SmartREST C++ Library

Developer's Guide

Version 1.23

Xinlei Cao

support@cumulocity.com

2016-06-29 Wed

Contents

List of Listings

List of Tables

Chapter 1

Introduction

The SmartREST `C++` library is a `C++` Software Developer Kit (SDK) for facilitating device integration to *Cumulocity*'s Internet of Things (IoT) platform.

SmartREST is *Cumulocity*'s innovative communication protocol specifically designed for the IoT world. It incorporates the highly expressive strength of the REST API, whereas at the same time replace JSON with Comma Separated Values (CSV) to avoid the complexity of JSON parsing for embedded devices. Additionally, the terseness of CSV renders it highly efficient for IoT communication via mobile networks. It can save up to 80% mobile traffic compared to other HTTP APIs.

The SmartREST `C++` library is designed for a wide range of devices which are powered by embedded Linux. It implements iterator-style lazy CSV lexer and parser, sophisticated request aggregation and robust request sending, and functionality for *Cumulocity*'s IoT integration, e.g., device registration, real-time device control, SmartREST template registration. The library employs a event-driven design which supports periodical timer callbacks and message based callbacks, which will greatly reduce the development process of integrating your IoT devices to *Cumulocity*'s IoT platform.

The `C++` library supports both HTTP and MQTT as the underlying communication protocol. HTTP is a well-established, widely-adopted application protocol. For IoT world, HTTP is considerably bloated and traffic heavy. Oppositely, MQTT is a emerging lightweight messaging protocol based on publish and subscribe mechanism, this renders it very suited for IoT use cases. The library is designed in a way such that any agent software based on the library can transit from HTTP to MQTT with very little effort.

In the following chapters, we will first provide guidelines about how to successfully build the library for your target environment. Then we demonstrate how to use the library by walk through a series of example agent ranging from simple "hello world" agent to complex agent which uses `Lua` plugins for sending measurements and handle operations. Subsequently, we will also demonstrate how to transit one complete example from using HTTP to use MQTT as underlying communication layer.

In the end, we provide a reference of all build macros for further tailor and tune the build to suit your needs in case you have special target devices.

Chapter 2

Building the Library

2.1 Prerequisites

Table 2.1: Prerequisites for building the library.

Software	Minimal Version	Comment
Linux	2.6.32	
gcc (clang)	4.7 (3.3)	both gcc and clang are supported
libcurl	7.26.0	older versions might work, but not tested
Lua	5.0	optional, for Lua plugin support only

2.2 Compiling the Library

First, download a copy of the library from the git repository and change to the directory[1].

```
1 $ git clone git@bitbucket.org:m2m/cumulocity-sdk-c.git
2 $ cd cumulocity-sdk-c
```

Second, initialize and update your submodule dependencies, since the library depends on the paho.mqtt.embedded-c library for MQTT support.

```
1 $ git submodule init
2 $ git submodule update
```

Then, create a *init.mk* file, and define specific macros *CPPFLAGS*, *CXXFLAGS* and *LDFLAGS*, *LDLIBS* and *CXX* if cross-compiling.

[1] You can also access the library repository at https://bitbucket.org/m2m/cumulocity-sdk-c.

```
CXX:=/usr/bin/g++
CPPFLAGS:=-I/usr/include
CXXFLAGS:=-Wall -pedantic -Wextra
LDFLAGS:=-L/usr/lib
LDLIBS:=-lcurl
```

Listing 1: example *init.mk* file

Listing 1 shows a typical *init.mk* file example. In essence, *init.mk* defines search path for required `c++` header files, preferred warning levels, search path for required `c++` library files, and necessary linking flags.

When you do host compiling, many of these settings can obviously be omitted, these are more relevant for cross-compiling, which shall be the prevalent use case for the library. Later we will explain the *init.mk* file is also very important for another purpse, i.e., build customization to tailor the library to your needs.

With the *init.mk* being defined, it's time to define your *makefile*.

```
$ cp Makefile.template Makefile
```

The default *Makefile.template* can be used unchanged in most cases. In case some settings are not suitable for your use case, e.g., you may want `-Os` optimization level instead of the default `-O2`, simply edit the copied *Makefile*.

Now we have done all preparation work, it's time to build the library for your target device.

```
$ make
```

If everything is configured correctly, this should compile the library and output the final binary into the *lib/* directory, and a watchdog daemon *srwatchdogd* into the *bin/* directory in the root directory.

The build system supports both *debug* and *release* modes. The above command calling make without any target defaults to *debug* build. *debug* build produces much larger binary, more verbose output, etc, which is suitable for development phase. When releasing your software, you will likely want a *release* build, you can clear all intermediate build files and re-build the library in *release* mode when you want to release your software.

```
$ make clean
$ make release
```

Chapter 3

Using the Library

3.1 Getting Started

Before we really get started, we will need a *Cumulocity* account. Go to `https://cumulocity.com/`, you can apply for a free trial account by click the "TRY CUMULOCITY FREE" button on the top-right corner. After signing-up and login to your tenant, you would find the device registration page in *Device Management*. Next, we will demonstrate how to register a device to *Cumulocity* using the library.

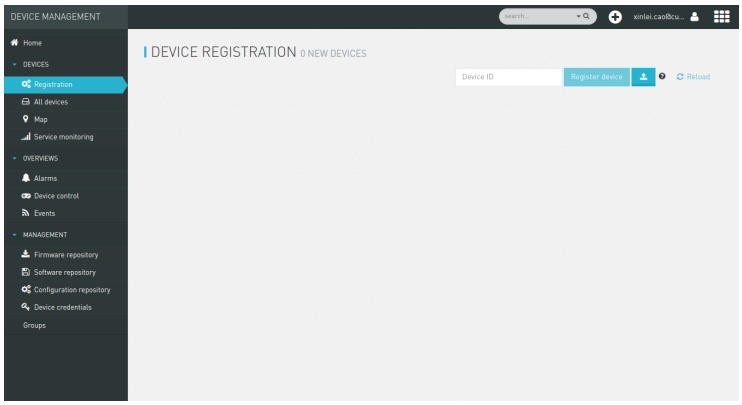

Figure 3.1: Cumulocity Registration Page.

Without any further ado, let's write our first program, the customary *hello world* [1] example shown in Listing 2.

[1] All examples can be found in the `examples` folder in the repository.

```
// ex-01-hello: src/main.cc
#include <iostream>
#include <sragent.h>
#include <srlogger.h>
using namespace std;

int main()
{
        const char *server = "http://developer.cumulocity.com";
        const char *credentialPath = "/tmp/helloc8y";
        const char *deviceID = "13344568";   // unique device identifier
        srLogSetLevel(SRLOG_DEBUG);          // set log level to debug
        SrAgent agent(server, deviceID);     // instantiate SrAgent
        if (agent.bootstrap(credentialPath)) // bootstrap to Cumulocity
                return 0;
        cerr << "Hello, Cumulocity!" << endl;
        return 0;
}
```

Listing 2: Hello Cumulocity

> It's strongly encouraged that you pick a different random value for `deviceID`, as it's the unique identifier of your device.

For convenience, let's define a shell variable `C8Y_LIB_PATH` to hold the library root path and use it to feed the compiler so it can find all necessary `C++` header files and shared library `.so` file.

```
$ export C8Y_LIB_PATH=/library/root/path
$ g++ -std=c++11 -I$C8Y_LIB_PATH/include -L$C8Y_LIB_PATH/lib -lsera main.cc
```

> You can define the variable `C8Y_LIB_PATH` in your `.bashrc` file so you don't need to define it every time when launching a new terminal. From now on, I'd assume you have done so and will mention no more about `C8Y_LIB_PATH` in later examples.

```
$ LD_LIBRARY_PATH=$C8Y_LIB_PATH/lib ./a.out
...
Hello, Cumulocity!
```

Finally, it's time to run our first program. Type the `deviceID` into the text field in your registration page (Fig 3.1) and click *Register device*. After the program is running, a green *Accept* button shall show up, click it to accept your device into your tenant.

As illustrated, the program will print *Hello, Cumulocity!* then exit. Voila, that's all we need to register a device to *Cumulocity*.

The obtained device credential is stored in `/tmp/helloc8y` as defined in variable `credentialPath`. You can also find the credential in the *Device credential* page in your *Cumulocity* portal.

> If you re-run the program the second time, the program will print *Hello, Cumulocity!* and exit immediately. This is because the program has loaded available credential from the given credential file. You can manually delete the credential file if you want the program to request a new credential.

3.2 Integrate to Cumulocity

Device integration is a little more complex. The whole process is depicted in Fig 3.2, please refer to the device integration guide for detailed explanation. Steps *1*, *2* and *3* are specific to SmartREST protocol as SmartREST requires predefined templates, see SmartREST guide and SmartREST reference for more information. Step *4* checks if the device is already stored in *Cumulocity*'s database and only create it when it's not found. Steps *6* and *7* get the *Cumulocity* ID of the device from *Cumulocity*'s database. Step *8* sets the *Cumulocity* ID as an alias for the device ID so that the device can find its *Cumulocity* ID next time by querying with its device ID.

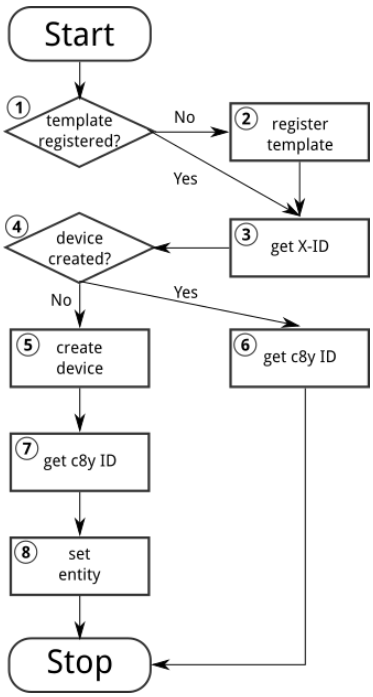

Figure 3.2: Device integration flowchart.

```
1  // ex-02-integrate: src/integrate.h
2  #ifndef INTEGRATE_H
3  #define INTEGRATE_H
4  #include <sragent.h>
5
6  class Integrate: public SrIntegrate
7  {
8  public:
9          Integrate(): SrIntegrate() {}
10         virtual ~Integrate() {}
11         virtual int integrate(const SrAgent &agent, const string &srv,
12                               const string &srt);
13 };
14
15 #endif /* INTEGRATE_H */
```

Listing 3: Integrate to Cumulocity: API inteface

Listing 3 shows the required API interface[2] by `SrAgent` when implementing your own integrate process. Basically, you need to subclass the pure virtual class `SrIntegrate` and realize its virtual function `integrate` with your particular integrate process. This is a callback function, which will be called by `SrAgent` when you call the `integrate` method of the `SrAgent`. By convention, the function shall returned 0 for success, and a non-0 value for failure.

```
1  // ex-02-integrate: src/integrate.cc
2  #include <srnethttp.h>
3  #include <srutils.h>
4  #include "integrate.h"
5  using namespace std;
6
7
8  int Integrate::integrate(const SrAgent &agent, const string &srv,
9                           const string &srt)
10 {
11         SrNetHttp http(agent.server()+"/s", srv, agent.auth());
12         if (registerSrTemplate(http, xid, srt) != 0) // Step 1,2,3
13                 return -1;
14
15         http.clear();
16         if (http.post("100," + agent.deviceID()) <= 0) // Step 4
17                 return -1;
18         SmartRest sr(http.response());
19         SrRecord r = sr.next();
20         if (r.size() && r[0].second == "50") { // Step 4: NO
21                 http.clear();
22                 if (http.post("101") <= 0) // Step 5
23                         return -1;
24                 sr.reset(http.response());
25                 r = sr.next();
26                 if (r.size() == 3 && r[0].second == "501") {
27                         id = r[2].second; // Step 7
28                         string s = "102," + id + "," + agent.deviceID();
29                         if (http.post(s) <= 0) // Step 8
30                                 return -1;
31                         return 0;
32                 }
33         } else if (r.size() == 3 && r[0].second == "500") { // Step 4: YES
34                 id = r[2].second;                           // Step 6
35                 return 0;
36         }
37         return -1;
38 }
```

Listing 4: Integrate to Cumulocity: implementation

Listing 4 realizes the flow chart depicted in Fig 3.2. You may have noticed all requests are Comma Separated Values (CSV) since we are using SmartREST instead of REST APIs directly. The corresponding SmartREST templates can be found in Listing 5. Important thing to note is that, you must store the correct SmartREST *X-ID* and device's *Cumulocity ID* in the inherited member variables `xid` and `id`, respectively. They will be used by `SrAgent` after the integrate process for initializing corresponding internal variables.

Listing 5 extends the code in Listing 2. The only addition inside the `main` function is the call to `SrAgent`'s member function `integrate` for integrating to *Cumulocity* and `loop` for executing the agent loop[3]. Above the `main` function is

[2]The API reference is located in relative path `doc/html/index.html` in the library repository.

[3]The agent loop is an infinite loop, so it will never really returns. We will get back to this

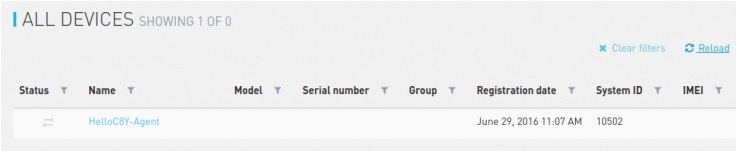

Figure 3.3: Created device in *Cumulocity* after integrate process.

the definition of the SmartREST template version number and actual template content[4].

Please refer to Section 3.1 about how to compile and run the code. After running this example code, you should see a device named `HelloC8Y-Agent` in *All devices* page in your *Cumulocity* tenant, as shown in Fig 3.3.

```
1  // ex-02-integrate: src/main.cc
2  #include <sragent.h>
3  #include <srlogger.h>
4  #include "integrate.h"
5  using namespace std;
6
7  static const char *srversion = "helloc8y_1"; // SmartREST template version
8  static const char *srtemplate =              // SmartREST template collection
9          "10,100,GET,/identity/externalIds/c8y_Serial/%%,,"
10         "application/json,%%,STRING,\n"
11
12         "10,101,POST,/inventory/managedObjects,application/json,"
13         "application/json,%%,,\"{\"\"name\"\":\"\"HelloC8Y-Agent\"\","
14         "\"\"type\"\":\"\"c8y_hello\"\",\"\"c8y_IsDevice\"\":{},"
15         "\"\"com_cumulocity_model_Agent\"\":{}}\"\n"
16
17         "10,102,POST,/identity/globalIds/%%/externalIds,application/json,,%%,"
18         "STRING STRING,\"{\"\"externalId\"\":\"\"%%\"\","
19         "\"\"type\"\":\"\"c8y_Serial\"\"}\"\n"
20
21         "11,500,$.managedObject,,$.id\n"
22         "11,501,,$.c8y_IsDevice,$.id\n";
23
24 int main()
25 {
26         const char *server = "http://developer.cumulocity.com";
27         const char *credentialPath = "/tmp/helloc8y";
28         const char *deviceID = "13344568"; // unique device identifier
29         srLogSetLevel(SRLOG_DEBUG);        // set log level to debug
30         Integrate igt;
31         SrAgent agent(server, deviceID, &igt); // instantiate SrAgent
32         if (agent.bootstrap(credentialPath))   // bootstrap to Cumulocity
33                 return 0;
34         if (agent.integrate(srversion, srtemplate)) // integrate to Cumulocity
35                 return 0;
36         agent.loop();
37         return 0;
38 }
```

Listing 5: Integrate to Cumulocity: main function

function later.

[4]Consult the SmartREST reference about how to define SmartREST templates.

3.3 Send Measurement

Now we have successfully integrated a demo device to *Cumulocity*, we can finally do something more interesting. Let's try sending CPU measurement every 10 seconds.

As shown in Listing 6[5], we need to first add a new SmartREST template for CPU measurement, and also increase the SmartREST template version number. Then we subclass the pure virtual class `SrTimerHandler` and implement the `()` operator. `CPUMEasurement` is a functor callback, which generates bogus CPU measurements using the `rand` function from the standard library. It will be called by the `SrAgent` at defined interval of the registered `SrTimer` .

In the `main` function, we instantiate a `CPUMEasurement` and register it to an `SrTimer` in the *constructor*. `SrTimer` supports millisecond resolution, so 10 seconds is 10 * 1000 milliseconds.

The library is built upon an asynchronous model. Hence, the `SrAgent` class is not responsible for any networking duty, it is essentially a scheduler for all timer and message handlers. `SrAgent.send` merely places a message into the `SrAgent.egress` queue, and returns immediately after. For actually sending SmartREST requests to *Cumulocity*, we need to instantiate a `SrReporter` object and execute it in a separate thread.

[5]The code excerpt only includes the added part, check the *examples* folder for the complete example code.

```cpp
// ex-03-measurement: src/main.cc
#include <cstdlib>

static const char *srversion = "helloc8y_2";
static const char *srtemplate =
// ...
        "10,103,POST,/measurement/measurements,application/json,,%%,"
        "NOW UNSIGNED NUMBER,\"{\"\"time\"\":\"\"%%\"\","
        "\"\"source\"\":{\"\"id\"\":\"\"%%\"\"},"
        "\"\"type\"\":\"\"c8y_CPUMeasurement\"\","
        "\"\"c8y_CPUMeasurement\"\":{\"\"Workload\"\":"
        "{\"\"value\"\":%%,\"\"unit\"\":\"\"%\"\"}}}\"\n"
// ...

class CPUMeasurement: public SrTimerHandler {
public:
        CPUMeasurement() {}
        virtual ~CPUMeasurement() {}
        virtual void operator()(SrTimer &timer, SrAgent &agent) {
                const int cpu = rand() % 100;
                agent.send("103," + agent.ID() + "," + to_string(cpu));
        }
};

int main()
{
        // ...
        CPUMeasurement cpu;
        SrTimer timer(10 * 1000, &cpu); // Instantiate a SrTimer
        agent.addTimer(timer);          // Add the timer to agent scheduler
        timer.start();                  // Activate the timer
        SrReporter reporter(server, agent.XID(), agent.auth(),
                            agent.egress, agent.ingress);
        if (reporter.start() != 0)      // Start the reporter thread
                return 0;
        agent.loop();
        return 0;
}
```

Listing 6: Send pesudo CPU measurement to Cumulocity.

If you add a `SrTimer` to the `SrAgent`, you must ensure its existence throughout the program lifetime[a], since there is no way to remove a `SrTimer` from the `SrAgent`. Instead, you can use `SrTimer.connect` to register a different callback or deactivate it by `SrTimer.stop`. This is a design choice for encouraging timer reuse, instead of dynamically creating and destroying timers.

[a]This is especially important when you dynamically allocate a timer on the heap, you must not destroy it during the program is running.

3.4 Handle Operation

Besides sending requests, e.g., measurements to *Cumulocity*, the other important functionality is handle messages, either responses from *GET* queries or real-time operations from *Cumulocity*. Listing 7 demonstrates how to handle the *c8y_Restart* operation. Again, first we will need to register necessary SmartREST templates. Then we define a message handler for handling restart operation.

In the `main` function, we register the `RestartHandler` for SmartREST template 502, which is the template for the restart operation. We also need to instantiate a `SrDevicePush` object and starting execute device push in another thread. From now on, as soon as you execute an operation from your *Cumuloc-*

ity portal, device push will receive the operation immediately and your message handler will be invoked by the `SrAgent`.

```
// ex-04-operation: src/main.cc
static const char *srversion = "helloc8y_3";
static const char *srtemplate =
// ...
        "10,104,PUT,/inventory/managedObjects/%%,application/json,,%%,"
        "UNSIGNED STRING,\"{\"\"c8y_SupportedOperations\"\":[%%]}\"\n"

        "10,105,PUT,/devicecontrol/operations/%%,application/json,,%%,"
        "UNSIGNED STRING,\"{\"\"status\"\":\"\"%%\"\"}\"\n"
// ...
        "11,502,,$.c8y_Restart,$.id,$.deviceId\n";
// ...

class RestartHandler: public SrMsgHandler {
public:
        RestartHandler() {}
        virtual ~RestartHandler() {}
        virtual void operator()(SrRecord &r, SrAgent &agent) {
                agent.send("105," + r.value(2) + ",EXECUTING");
                for (int i = 0; i < r.size(); ++i)
                        cerr << r.value(i) << " ";
                cerr << endl;
                agent.send("105," + r.value(2) + ",SUCCESSFUL");
        }
};

int main()
{
        // ...
        // Inform Cumulocity about supported operations
        agent.send("104," + agent.ID() + ",\"\"c8y_Restart\"\"");
        RestartHandler restartHandler;
        agent.addMsgHandler(502, &restartHandler);
        SrDevicePush push(server, agent.XID(), agent.auth(),
                          agent.ID(), agent.ingress);
        if (push.start() != 0)      // Start the device push thread
                return 0;
        agent.loop();
        return 0;
}
```

Listing 7: Handle relay operation

Now run the program, then go to your *Cumulocity* tenant, execute an restart operation as shown in Fig 3.4. You should see the message printed in `cerr` and the operation is set to *SUCCESSFUL* in your control tab in *Cumulocity*.

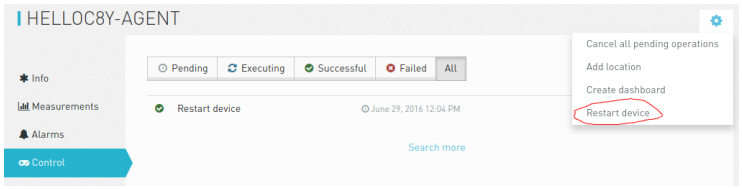

Figure 3.4: Execute a restart operation in *Cumulocity*.

3.5 Store SmartREST Templates in a File

Over time, your template collection would grow large, and you would like to store them in a text file instead of hard coding them in your source code. The benefits are tow-fold: you don't need to recompile the code every time only because the templates change, and there is no need to escape special characters which is error-prone.

A utility function `readSrTemplate` is provided for reading template collection from a text file. Listing 8 shows the usage of this function. It reads file *srtemplate.txt* from the current directory and stores the version number and template content into arguments `srversion` and `srtemplate`, respectively.

```
1  // ex-05-template: src/main.cc
2  #include <srutils.h>
3  // ...
4
5  int main()
6  {
7          // ...
8          string srversion, srtemplate;
9          if (readSrTemplate("srtemplate.txt", srverision, srtemplate) != 0)
10                 return 0;
11         // ...
12 }
```

Listing 8: Read SmartREST template from a text file.

The file format required by `readSrTemplate` is as simple as following:

- First line contains only the template version number.
- Every template must be on one line of its own.
- A line starts with # as first character (with no leading spaces or tabs) is considered a comment line and will be ignored.
- A complete empty line (with no spaces and tabs) will be ignored.
- No trailing spaces or tabs are allowed for any line except comment lines.

See listing 9 for an example of template file.

```
helloc8y_3

10,100,GET,/identity/externalIds/c8y_Serial/%%,,application/json,%%,STRING,

10,101,POST,/inventory/managedObjects,application/json,application/json,%%,,
    "{""name"":""HelloC8Y-Agent"",""type"":""c8y_hello"",
    ""c8y_IsDevice"":{},""com_cumulocity_model_Agent"":{}}"

10,102,POST,/identity/globalIds/%%/externalIds,application/json,,%%,STRING
    STRING,"{""externalId"":""%%"",""type"":""c8y_Serial""}"

10,103,POST,/measurement/measurements,application/json,,%%,NOW UNSIGNED
    NUMBER,"{""time"":""%%"",""source"":{""id"":""%%""}, ""type"":""c8y_CPUMeasurement"",
    ""c8y_CPUMeasurement"":{""Workload"":{""value"":%%,""unit"":""%""}}}"

10,104,PUT,/inventory/managedObjects/%%,application/json,,%%,UNSIGNED STRING,
    "{""c8y_SupportedOperations"":[%%]}"

10,105,PUT,/devicecontrol/operations/%%,application/json,,%%,UNSIGNED STRING,
    "{""status"":""%%""}"

11,500,$.managedObject,,$.id

11,501,,$.c8y_IsDevice,$.id

11,502,,$.c8y_Restart,$.id,$.deviceId
```

Listing 9: SmartREST template collection stored in a text file.

3.6 Lua Plugin

Instead of using `c++` for your development, the library also supports rapid development in `Lua`. For `Lua` plugin support, you must build the library with explicitly enabling `Lua` support, as it's disabled by default, see Chapter 4 about how to enable `Lua` plugin support.

Listing 10 demonstrates how to load a `Lua` plugin and add path `lua/` into `Lua`'s `package.path` for library search path.

```
// ex-06-lua: src/main.cc
#include <srluapluginmanager.h>
// ...

int main()
{
        // ...
        SrLuaPluginManager lua(agent);
        lua.addLibPath("lua/?.lua");  // add given path to Lua package.path
        lua.load("lua/myplugin.lua"); // load Lua plugin
        // ...
        return 0;
}
```

Listing 10: Load `Lua` plugin.

Listing 11 shows how to send CPU measurements and handle operation in `Lua` instead of `c++`. All `Lua` plugins are managed by `SrLuaPluginManager`, it is exposed to all `Lua` plugins as an opaque object named *c8y*. The only requirement for a `Lua` plugin is having a `init` function, which will be called by `SrLuaPluginManager` at load time to initialize the `Lua` plugin[6].

[6] Check `Lua` API reference in `doc/lua.html` for a complete list of all available APIs.

The example also shows how to define your own `Lua` library and share its variable `myString` in your `Lua` plugins.

```
-- ex-06-lua: lua/mylib.lua
myString = "Hello, Cumulocity!"

----------------------------------------

-- ex-06-lua: lua/myplugin.lua
require('mylib')
local timer

function restart(r)
   c8y:send('105,' .. r:value(2) .. ',EXECUTING')
   for i = 0, r.size - 1 do      -- index in C++ starts from 0.
      srDebug(r:value(i))
   end
   c8y:send('105,' .. r:value(2) .. ',SUCCESSFUL')
end

function cpuMeasurement()
   local cpu = math.random(100)
   c8y:send('103,' .. c8y.ID .. ',' .. cpu)
end

function init()
   srDebug(myString)            -- myString from mylib
   timer = c8y:addTimer(10 * 1000, 'cpuMeasurement')
   c8y:addMsgHandler(502, 'restart')
   return 0                     -- signify successful initialization
end
```

Listing 11: Send measurement and handle operation using `Lua`.

> You may encounter an error saying "Package lua was not found in the pkg-config search path." when building this example, then you would need to modify the expression `$(shell pkg-config --cflags lua)` to add a proper version number to `lua`. The proper version number depends on your installed Lua version and your Linux distribution.

3.7 Use MQTT instead of HTTP

MQTT is a publish and subscribe based light-weight messaging protocol, renders it very suitable for IoT communication. It solves two major issues inherit to HTTP: 1) HTTP header predominantly overweights SmartREST payload since SmartREST messages are generally very short. 2) MQTT has built-in support for real-time notification via subscribe and publish mechanism, hence, there is no need for a separate connection for device push.

Above examples are all using HTTP as the transportation layer. Besides HTTP, `SrReporter` also supports MQTT as the transportation layer. Listing 12 shows the modification needed for transforming the example in Section 3.4 from using HTTP into using MQTT.

```
// ex-07-mqtt-legacy: src/main.cc

int main()
{
        // ...
        SrReporter reporter(string(server) + ":1883", deviceID, agent.XID(),
                            agent.tenant() + '/' + agent.username(),
                            agent.password(), agent.egress, agent.ingress);
        // set MQTT keep-alive interval to 180 seconds.
        reporter.mqttSetOpt(SR_MQTTOPT_KEEPALIVE, 180);
        if (reporter.start() != 0)      // Start the reporter thread
                return 0;
        agent.loop();
        return 0;
}
```

Listing 12: Use MQTT for sending measurements and handle operations.

As you can see, all modification needed is to construct `SrReporter` with a different constructor so `SrReporter` knows to use MQTT as underlying communication protocol, and remove `SrDevicePush` in the code since MQTT has built-in support for real-time notification. Optionally, you can set the keep-alive interval for MQTT to prevent the underlying TCP connection from being interrupted.

Chapter 4

Build Customization

In Chapter 2 we briefly explained how to build the library, in this chapter we will go into depth about how to customize the build options to tailor a optimal build for your particular use case.

All customization options listed in the following shall be added in your *init.mk* file.

1. `SR_PLUGIN_LUA=0`

   Switch for `Lua` plugin support, defaults to 0, which disables `Lua` support. Set it to 1 will enable `Lua` plugin support. Also remember to provide necessary `Lua`'s `C` library and add to your `CPPFLAGS`, `CXXFLAGS`, `LDFLAGS` and `LDLIBS` the required compile and link flags, etc.

2. `SR_PROTO_HTTP_VERSION=1.1`

   HTTP version, defaults to 1.1. Set it to 1.0 for environments when `HTTP/1.1` is not supported.

3. `SR_SOCK_RXBUF_SIZE=1024`

   Maximum receive buffer size for `SrNetSocket`, defaults to 1024 bytes. This number dictates the maximum number of bytes the `recv` method of `SrNetSocket` can block waiting for response. This parameter only affects the receive buffer of `SrNetSocket`.

4. `SR_AGENT_VAL=5`

   Polling interval for `SrAgent`, defaults to 5 milliseconds. Internally `SrAgent` schedules all `SrTimerHandler` and `SrMsgHandler` by constantly polling for expired `SrTimer` and arrived messages from ingress `SrQueue`, this parameter dictates the interval between two consecutive polling. When is parameter is set too high, the agent may appear to be sluggish, whereas when set too low, many CPU cycles are wasted. This is a trade-off parameter that needs to be fine-tuned for any particular device.

5. `SR_REPORTER_NUM=512`

   Maximum number of aggregated requests, defaults to 512. For saving traffic use, `SrReporter` has a mechanism to aggregate many messages into one request and send them all in once. This number dictates the maximum number of messages that can be aggregated.

6. `SR_REPORTER_VAL=400`

   Maximum waiting time between two consecutive requests for aggregation, defaults to 400 milliseconds. When aggregating requests, `SrReporter` will wait for consecutive messages with a defined timeout. If the next messages comes after the timeout, `SrReporter` will stop the waiting loop and starts sending the already aggregated messages. When set to a higher number, higher aggregation can be expected, therefore, results in lower traffic use, whereas when set to a lower number, agent will be more responsive since it will not wait for aggregating next message. This is a trade-off parameter that needs to be fine-tuned for any particular use case.

7. `SR_REPORTER_RETRIES=9`

   Maximum number of retries when sending fails, defaults to 9 times. For counteracting temporary network failures, `SrReporter` implemented an exponential wait and multi-trials measure. When the first trial fails, it waits 1 second and retries again, when the second trial fails, it waits 2 seconds, when the third trial fails, it waits 4 seconds, and so on, until the defined number of retries exhausted. Note when `SrReporter` enters the retry loop, messages sent via `SrAgent` will be queued up in the egress `SrQueue`, until the `SrReporter` successfully sends the aggregated requests so far or exhausts all retries.

8. `SR_CURL_SIGNAL=1`

   Whether allow *libcurl* from installing any signal handlers, defaults to 1, which allows *libcurl* to install signal handlers. Certain versions of *libcurl* contains a bug that when built with a synchronous DNS resolver, randomly crashes when the DNS lookup timed out. When you experience this issue, you can workaround this bug by disabling *libcurl* from installing signal handlers. As a side effect, *libcurl* will not be able to terminate DNS lookup, recommended approach is to re-built *libcurl* with an asynchronous DNS resolver.

9. `SR_SSL_VERIFYCERT=1`

   Whether to verify server's certificate when using HTTPS, defaults to 1. Many embedded devices have no CA certificates installed and thus not be able to verify server's certificate when communicating via HTTPS. As a workaround, you can disable certificate verification by setting this macro to 0.

10. `SR_FILEBUF_PAGE_SCALE=3`

    Set scale of page size for file backed buffering, default is 3. When `filebuf` feature is enabled for `SrReporter`, messages are managed at a minimum unit of one page, instead of single message, for easy and efficient buffer managing. Therefore, larger page size will buffer more messages, but messages are also discarded in bigger chunks. In contrary, smaller page size buffers less messages, but messages are also discarded in smaller chunks. Possible page scale values and corresponding page size can be found in Table 4.1.

Table 4.1: List of page scale and corresponding page size for filebuf.

Page Scale	Page Size
0	512 B
1	1 KB
2	2 KB
3	4 KB
4	8 KB
5	16 KB
6	32 KB
7	64 KB