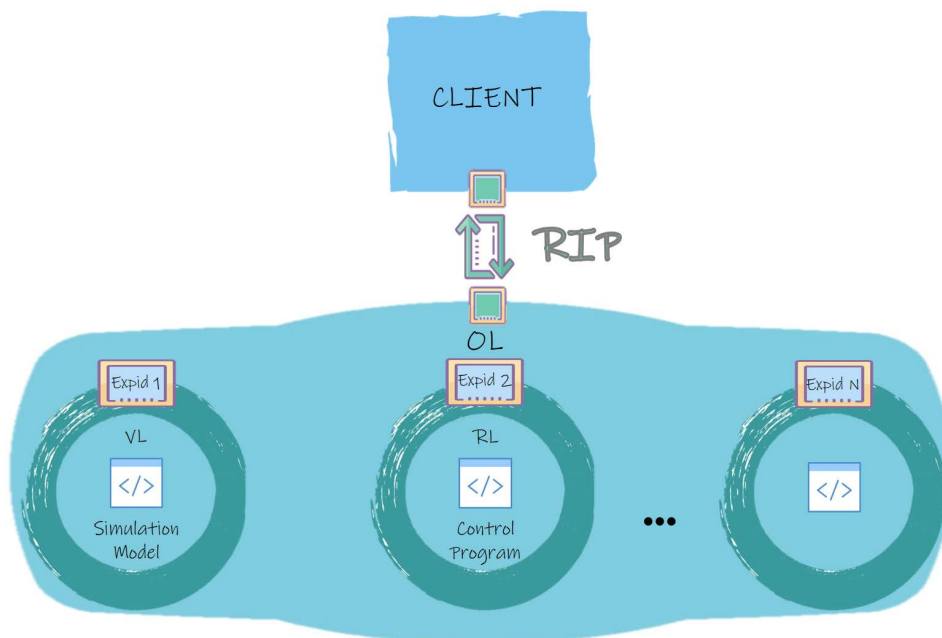


Figure 1.1: RIP protocol is used by a RIP client and a RIP server to communicate both. The RIP server is implemented in an OL and the RIP client is implemented in a web browser application.

Chapter 2

Overall Description

2.1 Protocol Perspective

The protocol is an open source, under the GNU general Public License. It is a communication protocol to be used in the client-server model, especially designed for OLs in which the client runs within a web browser. RIP provides a simple mechanism for users and client machines/programs to acquire information about the lab *experiences* defined in the server and about each *experience's* inputs and outputs. The protocol also defines methods for reading and writing the values of these inputs and outputs, respectively.

The main features of RIP are the following:

1. Defining *experiences* on the OL.
2. Obtaining meta-data related to each defined *experience*.
3. Obtaining a list of readable and writable variables for each *experience*.
4. Obtaining a list of methods to read and write variables in each *experience*.
5. Invoking methods to read and write variables in each *experience*.
6. Defining server events to send data either periodically or based on any other triggering condition defined in an *experience*.
7. Subscribing a client to any server event declared in an *experience*.

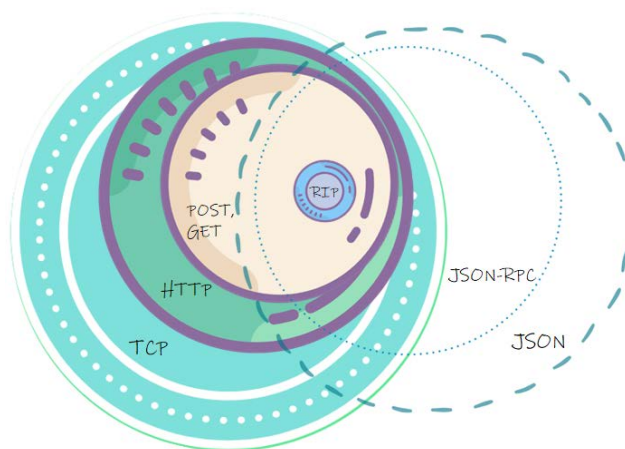


Figure 2.1: RIP is based on POST and GET HTTP methods and on the JSON-RPC format.

RIP is based on two cornerstones: POST and GET methods for the communications transport and JSON-RPC for formatting messages. POST and GET are HTTP methods, which, in turn, is based on TCP communications. On the other hand, JSON-RPC is based on the JSON format. Figure 2.1 represents these ideas.

Communication between two software entities is possible when they talk the same protocol. Therefore, a RIP implementation is needed in both the client and the server.

2.1.1 Server Implementation Perspective

Figure 2.2 depicts the architecture of a *RIP Server* that implements the RIP protocol. To sum it up, there are three functional subsystems: the *Web Server*, that handles client connections, user sessions, etc., the *Command Interpreter*, that speaks RIP, and the *Executor*, that controls the execution of the laboratory *control programs* or *simulation models*.

The *Web Server* component admits (and handles in different ways) three types of requests: GET (used to retrieve *experiences*' meta-data), SSE (used to get server-to-client data updates) and POST (used to send client-to-server updates or client-to-server requests for data updates). These different methods are each associated with the three basic cases of use, namely:

- *Meta-data* - A client, wanting to obtain information about the laboratory, launches an HTTP GET request to the URL associated with the laboratory. The RIP server responds with a JSON-RPC structure that informs the client, depending on the request's parameters, with one of the following:
 1. General information about the OL: what are the *experiences* defined and how they can be accessed.
 2. Detailed information of a particular *experience* (when the request includes the experience id as a parameter).
- *Observer* - A client, that desires to receive updates on the state of the plant, subscribes to an SSE event stream associated to the *experience* of interest.
- *Operator* - A client, wanting to act over the OL or to receive an update on demand, sends a POST request with the command codified as a RIP-JSON-RPC structure.

An *experience* represents a lab activity associated with an OL. In the case of RLs, each *experience* is implemented as a *control program*, which in general is responsible of managing the physical connections with the

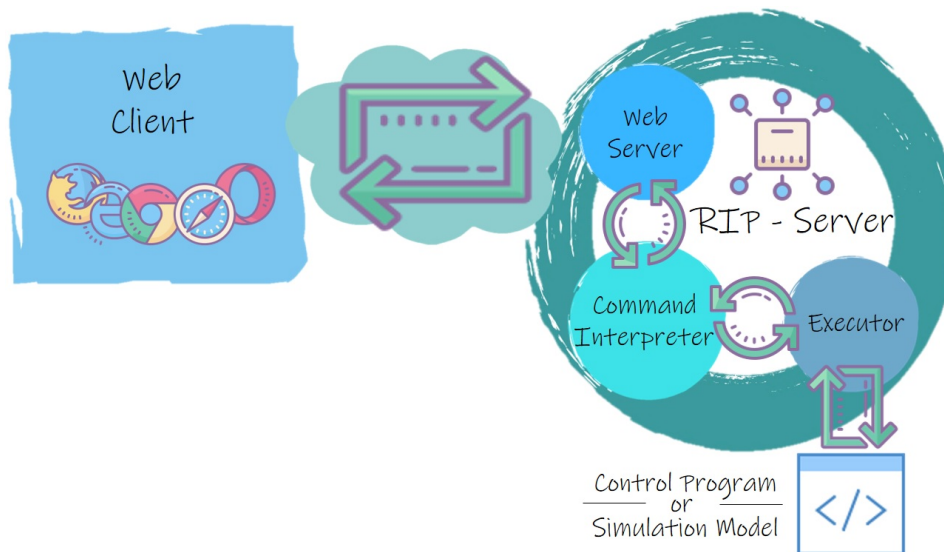


Figure 2.2: Architectural view of a RIP Server

hardware, safe measures, and any other functionality the lab designer has considered appropriate to include. In the case of VLS, each *experience* is implemented as a *simulation model* that represents a real system. The *experience* abstraction is useful for two purposes: 1) to publish information about OLs in a standard and structured way and 2) to allow for hosting and running several different *control programs* or *simulation models* in the same computer.

2.1.2 Client Implementation Perspective

A RIP client must simply implement the required communication protocol methods (that is, POST, GET and SSE) with the appropriate format for reading and writing the messages content (that is, JSON-RPC) and the structure and functions defined by RIP (detailed in later sections).

2.2 Protocol Functions (High-Level API)

The functions defined and used in the protocol are divided in two types: internal and external.

The internal functions are private functions, used internally by RIP server implementations to communicate either with the *simulation model* or the *control program* used in the OL.

The external functions are client-side methods. These functions are used by the RIP clients to both get meta-data about the defined *experiences* and to write and read data to and from the OL.

In this way, RIP servers must implement the internal functions, while RIP clients must implement the external functions (see Figure 2.3). Client-based methods of the external functions in a RIP client communicate with the *Web Server* of a RIP server, get translated by the *Command Interpreter* into a series of one or more internal functions, and are finally executed by the *Executor* component (see Figure 2.2).

2.2.1 Internal Functions

The list of internal functions a RIP server must implement is:

1. *info* = **basicInfo**() : Returns the list of *experiences* defined in the OL and meta-data information about the method that can be used to retrieve more information about a particular *experience*.
2. *info* = **experienceInfo**(*expId*) : Returns the list of variables defined in the *control program* or *simulation model* associated to the *experience* defined by *expId*, meta-data related to such *experience* and information about the methods that can be used to communicate with the OL.
3. *readablelist*, *writablelist* = **open**(*expId*) : Returns the list of readable and writable variables (in two different variables) defined in the *experience* associated to the input *expId* parameter.

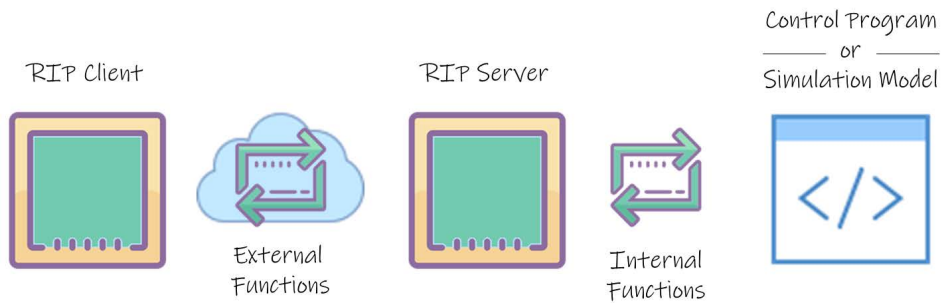


Figure 2.3: Internal and external functions implementation in RIP clients and servers

4. **close**(*expId*): Closes the control program (if it is a RL) or the simulation model (if it is a VL) of the OL *experience* defined by the input *expId* parameter.
5. **start**(*expId*): Starts the execution of the *control program* or *simulation model* associated to the OL *experience* defined by the input *expId* parameter by: first, opening it and second, running it.
6. **run**(*expId*): Runs the *control program* or *simulation model* associated to the OL *experience* defined by the input *expId* parameter.
7. **stop**(*expId*): Stops the execution of the *control program* or *simulation model* associated to the OL *experience* defined by the input *expId* parameter.
8. **readVariableNames**, **readVariableValues** = **get**(*expId*, *variableNames*): Retrieves the current values of the variables (*readVariableValues*) specified by the *variableNames* input parameter. For this to happen, these variables must exist in the *control program* or *simulation model* associated to the *experience* defined by the *expId* input parameter. When the **get**() method is called, *readVariableNames* contains only the names of the variables that were successfully read, not all requested ones in *variableNames*.
9. **set**(*expId*, *variableNames*, *variableValues*): Writes the received values (*variableValues*) in the specified variables (*variableNames*) of the appropriate OL *experience*. For this to happen, these variables must exist in the *control program* or *simulation model* associated to the (*expId*) input parameter.

2.2.2 External Functions

The list of external functions a RIP client must implement is:

1. **result** = **info**(*callback*, *expId* = null): Retrieves the meta-data information about the *experience* defined by the input *expId* parameter. This parameter is optional and if it is not specified, the method then returns meta-data information about all the *experiences* defined.
2. **connect**(*expId*, *callback*): Connects to an *experience* defined by the input *expId* parameter and establishes connection to the SSE to start receiving server data updates.
3. **set**(*variableNames*, *variableValues*, *callback*, *expId*): Writes the received values (*variableValues*) in the specified variables (*variableNames*) of the appropriate OL *experience*. For this to happen, these variables must exist in the *control program* or *simulation model* associated to the (*expId*) input parameter.
4. **result** = **get**(*variableNames*, *callback*, *expId*): Retrieves the current values of the variables (*readVariableValues*) specified by the *variableNames* input parameter. For this to happen, these variables must exist in the *control program* or *simulation model* associated to the *experience* defined by the *expId* input parameter.

Where:

Variables *variableNames* and *variableValues* are arrays of text. For example: *variableNames* = ["x", "y", ...], *variableValues* = ["10", "a", ...].

At the moment, only numbers, text and booleans are supported as valid types for *variableValues* and *readVariableValues* in the **set**() and **get**() methods, respectively.

Sections 2.7 and 2.8 give more information about the external and internal functions, respectively.

2.3 Protocol Communication Methods (Low-Level API)

Three HTTP communication methods are available in RIP for communicating the client with the server:

1. **GET** - To obtain OL meta-data. A RIP server must implement a web service endpoint at `BaseURL:port/RIP` for attending these requests.

RIP client function (High-Level API)	HTTP Method (Low-Level API)	Web server endpoint
info(callback), info(callback, expId)	GET	/BaseURL:port/RIP
set(variableNames, variableValues, callback, expId), get(variableNames, expId)	POST	/BaseURL:port/RIP/POST
connect(expId, callback)	SSE	/BaseURL:port/RIP/SSE

Table 2.1: Correspondence between RIP external functions, HTTP methods and RIP’s web server endpoints

2. **POST** - To send client-to-server requests for (i) writing OL variables’ values and (ii) reading OL variables’ values. A RIP server must implement a web service endpoint at `BaseURL:port/RIP/POST` for attending these requests.
3. **SSE** - To subscribe the client to data streams so that it receives server-to-client OL variables’ values updates. A RIP server must implement a web service endpoint at `BaseURL:port/RIP/SSE` for attending these requests.

Table 2.1 shows the correspondence between the external protocol functions of the high-level API and the HTTP methods of the low-level API. The right column also indicates the RIP’s web server endpoint to which each of the methods and functions communicate with. A representation of these same ideas is shown in Figure 2.4. While they should, not all internal functions are represented in the Figure; only six out of nine appear in it.

Section 2.8 gives more information about the use of these HTTP methods in RIP’s low-level API.

2.4 Operating Environment

RIP uses only HTTP methods for the communication. Therefore, it works in any major web browser. However, RIP also relies on the use of SSEs, which, up to date, are not supported by Microsoft Internet Explorer nor Microsoft Edge. Nevertheless, there are numerous poly-fill solutions for implementing SSE so that they work on these browsers that do not support them natively.

2.5 Design and Implementation Constraints

RIP is designed to only use pure HTTP methods on purpose, with the aim of guaranteeing its correct functioning from web browsers. Therefore, the only hard implementation constraint is that HTTP is used for implementing RIP communications.

2.6 Assumptions and Dependencies

RIP depends solely on the use of HTTP POST and GET methods and on the JSON format for exchanging data. A common and easy way of implementing RIP in web clients is through the use of the EventSource object [3] (for the SSEs) and the XMLHttpRequest object [5] or the Fetch API [6] (for the POST messages), which are both supported by all major web browsers. Still, implementations without the use of such APIs are possible, just more laborious.

RIP server implementations may differ a lot depending on the language used to make the implementation, and so do their possible dependencies.

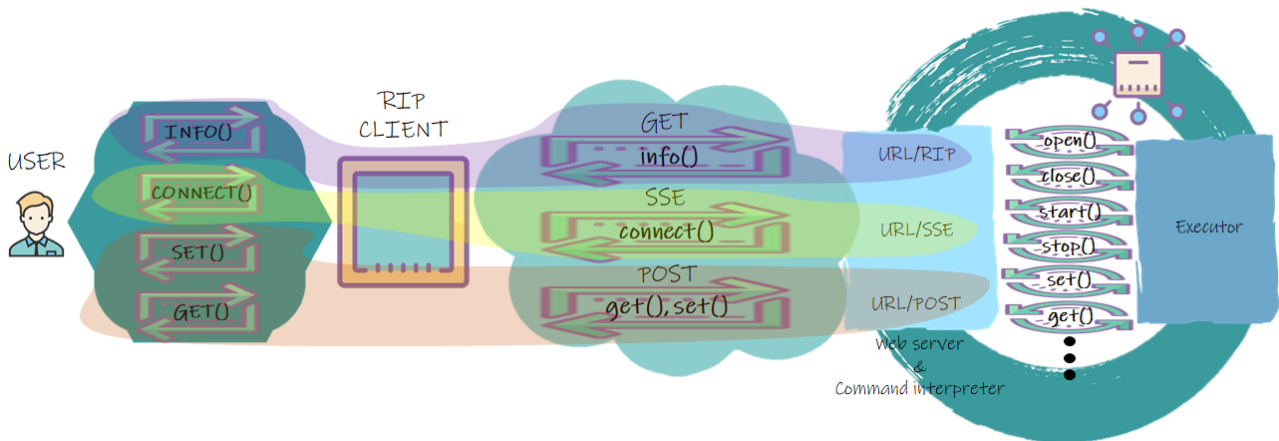


Figure 2.4: Illustration of the existing correspondence between RIP functions, HTTP methods and RIP's web server endpoints

2.7 User Documentation

This section provides a deeper insight on the **external functions of the high-level API** from a final user point of view. This is the only knowledge required by such a user to use a RIP client implementation to communicate with an OL running a RIP server.

A RIP client must implement a class with the methods described in Section 2.2.2. Next, detailed information of each of them is given. This information includes a description of: 1) the method itself, 2) its input parameters and 3) the result returned by the method. All methods have a *callback* parameter; a function to be called after the method completes its tasks.

2.7.1 Method info

This method has two input parameters: *callback* and *expId*. It can be called with or without the *expId* parameter.

When called without this parameter, `info()` returns meta-data with the list of experience identifiers associated to all *experiences* defined in the OL. It also returns more information about the low-level API method call for retrieving meta-data.

When called with the *expId* parameter, `info(expId)` returns meta-data of the referenced *experience*.

The result returned by this method is a string in JSON format containing information about the experience defined in the RIP server with name *expId*. The structure and content of this string depends on whether the method is called with or without the *expId* parameter. In any case, the result is described in detail in section 2.8.2.1.

The way to call it is:

```
result = info(callback, expId = null)
```

2.7.2 Method connect

This method has two input parameters: *expId* and *callback*.

When this method is called, the client connects to an SSE in the RIP server to: 1) control the user session in the server and 2) receive updates of the variables value in the *experience* defined by *expId*.

Although updates on the server's variables values are received upon the connection, this method returns nothing by itself. The way to call it is:

connect(*expId*, *callback*)

2.7.3 Method set

This method has four input parameters: *variableNames* and *variableValues*, *callback* and *expId*, .

When this method is called, the values specified in *variableValues* are assigned to the *control program's* or *simulation model's* variables named as the strings in *variableNames*. The *control program* or *simulation model* is determined with the *expId* parameter.

This method returns nothing and the way to call it is: **set**(*variableNames*, *variableValues*, *callback*, *expId*)

2.7.4 Method get

This method has three input parameters: *variableNames*, *callback* and *expId*.

When this method is called, the values of the *control program's* or *simulation model's* variables named as the strings in *variableNames* are read and returned. The *control program* or *simulation model* is determined with the *expId* parameter.

The result returned by this method is a mixed array with the current values of the *control program's* or *simulation model's* variables specified in the *variableNames* input parameter.

The way to call it is:

result = **get**(*variableNames*, *callback*, *expId*)

2.8 Developer Documentation

This section provides more details on the internal functions of the high-level API and on the low-level API, which represents the knowledge required to either implement a new RIP client or a new RIP server, or to modify existing ones.

2.8.1 Internal Functions of the High-Level API

The High-Level API internal functions are run in the RIP server upon the reception of a GET or POST message, or when an SSE connection is established, mostly. Table 2.2 shows the correspondence between the external function called by the client and the internal function executed at the server.


RIP (client) external function	RIP (server) internal functions
info(callback)	basicInfo()
info(callback, expId)	experienceInfo()
connect(expId, callback)	1. - start(), 2. - get() 
set(variableNames, variableValues, callback, expId)	set()
get(variableNames, callback, expId)	get()

Table 2.2: Correspondence between RIP external functions and RIP internal functions

However, the execution of some internal functions are not fired due to the reception of an external function in the RIP server. On the contrary, some internal functions are run when a client disconnection is detected (that's the case of the *stop()* and *close()* methods), or when other internal functions are called (*open()* and *run()* methods). Table summarizes this information.

Functions between brackets in Table 2.3 are not necessarily run, and only get executed when needed.

As can be seen in Tables 2.2 and 2.3, the *get()* internal function can be either run upon a client function reception (*connect(expId)* and *get(expId, variableNames)*) or due to the execution of the *experienceInfo()* internal method. When the client invokes the *connect(expId)* external function, the RIP server runs the *get()* internal function in a loop to get, and then send, constant updates to the client through the established SSE connection.

Each method returns an error indicator when the operation is not completed successfully.

2.8.1.1 Method basicInfo

This method is called only when the RIP server receives an HTTP message from a client with the *info* method and no parameters.

It returns a list of *experiences* defined in the OL and meta-data information about the GET HTTP RIP method described in previous sections to get meta-data.

2.8.1.2 Method experienceInfo

This method is called only when the RIP server receives an HTTP message from a client with the *info* method and the *expId* parameter.

It returns meta-data information about the *experience* and the available HTTP RIP methods described in previous sections to communicate with the OL.

2.8.1.3 Method start

This method is called only when the RIP server receives an HTTP message from a client with the *connect* method and the *expId* parameter.

It runs (when needed) two internal functions in order: first, *open()* and second, *run()*, to open and run the *control program* or *simulation model* associated to the experience, respectively.

2.8.1.4 Method open

This method is run only when either the *experienceInfo()* or the *start()* methods are called in the RIP server and there are no other clients using the associated *experience*, which means the corresponding *control program* or *simulation model* is closed.

Trigger condition	Executed RIP (server) internal functions
<i>experienceInfo()</i> is run	1. - [<i>open()</i>], 2. - <i>get()</i> , 3. - [<i>close()</i>]
<i>start()</i> is run	1. - [<i>open()</i>], 2. - [<i>run()</i>]
Client disconnects	1. - [<i>stop()</i>], 2. - [<i>close()</i>]

Table 2.3: Triggering conditions and fired internal methods

It opens the *control program* or *simulation model* associated to the experience.

2.8.1.5 Method close

This method is run either when a client disconnection is produced, with the objective of closing the *control program* or *simulations model* that was open to attend the client requests, or when the `experienceInfo()` method is called and there are no other clients using the associated *experience*.

It closes the *control program* or *simulation model* associated to the experience.

2.8.1.6 Method run

This method is run only when either the `start()` method is called in the RIP server and there are no other clients using the associated *experience*, which means the corresponding *control program* or *simulation model* is not currently running.

It runs the *control program* or *simulation model* associated to the experience.

2.8.1.7 Method stop

This method is only run when a client disconnection is produced, with the objective of stopping the execution of the *control program*, or *simulations model* that was running to attend the client requests.

It stops the *control program* or *simulation model* associated to the experience.

2.8.1.8 Method get

This method is called only when the RIP server receives an HTTP message from a client either with the *get* method and the *expId* and *variableNames* parameters or with the *connect* method and the *expId* parameter. In the first case, this method is called only once. In the second, as an SSE communication is established, the method is called receptively, depending on the events defined in the RIP server.

It returns the list of current values of the variables defined in the *control program* or simulation model and the names of the read variables.

2.8.1.9 Method set

This method is called only when the RIP server receives an HTTP message from a client with the *set* method and the *expId*, *variableNames* and *variableValues* parameters.

It writes the values received in the specified variables of the appropriate OL *experience*.

2.8.2 Low-Level API

Each external method from the High-Level API is implemented with an HTTP method that follows the Low-Level API. The nature and structure of the HTTP messages that must be built by each High-Level API method, the RIP server endpoint to which it must connect, and the result returned by the RIP server to each of these messages is detailed in this section.

For the sake of clarity, each method also includes an example. The relevant parameters for this example can be found in Table 2.4.

Parameter	Value
Protocol	http
BaseURL	10.192.38.56
Communication port	8080
Declared experiences	Test1, Test2
Declared inputs in <i>control program</i> or <i>simulation model</i> for Test1	stringin, intin, doublein, booleanin
Declared outputs in <i>control program</i> or <i>simulation model</i> for Test1	stringout, intout, doubleout, booleanout

Table 2.4: Relevant parameters for the examples.

2.8.2.1 Method info - GET

The *info(callback)* RIP client method sends a GET request to `http(s)://BaseURL:port/RIP`, while *info(callback, expIdValue)* does the same with `http(s)://BaseURL:port/RIP?expId=expIdValue`.

The result of calling this method without the *expId* input parameter is a JSON object containing one single top-level member:

- (1) **experiences**: A JSON object with the following top-level members:
 - (I) **list**: An array of JSON objects, each of which contains one single top-level member:
 - (A) **id**: A string with the experience identifier (*expId*) that is unmistakably related to one and only one *experience* defined in the OL.
 - (II) **methods**: An array of JSON objects, each of which contains the following top-level members:
 - (A) **url**: A string with the URL to call the method.
 - (B) **type**: A string specifying the HTTP method to use.
 - (C) **description**: A string describing the method.
 - (D) **params**: An array of JSON objects detailing the required and optional parameters for the method:
 - (i) **name**: A string with the name of the parameter
 - (ii) **required**: A string ("yes" or "no") specifying whether the parameter is required ("yes") or optional ("no").
 - (iii) **location**: A string referencing the location for the parameter ("query", "header" or "body").
 - (iv) **value**: [Optional] A string with the required value for the parameter.
 - (v) **type**: [Optional] A string detailing the type of the parameter.
 - (vi) **elements**: [Optional] An array of JSON objects with the following top-level members:
 - (a) **description**: A string with a description of the parameter element.
 - (b) **type**: A string specifying the type of the parameter element.
 - (c) **subtype**: [Optional] A string detailing the type of the elements inside an array parameter element when they are all of the same type.
 - (vii) **subtype**: [Optional] A string detailing the type of the elements inside an array parameter when they are all of the same type.
 - (E) **returns**: A string with the MIME-type of the method's result.
 - (F) **example**: A JSON object with the following top-level members:

- (i) **url**: A string with the URL to call the method.
- (ii) **headers**: [Optional] A JSON object with the parameters to be placed on the HTTP request headers with as many top-level members as headers are set:
 - (a) **HeaderName**: HeaderValue
- (iii) **body**: [Optional] A JSON object with the parameters to be placed on the HTTP request body and with the following top-level members:
 - (a) **jsonrpc**: "2.0"
 - (b) **method**: A string with the RIP method to invoke ("get", "set" or "connect").
 - (c) **params**: An array with the following elements:
 - (1) **expId**: A string with the *experience* identifier.
 - (2) **variableNames**: An array of strings in which each element has the name of a variable in the *control program* or in the *simulation model*.
 - (3) **variableValues**: [Optional] An array of strings with the values to be written in the variables specified in the *variableNames* member, following the same order.
 - (d) **id**: A string with a number that indicates how many request have been sent till now, including this one.

Example:

Request:

The *info(callback)* RIP client method sends a GET request to: `http://10.192.38.56:8080/RIP`

Response:

The RIP server responds to the GET message with the following body:

```

1  {
2    "experiences": {
3      "list": [
4        {
5          "id": "Test1"
6        },
7        {
8          "id": "Test2"
9        }
10     ],
11     "methods": [
12       {
13         "url": "10.192.38.56:8080/RIP",
14         "type": "GET",
15         "description": "Retrieves information (variables and methods) of the experiences in the
16                       server",
17         "params": [
18           {
19             "name": "Accept",
20             "required": "no",
21             "location": "header",
22             "value": "application/json"
23           },
24           {
25             "name": "expId",
26             "required": "no",
27             "location": "query",
28             "type": "string"
29           }
30         ],
31         "returns": "application/json",
32         "example": {

```

```

32         "url": "10.192.38.56:8080/RIP?expId=Test1"
33     }
34 }
35 ]
36 }
37 }

```

The result of calling this method with the *expId* input parameter is a JSON object containing three top-level members:

- (1) **info**: A JSON object with the following top-level members:
 - (I) **name**: A string with the name given to the experience.
 - (II) **description**: A string with a description of the experience.
 - (III) **authors**: A string with the name of the authors of the experience, separated by commas.
 - (IV) **keywords**: An array of strings, each with a keyword related to the experience.
- (2) **readables**: A JSON object with the following top-level members:
 - (I) **list**: An array of JSON objects with the following top-level members:
 - (A) **name**: A string with the name of the variable.
 - (B) **description**: A string with a description of the variable.
 - (C) **type**: A string with the type of the variable: *"string"*, *"int"*, *"float"* or *"boolean"*.
 - (D) **min**: A string with the minimum value the variable can have. In case of boolean variables, *"false"*; in case of string variables, *" "*; in any other case, any number in string format.
 - (E) **max**: A string with the minimum value the variable can have. In case of boolean variables, *"true"*; in case of string variables, *" "*; in any other case, any number in string format.
 - (F) **precision**: A string with the minimum value the variable can have. In case of boolean or string variables, *" "*; in any other case, any number in string format.
 - (II) **methods**: EXACTLY AS the *methods* top-level member from the *experiences* JSON object returned by the *info(callback)* method when called without the *expId* parameter: an array of JSON objects, each of which contains the following top-level members:
 - (A) **url**: A string with the URL to call the method.
 - (B) **type**: A string specifying the HTTP method to use.
 - (C) **description**: A string describing the method.
 - (D) **params**: An array of JSON objects detailing the required and optional parameters for the method:
 - (i) **name**: A string with the name of the parameter
 - (ii) **required**: A string ("yes" or "no") specifying whether the parameter is required ("yes") or optional ("no").
 - (iii) **location**: A string referencing the location for the parameter ("header", "query" or "body").
 - (iv) **value**: [Optional] A string with the required value for the parameter.
 - (v) **type**: [Optional] A string detailing the type of the parameter.
 - (vi) **elements**: [Optional] An array of JSON objects with the following top-level members:
 - (a) **description**: A string with a description of the parameter element.
 - (b) **type**: A string specifying the type of the parameter element.
 - (c) **subtype**: [Optional] A string detailing the type of the elements inside an array parameter element when they are all of the same type.
 - (vii) **subtype**: [Optional] A string detailing the type of the elements inside an array parameter when they are all of the same type.
 - (E) **returns**: A string with the MIME-type of the method's result.
 - (F) **example**: A JSON object with the following top-level members:

- (i) **url**: A string with the URL to call the method.
 - (ii) **headers**: [Optional] A JSON object with the parameters to be placed on the HTTP request headers with as many top-level members as headers are set:
 - (a) **HeaderName**: HeaderValue
 - (iii) **body**: [Optional] A JSON object with the parameters to be placed on the HTTP request body and with the following top-level members:
 - (a) **jsonrpc**: "2.0"
 - (b) **method**: A string with the RIP method to invoke ("get", "set" or "connect").
 - (c) **params**: An array with the following elements:
 - (1) **expId**: A string with the *experience* identifier.
 - (2) **variableNames**: An array of strings in which each element has the name of a variable in the *control program* or in the *simulation model*.
 - (3) **variableValues**: [Optional] An array of strings with the values to be written in the variables specified in the *variableNames* member, following the same order.
 - (d) **id**: A string with a number that indicates how many request have been sent till now, including this one.
- (3) **writables**: A JSON object with the following top-level members:
- (I) **list**: EXACTLY AS the *list* top-level member from *readables*: an array of JSON objects with the following top-level members:
 - (A) **name**: A string with the name of the variable.
 - (B) **description**: A string with a description of the variable.
 - (C) **type**: A string with the type of the variable: "string", "int", "float" or "boolean".
 - (D) **min**: A string with the minimum value the variable can have. In case of boolean variables, "false"; in case of string variables, ""; in any other case, any number in string format.
 - (E) **max**: A string with the minimum value the variable can have. In case of boolean variables, "true"; in case of string variables, ""; in any other case, any number in string format.
 - (F) **precision**: A string with the minimum value the variable can have. In case of boolean or string variables, ""; in any other case, any number in string format.
 - (II) **methods**: EXACTLY AS the *methods* top-level member from the *experiences* JSON object returned by the *info(callback)* method when called without the *expId* parameter: an array of JSON objects, each of which contains the following top-level members:
 - (A) **url**: A string with the URL to call the method.
 - (B) **type**: A string specifying the HTTP method to use.
 - (C) **description**: A string describing the method.
 - (D) **params**: An array of JSON objects detailing the required and optional parameters for the method:
 - (i) **name**: A string with the name of the parameter
 - (ii) **required**: A string ("yes" or "no") specifying whether the parameter is required ("yes") or optional ("no").
 - (iii) **location**: A string referencing the location for the parameter ("header", "query" or "body").
 - (iv) **value**: [Optional] A string with the required value for the parameter.
 - (v) **type**: [Optional] A string detailing the type of the parameter.
 - (vi) **elements**: [Optional] An array of JSON objects with the following top-level members:
 - (a) **description**: A string with a description of the parameter element.
 - (b) **type**: A string specifying the type of the parameter element.
 - (c) **subtype**: [Optional] A string detailing the type of the elements inside an array parameter element when they are all of the same type.
 - (vii) **subtype**: [Optional] A string detailing the type of the elements inside an array parameter when they are all of the same type.

- (E) **returns:** A string with the MIME-type of the method's result.
- (F) **example:** A JSON object with the following top-level members:
 - (i) **url:** A string with the URL to call the method.
 - (ii) **headers:**[Optional] A JSON object with the parameters to be placed on the HTTP request headers with as many top-level members as headers are set:
 - (a) **HeaderName:** HeaderValue
 - (iii) **body:** [Optional] A JSON object with the parameters to be placed on the HTTP request body and with the following top-level members:
 - (a) **jsonrpc:** "2.0"
 - (b) **method:** A string with the RIP method to invoke ("get", "set" or "connect").
 - (c) **params:** An array with the following elements:
 - (1) **expId:** A string with the *experience* identifier.
 - (2) **variableNames:** An array of strings in which each element has the name of a variable in the *control program* or in the *simulation model*.
 - (3) **variableValues:** [Optional] An array of strings with the values to be written in the variables specified in the *variableNames* member, following the same order.
 - (d) **id:** A string with a number that indicates how many request have been sent till now, including this one.

Example:

Request:

The *info(callback, expId)* RIP client method sends a GET request to: `http://10.192.38.56:8080/RIP?expId=Test1`

Response:

The RIP server responds to the GET message with the following body:

```

1 {
2   "info": {
3     "name": "Test1",
4     "description": "Test1",
5     "authors": "L. de la Torre",
6     "keywords": ["Test", "Example"]
7   },
8   "readables": {
9     "list": [
10      {
11        "name": "intout",
12        "description": "Integer output",
13        "type": "int",
14        "min": "-20",
15        "max": "10",
16        "precision": "1"
17      },
18      {
19        "name": "stringout",
20        "description": "String output",
21        "type": "string",
22        "min": "",
23        "max": "",
24        "precision": ""
25      },
26      {
27        "name": "booleanout",
28        "description": "Boolean output",
29        "type": "boolean",

```

```

30         "min": "false",
31         "max": "true",
32         "precision": ""
33     },
34     {
35         "name": "doubleout",
36         "description": "Double output",
37         "type": "float",
38         "min": "-Inf",
39         "max": "Inf",
40         "precision": "0"
41     }
42 ],
43 "methods": [
44     {
45         "url": "10.192.38.56:8080/RIP/SSE",
46         "type": "GET",
47         "description": "Subscribes to an SSE to get regular updates on the servers' variables",
48         "params": [
49             {
50                 "name": "Accept",
51                 "required": "no",
52                 "location": "header",
53                 "value": "application/json"
54             },
55             {
56                 "name": "expId",
57                 "required": "yes",
58                 "location": "query",
59                 "type": "string"
60             },
61             {
62                 "name": "variables",
63                 "required": "no",
64                 "location": "query",
65                 "type": "array",
66                 "subtype": "string"
67             }
68         ],
69         "returns": "text/event-stream",
70         "example": "10.192.38.56:8080/RIP/SSE?expId=TestOK"
71     },
72     {
73         "url": "10.192.38.56:8080/RIP/POST",
74         "type": "POST",
75         "description": "Sends a request to retrieve the value of one or more servers' variables
76                        on demand",
77         "params": [
78             {
79                 "name": "Accept",
80                 "required": "no",
81                 "location": "header",
82                 "value": "application/json"
83             },
84             {
85                 "name": "Content-Type",
86                 "required": "yes",
87                 "location": "header",
88                 "value": "application/json"
89             },
90             {

```

```

90         "name": "jsonrpc",
91         "required": "yes",
92         "type": "string",
93         "location": "body",
94         "value": "2.0"
95     },
96     {
97         "name": "method",
98         "required": "yes",
99         "type": "string",
100        "location": "body",
101        "value": "get"
102    },
103    {
104        "name": "params",
105        "required": "yes",
106        "type": "array",
107        "elements": [
108            {
109                "description": "Experience id",
110                "type": "string"
111            },
112            {
113                "description": "Name of variables to be retrieved",
114                "type": "array",
115                "subtype": "string"
116            }
117        ],
118        "location": "body"
119    },
120    {
121        "name": "id",
122        "required": "yes",
123        "type": "int",
124        "location": "body"
125    }
126 ],
127 "returns": "application/json",
128 "example": {
129     "url": "10.192.38.56:8080/RIP/POST",
130     "headers": {
131         "Accept": "application/json",
132         "Content-Type": "application/json"
133     },
134     "body": {
135         "jsonrpc": "2.0",
136         "method": "get",
137         "params": [
138             "Test1",
139             ["intout", "booleanout"]
140         ],
141         "id": "1"
142     }
143 },
144 {
145     "url": "http://camera1_ip/axis-cgi/mjpg/video.cgi",
146     "type": "GET",
147     "description": "Retrieve an image of the lab captured from the camera: 'Camera 1'",
148     "params": [
149         {

```

```

151         "name": "resolution",
152         "required": "no",
153         "location": "query",
154         "type": "string"
155     },
156     {
157         "name": "user",
158         "required": "no",
159         "location": "query",
160         "type": "string"
161     },
162     {
163         "name": "password",
164         "required": "no",
165         "location": "query",
166         "type": "string"
167     }
168 ],
169 "returns": "video/x-motion-jpeg",
170 "example": {
171     "url": "http://camera1_ip/axis-cgi/mjpg/video.cgi"
172 }
173 ]
174 },
175 "writables": {
176     "list": [
177         {
178             "name": "intin",
179             "description": "Integer input",
180             "type": "int",
181             "min": "-20",
182             "max": "10",
183             "precision": "1"
184         },
185         {
186             "name": "booleanin",
187             "description": "Boolean input",
188             "type": "boolean",
189             "min": "false",
190             "max": "true",
191             "precision": ""
192         },
193         { "name": "stringin",
194           "description": "String input",
195           "type": "string",
196           "min": "",
197           "max": "",
198           "precision": ""
199         },
200         {
201             "name": "doublein",
202             "description": "Double input",
203             "type": "float",
204             "min": "-Inf",
205             "max": "Inf",
206             "precision": "0"
207         }
208     ],
209     "methods": [
210         {
211             "url": "10.192.38.56:8080/RIP/POST",

```

```

212     "type": "POST",
213     "description": "Sends a request to write the value of one or more servers' variables on
        demand",
214     "params": [
215         {
216             "name": "Accept",
217             "required": "no",
218             "location": "header",
219             "value": "application/json"
220         },
221         {
222             "name": "Content-Type",
223             "required": "yes",
224             "location": "header",
225             "value": "application/json"
226         },
227         {
228             "name": "jsonrpc",
229             "required": "yes",
230             "type": "string",
231             "location": "body",
232             "value": "2.0"
233         },
234         {
235             "name": "method",
236             "required": "yes",
237             "type": "string",
238             "location": "body",
239             "value": "set"
240         },
241         {
242             "name": "params",
243             "required": "yes",
244             "type": "array",
245             "elements": [
246                 {
247                     "description": "Experience id",
248                     "type": "string"
249                 },
250                 {
251                     "description": "Name of variables to write",
252                     "type": "array",
253                     "subtype": "string"
254                 },
255                 {
256                     "description": "Value for variables",
257                     "type": "array",
258                     "subtype": "mixed"
259                 }
260             ],
261             "location": "body"
262         },
263         {
264             "name": "id",
265             "required": "yes",
266             "type": "int",
267             "location": "body"
268         }
269     ],
270     "returns": "application/json",
271     "example": {

```



```

272         "url": "10.192.38.56:8080/RIP/POST",
273         "headers": {
274             "Accept": "application/json",
275             "Content-Type": "application/json"
276         },
277         "body": {
278             "jsonrpc": "2.0",
279             "method": "set",
280             "params": [
281                 "Test1",
282                 ["intin", "stringin"],
283                 ["2", "hello"]
284             ],
285             "id": "1"
286         }
287     }
288 ]
289 }
290 }
291 }

```

2.8.2.2 Method connect - SSE

The *connect(expIdValue, callback)* RIP client method establishes an SSE connection with: `http(s)://BaseURL:port/RIP/SSE?expId=expIdValue`.

When this method is called, an SSE communication is established with the RIP server. The result is a stream of data (which periodicity or aperiodicity is defined in the server) that can be captured by the *onmessage()* and the *addEventListener()* methods of the *EventSource* object, for example [3]. The structure of the data obtained through the established SSE follows the SSE standard:

- (1) **retry**: A number specifying the time (in milliseconds) to wait before the client tries to automatically reconnect to the SSE. This field appear only once per connection.
- (2) **event**: The name of the event that follows
- (3) **id**: A unique number that identifies the event.
- (4) **data**: A JSON object with just one top-level members:
 - (I) **result**: An array with the following elements:
 - (A) **variableNames**: An array of strings with the names of the retrieved variables.
 - (B) **variableValues**: A mixed array with the values of the retrieved variables.

Example:

Request:

The *connect("Test1")* RIP client method establishes an SSE connection with: `http://10.192.38.56:8080/RIP/SSE?expId=Test1`.

Response:

The RIP server responds with a stream of data that follows this format:

```

1  retry: 2000
2
3  event: periodiclabdata
4  id: 200
5  data:

```

```

6 {
7   "result": [
8     [
9       "intout",
10      "doubleout",
11      "stringout",
12      "booleanout"
13     ],
14     [
15       -2,
16       3.5,
17       "testing",
18       true
19     ]
20   ]
21 }

```

Where *event* defines the name of the event stream the client can subscribe to and *id* is a unique identifier for that particular *event*. In RIP, the *id* is set to be the time in milliseconds since the client first connected to the server.

A RIP client implementation must automatically capture these events and assign the received values to their corresponding variables in the client side.

2.8.2.3 Method set - POST

The *set(variableNames, variableValues, callback, expIdValue)* RIP client method sends a POST request to: `http(s)://BaseURL:port/RIP/POST?expId=expIdValue` with a JSON body with the following top-level members:

- (1) **jsonrpc**: "2.0"
- (2) **method**: "set".
- (3) **params**: An array with the following elements:
 - (I) **expId**: A string with the *experience* identifier.
 - (II) **variableNames**: An array of strings with the name of the variables in the *control program* or *simulation model* whose values are going to be overwritten.
 - (III) **variableValues**: A mixed array with the values to write in the *control program's* or *simulation model's* variables.
- (4) **id**: A string with a number that indicates how many request have been sent till now, including this one.

This method returns a JSON object with information about the result of the operation. The resulting JSON object contains the following top-level members:

- (1) **jsonrpc**: "2.0"
- (2) **result**: Either *true*, if the operation was completed successfully by the RIP server, or *false*, if it was not.
- (3) **id**: A string with a number that indicates how many request have been sent till now, including this one.

Example:

Request:

The *set("Test1", ["doublein", "intin"], [0.5, -1])* RIP client method sends a POST request to: `http(s)://BaseURL:port/RIP/POST?expId=Test1` with body:

```

1 {
2   "jsonrpc": "2.0",
3   "method": "set",
4   "params": [
5     "Test1",
6     ["doublein", "intin"],
7     [0.5, -1]
8   ],
9   "id": "2"
10 }

```

Response:

The RIP server responds to the POST message with the following body:

```

1 {
2   "jsonrpc": "2.0",
3   "result": true,
4   "id": "2"
5 }

```

2.8.2.4 Method get - POST

The *set(variableNames, callback, expIdValue)* RIP client method sends a POST request to: `http(s)://BaseURL:port/RIP/POST?expId=expIdValue` with a JSON body with the following top-level members:

- (1) **jsonrpc**: "2.0"
- (2) **method**: "get".
- (3) **params**: An array with the following elements:
 - (I) **expId**: A string with the *experience* identifier.
 - (II) **variableNames**: An array of strings with the name of the variables in the *control program* or *simulation model* whose values are going to be retrieved.
- (4) **id**: A string with a number that indicates how many request have been sent till now, including this one.

This method returns a JSON object that contains the following top-level members:

- (1) **jsonrpc**: "2.0"
- (2) **result**: An array with the following elements:
 - (I) **variableNames**: An array of strings with the names of the retrieved variables.
 - (II) **variableValues**: A mixed array with the values of the retrieved variables.
- (3) **id**: A string with a number that indicates how many request have been sent till now, including this one.

Example:

Request:

The *get("Test1", ["doubleout", "intout"])* RIP client method sends a POST request to: `http(s)://BaseURL:port/RIP/POST?expId=Test1` with body:

```
1 {
2   "jsonrpc": "2.0",
3   "method": "get",
4   "params": [
5     "Test1",
6     ["doubleout", "intout"]
7   ],
8   "id": "3"
9 }
```

Response:

The RIP server responds to the POST message with the following body:

```
1 {
2   "jsonrpc": "2.0",
3   "result": [
4     [
5       "doubleout",
6       "intout"
7     ],
8     [
9       0.5,
10      -1
11     ]
12   ],
13   "id": "3"
14 }
```

Chapter 3

Protocol Features

This chapter describes the main features of RIP. Some features are client-based while others are server-based. This is specified every time a new features is presented.

3.1 Defining Experiences and Meta-data

This is a server-based feature. RIP supports the definition of different experiences within a server. Experiences can be all associated to just one OL or to different ones. An experience gets defined by the next required data:

- **Control program or simulation model:** Each OL experience requires the *control program* or *simulation model* that handles the connection to the hardware, in case of a RL, or that computes the mathematical model of the system, in case of a VL.
- **Path to control program or simulation model:** All experiences must define the path to the *control program* or *simulation model* within the machine that hosts the RIP server.
- **Experience identifier (expId):** The experience id, or *expId*, unequivocally identifies a particular experience defined in the RIP server.

Additionally, the next optional meta-data can also be provided for each experience:

- **Authors:** Experiences may include a list of authors. When not specified, this information is sent as an empty string.
- **Keywords:** Experiences may include a list of related terms or keywords. When not specified, this information is sent as an empty string.
- **Description:** Experiences may include a description that gives more information about the experience objectives, possible experimental tasks and so on. When not specified, this information is sent as an empty string.
- **Cameras:** Experiences may include a list of accessible URLs that stream the video grabbed by the associated cameras.

3.2 Defining Readable and Writable Variables

This is a server-based feature. The RIP server implementation has a list of fully defined readable and writable variables from the control program or simulation model associated to each defined experience. In this context, we understand a variable is fully defined when it provides the following information:

- **Name:** The name of the variable.
- **Nature:** Whether it is a readable or a writable variable or both.
- **Description:** A brief description about the variable purpose/meaning.
- **Type:** Whether the variable is a boolean, a number, a string or an array.
- **Min value:** The minimum value the variable can take. In case of boolean variables, *"false"*; in case of string variables, *" "*; in any other case, any number in string format.
- **Max value:** The maximum value the variable can take. In case of boolean variables, *"true"*; in case of string variables, *" "*; in any other case, any number in string format.
- **Precision:** The minimum increase/decrease in which the values of the variable can be modified. In case of boolean or string variables, *" "*; in any other case, any number in string format.

Ideally, and if the control programs/simulation models allow it, the RIP server implementation will build the list of fully defined variables automatically. This may depend on the the features offered by the control programs and simulation models. If this list cannot be built automatically by the RIP server, then it must provide a way to allow a human user to create such list for each control program or simulation model that requires it.

3.3 Obtaining Meta-data

This is a client-based feature. There are two levels of meta-data a client can obtain from a RIP server:

- **General:** The client retrieves information about the experiences defined in the RIP server and the available method to get more information about them. This was presented in Section 2.8.2.1, when the GET method is called without parameters.
- **Related to an experience:** The client retrieves information about the input and output variables (defined in the "Defining Readable and Writable Variables" server-based feature). It also gets the meta-data associated to that experience (defined in the "Defining Experiences and Meta-data" server-based feature). Finally the client also obtains information about the built-in interface methods, provided by RIP, it can call to communicate with the OL. This was presented in Section 2.8.2.1, when the GET method is called with the *expId* parameter.

3.4 Reading Readable Variables and Writing Writable Variables

This is a client-based feature. It allows a client to use RIP's built-in interface methods to read and write variables from/to the control programs and simulation models. It was presented in Sections 2.8.2.3 (for writable variables) and 2.8.2.2 (for readable ones).

3.5 Concurrent Multi-user Connection

This is a client-based feature. RIP communication methods (POST and SSE) allow multiple clients to send and receive data from the OLs at the same time. The server does not store any state about the client session on the server side that could prevent on a new user to send and/or receive data.

In particular, any number of clients can connect to the SSE to receive updates on an OL's variables simultaneously. POST messages received from clients wanting to write a value in a variable are processed in a first-in, first-out basis.

Any session control, aside from detecting client connections and disconnections to and from the SSE, must be done outside RIP.

3.6 Defining and Subscribing to Server Events

The communication protocol needs to consider two equally important aspects: the definition of the triggering conditions that fire an event in the SSE in the server and the subscription to the produced SSE event streams. Indeed, a certain client user may be interested in subscribing only to a subset of the event streams produced by one or more already defined event triggers, a second client user could need to subscribe to a different subset to carry out its operations, and a third client user may need to define new event triggers to properly work.

The global communication workflow (see Figure 3.1) is as follows:

1. The client application sends a POST request to ask the server about the already defined event triggers (which can may have been defined either by the communication protocol implementation or by the plant owner/expert, as it is described in the next subsection).
2. The server answers with the information, containing all relevant data associated to each defined rule: name, author, description and a list of the parameters that need to be set.
3. The client user decides whether to define a new event trigger or not.
4. The client user decides which event streams to subscribe to, including those produced by event triggers defined by the RIP communication protocol implementation, the plant owner/expert and those defined by himself. If some parameters need to be configured with values (for instance, δ , in the send-on-delta strategy example presented in the next subsection), the client sets them in this step.

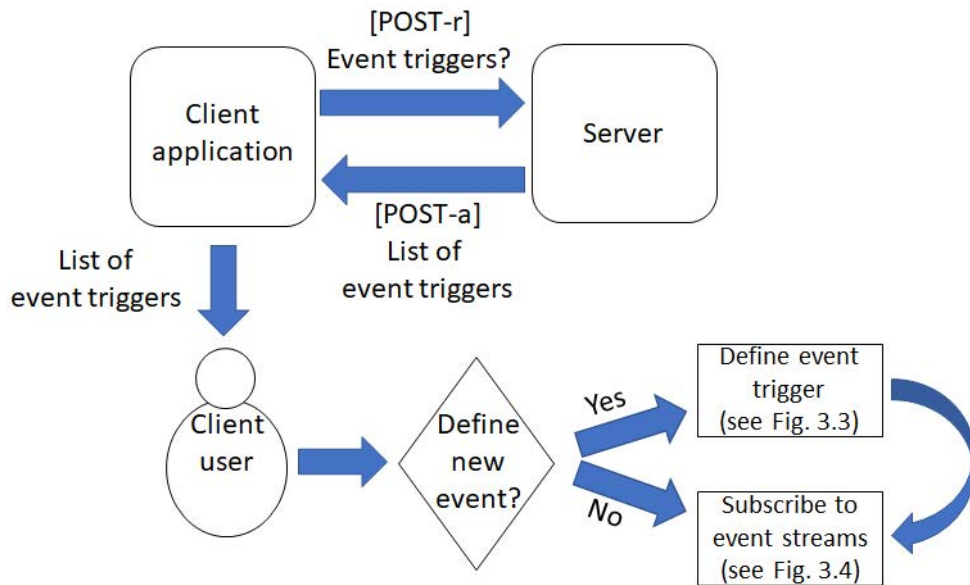


Figure 3.1: Global communication workflow in RIP for event subscription and definition. [POST-r] means a new POST request is issued from the client to the server and [POST-a] indicates that the server answers to the last received POST request.

3.6.1 Defining Server Events

The RIP specification considers three actors who can define server events: 1) the OL owner/provider, 2) the RIP specification itself and 3) the OL user. For the first two actors, this is a server-based feature. For the last one, this feature requires implementation in both the server and the client. Figure 3.2 provides more details.

While supporting three different actors to define event triggers may look redundant, each case presents its own advantages and disadvantages. This document will use the send-on-delta strategy as an event triggering condition example to illustrate the way each of the actors could define and implement it. While a detailed description of this technique can be found in the literature, for the purpose of this document it is enough to say that it relies on sending data every time the error of a controlled variable is bigger than a certain δ value. In mathematical form, and following the notation in Figure 3.2, data is sent to the client when:

$$e(t) = |y_{ref}(t) - y(t)| > \delta \quad (3.1)$$

1. *Event triggers included by the RIP SSE-based communication protocol implementation.* Any implementation of the proposed communication protocol must: 1) send the client signals to the plant and 2) send the output of the plant to the client. This information can be used by the RIP communication protocol implementation to check and determine if certain conditions are met to decide whether to send data (trigger the event) to the client or not. These built-in event triggers would only include parameterizable basic types of well-known event triggering conditions that require little insight of the process itself. The main advantage of letting RIP to include the definition of some general events is that neither the plant expert nor the user need to do it, saving time and work in many applications. Instead, only the parameters that affect the event trigger must be set, either by the plant expert or by the client user. This is the way any of these two actors provide its knowledge to the process control. In the example of the send-on-delta strategy, the logic to send data when Equation 3.1 is satisfied, would already be written in the RIP communication protocol implementation, ready to be used by any plant and client without any extra work rather than setting the value of δ in the triggers configuration file.
2. *Event triggers defined by the plant expert.* The RIP communication protocol implementation may not include an event trigger strategy that suits the needs of an specific OL, plant or process. In this case, the expert/owner of such system can easily define its own rules so that the communication protocol

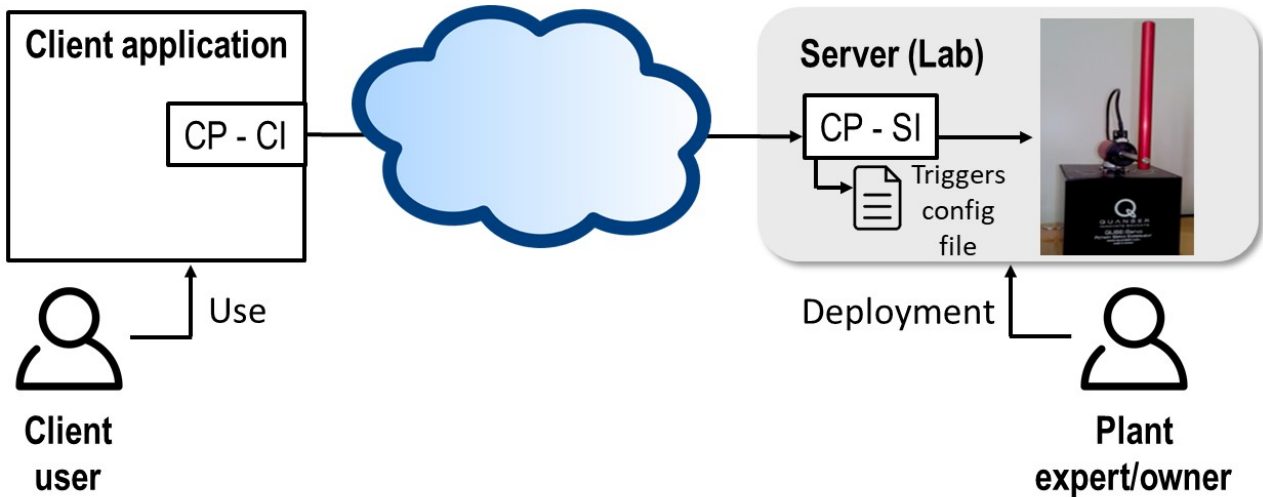


Figure 3.2: Actors that can play a role in defining event triggers: the plant owner/expert (who usually does the deployment of the system on the Internet), the client user (who uses the client application to control the plant) and the RIP communication protocol (CP) implementation itself, formed by both the RIP client implementation (CI) and the RIP server implementation (SI). Rules that define the event triggers are written in the *triggers configuration file*, which can be edited by any of the three actors.

understands them and use them for triggering the events that send the data to the client. The main advantage of this approach is that it provides the communication protocol with a great flexibility to support very specific and specialized event triggering conditions, allowing the expert/owner of the system to define the strategy that works best with the plant. The disadvantage is that it requires more work than just tuning the parameters of built-in event triggers. In the example of the send-on-delta strategy, the system expert/owner would need to write Equation 3.1 in the triggers configuration file, as well as set the value for the δ parameter.

3. *Event triggers defined by the client user.* It is easy to imagine scenarios in which it is interesting to allow a client user to define her own event triggering conditions. The first one is in the context of educational OLs where part of the learning practice is to make the student decide and implement the event trigger strategy. A second one is when neither the communication protocol implementation nor the plant owner/expert provide any event trigger on their own. The greatest advantage of this option is that any user can define its own event triggering rules, whether the communication protocol implementation or the plant expert/owner have include any or not. This allows the greatest flexibility but the cost is that it requires more knowledge from the client. In the example of the send-on-delta strategy, and end user would write Equation 3.1 in the web application and RIP would send it to the server where this rule would be written into the triggers configuration file.

Writing the event triggering conditions in the triggers configuration file can be done in two different ways, depending on the actor. When the plant owner/expert is defining her own rules or just setting the value of built-in triggers' parameters, the best option is that the RIP server implementation provides an editor interface to read and write the file. For supporting client users to define these rules or set the parameters of existing ones, RIP provides this feature as a web service. This is described in the next subsection.

The communication workflow for a client user to define event triggers is as follows (see Figure 3.3):

1. The client user indicates in the client application that a new event trigger is being defined.

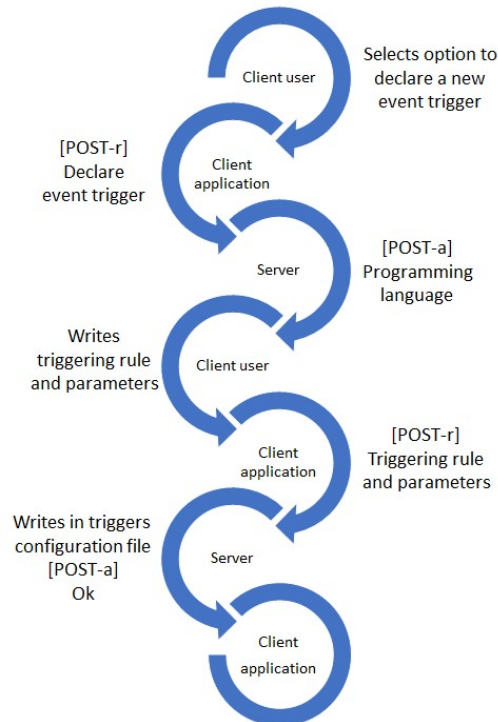


Figure 3.3: Communication workflow for a client that defines a new event trigger. [POST-r] means a new POST request is issued from the client to the server and [POST-a] indicates that the server answers to the last received POST request.

2. Through the communication protocol, the client application fires a POST request to the server.
3. The server answers with the name of the programming language in which the event triggering condition must be written.
4. The client user writes the triggering rule, including its possible parameters.
5. The client application, through the communication protocol, sends a new POST request to the server with this information.
6. The server writes the rule in the triggers configuration file.
7. The server responds to the client to let it know that it can start the parameterization and subscription request.

3.6.2 Subscribing to Server Events

This is a client-based feature.

The communication workflow for a client user to subscribe to a certain subset of event streams and set values for the required parameters is as follows (see Figure 3.4):

1. The client user selects which events is going to subscribe to and writes the values for the associated parameters.
2. The client application, through the communication protocol, fires a POST request to the server with the list of selected event streams and the values of the parameters.
3. The server answers to let know the client application that everything is settled and that the control of the process can start.

After the process is finished, the communication protocol implementation configures the SSE so that only the subscribed events, triggered by the selected triggering conditions, are received by the client. For simplicity, this work has considered that all even streams would contain the same information; for example, the output of the plant and its state. However, the proposed communication protocol can be easily extended to allow the

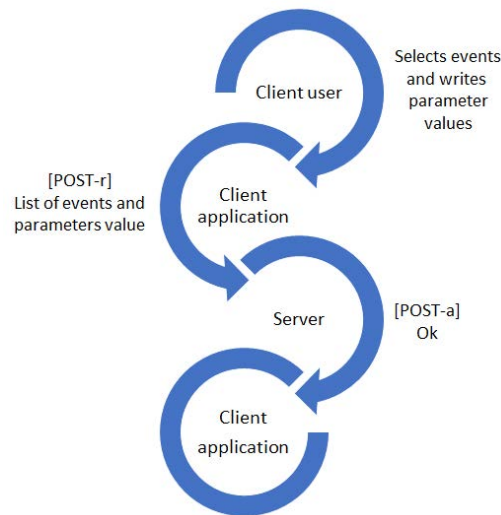


Figure 3.4: Communication workflow for a client that sets the parameters of an event trigger and subscribes to its event stream. [POST-r] means a new POST request is issued from the client to the server and [POST-a] indicates that the server answers to the last received POST request.

three actors to also define the information each event stream could contain. For example, an event stream created by a certain event trigger, A , may send to the client only the output of the plant, while the event stream produced by a different event trigger, B , may send the whole state of the plant.

3.6.3 Built-in Event Triggering Conditions

As mentioned before, RIP can include, and actually does, several event triggers that are predefined. These triggering conditions can be either used or ignored, depending on the configuration in the RIP server implementation and on the choices made by the end user.

Available built-in or predefined triggering conditions are:

1. *Periodiclabdata*. The first time this event is fired is upon the client connection to the server. Then, this event is triggered on a periodic basis, every time t milliseconds have passed since the client connected to the server. When fired, this event sends all readable variables in the control program or simulation model to the client through the SSE. Therefore, the only configuration parameter required to customize this event is t . See the example in section 2.8.2.2 to get a view on the contents of the *periodiclabdata* event.
2. *Send-on-delta*. TODO.
3. *TODO*.

Appendices

Appendix A

Glossary

Client implementation - The final software implementation made on the client side to talk RIP.

Control program - A software program which purpose is to control and monitor the hardware equipment used to in a laboratory activity.

Experience - Any lab activity that can be performed in an OL.

OL - Online laboratory.

RIP - Remote Interoperability Protocol.

RL - Remote laboratory.

Server implementation - The final software implementation made on the server side to talk RIP and communicate with the control programs or simulation models.

Simulation model - A mathematical simulation that models a certain system with which laboratory activities can be carried out.

SSE - Server-sent events.

VL - Virtual laboratory.

References

- [1] (1981) Transmission control protocol specification. [Online]. Available: <https://tools.ietf.org/html/rfc793>
- [2] (1999) Http 1.1 specification. [Online]. Available: <https://tools.ietf.org/html/rfc2616.html>
- [3] (2009) Server-sent events specification. [Online]. Available: <https://www.w3.org/TR/eventsource>
- [4] (2010) Json-rpc 2.0 specification. [Online]. Available: <https://www.jsonrpc.org/specification>
- [5] (2018) Xmlhttprequest standard. [Online]. Available: <https://xhr.spec.whatwg.org/>
- [6] (2018) Fetch standard. [Online]. Available: <https://fetch.spec.whatwg.org/>