Cave Canem—An extensible DDS-based monitoring and Intrusion Detection System

Developer's Guide

1 Introduction

Performance and health monitoring are essential to ensure that a system meets its operational needs—especially in distributed systems. Currently, there are plenty of solutions addressing those issues, however, they tend to serve a single purpose (e.g., performance monitoring vs. intrusion detection) and to be limited in scalability (e.g., based on a centralized client-server model).

In this context, we propose Cave Canem, an open distributed framework for the collection and integration of events, alerts, and information regarding the system status. To achieve this goal, Cave Canem builds a Common Operational Picture (COP) to combine information from disparate sensor classes. Its COP normalizes all the relevant information from those sensors into a distributed operational model, making the information available for observation in a seamless and efficient manner.

To construct the COP, Cave Canem exploits the benefits of the Object Management Group (OMG) Data Distribution Service for Real-Time Systems (DDS) standard. DDS defines a data-centric publish-subscribe (DCPS) communication model to exchange the information. It allows the middleware infrastructure to reflex the essential aspects of the information model, cache the required information, provide content- and time- based filtering and deliver the information to the applications with the correct quality-of-service (QoS).

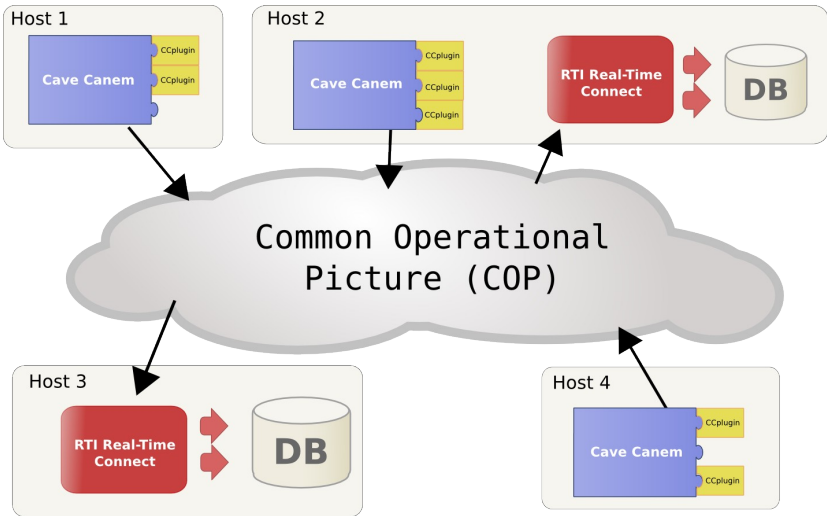

Figure 1: Global overview.

2 Architecture

Cave Canem is based on an extensible plugin architecture. It is built on a core application that deals with configuration issues, the load and unload of plugins, and the inner aspects of the creation and destruction of DDS entities.

To preserve its modularity, all the functionality of Cave Canem relays on its set of plugins. Each plugin represents a monitoring agent that gathers and publishes periodically information related to a certain monitoring target. As a result, Cave Canem can be described as a universal monitoring agent that remains open to address new monitoring objectives without making changes to its structure and avoiding the need to recompile the application. Figure 2 presents a simple monitoring scenario where two plugins—CPU plugin and Snort plugin—collect information related to their monitoring target and publish it on the DDS Global Data Space (GDS).

One other important aspect of the monitoring infrastructure is the storage of information. Cave Canem uses RTI Real-Time Connect (RTC) to collect the information published and to store it on a relational database for online and offline processing. To ease that processing, Cave Canem includes a Web User Interface (UI) that provides simple graphical and textual information regarding the status of the COP.

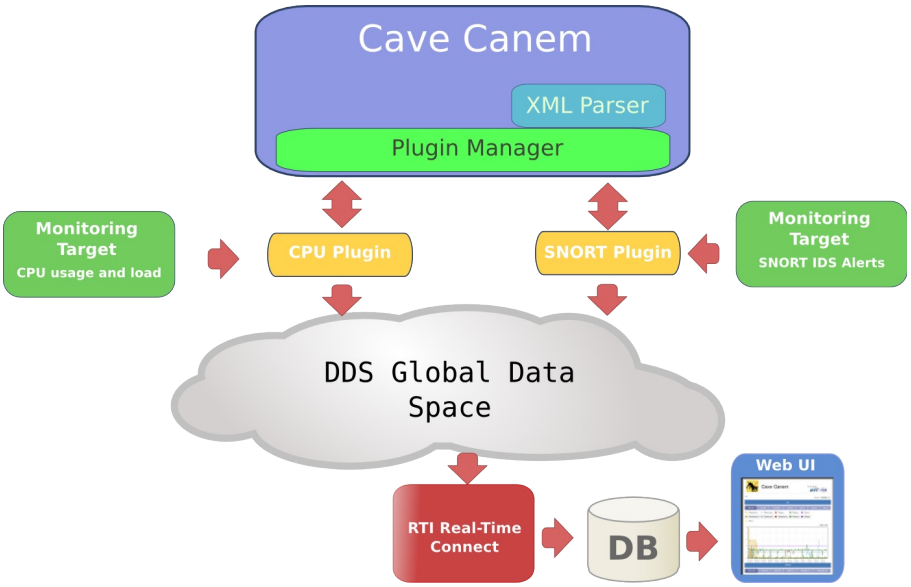

Figure 2: Simple monitoring scenario.

2.1 Core

As mentioned above, Cave Canem's core deals with the basic aspects of application's management. In particular, the module in charge of those tasks is *plugin manager*.

At the beginning of Cave Canem's execution, the plugin manager extracts the information contained in the main configuration file, *cavecanem.xml*. This XML configuration file includes basic information regarding the behavior of the application, i.e., the setting up of the DDS entities, and the plugins to be loaded during the execution of the program. Once recognized, *plugin manager* loads the specified plugins accessing the dynamic libraries where they are stored. Afterwards, it creates all the DDS entities, i.e., one Domain Participant and Publisher—that will be shared by all the plugins—and one Data Writer for each plugin. The properties of this latter entity are defined in the XML configuration file of each plugin.

Plugin manager relies on a different module to parse the information contained in the XML files, *xml parser*. It uses the RTI DDS XML parsing API, which provides functions to parse QoS policies and data type definitions. Therefore, the definition of the data types used by the topics, as well as the QoS policies of the Data Writer, may be specified in the plugin configuration files (see Section 3.3).

Finally, the core controls the finalization of the application, unloading the plugins and deleting the existing DDS entities when Cave Canem is called to close.

2.2 Plugins

In Cave Canem, each plugin represents a monitoring agent. Therefore, each plugin must check periodically the current status of a monitoring target and publish the information collected in the DDS GDS. Cave Canem plugins share a common structure defined in the *plugin.hpp* file. As a result, each plugin must implement a couple of methods to allow *plugin manager* to guide the process of calling the plugins to collect the monitoring information and to publish it at a given rate. That rate is specified in the configuration file of each plugin.

The development of plugins in Cave Canem does not involve a deep use of the API of DDS. Instead, *plugin.hpp* provides a method to abstract the developer from using the API to publish the information. This process is detailed in Section 4.2.

2.3 RTI Real-Time Connect

Cave Canem uses RTI RTC to store the monitoring information published by different hosts running Cave Canem instances. RTI RTC provides bidirectional integration between RTI DDS and a relational database. As a result, it is able to publish the information gathered and stored in the database according to the needs of the user.

The goal of Cave Canem's use of RTI RTC is to keep the monitoring information to perform both online and offline processing, e.g., the information stored in the database may be used for anomaly detection.

RTI RTC's configuration is based on XML files. In those files it is possible to specify the location of the database and the properties of the subscription—including content-filtering, QoS definitions, etc. The contents of the information published by each plugin—regarding a DDS topic—are stored on a table in the database.

2.4 Web User Interface

To provide a general overview of the status of Cave Canem's COP, the system includes a Web UI. It uses the relational database where RTI RTC stores information to extract and present a textual and graphical description of the monitoring scenario and its evolution.

3 Configuration System

As stated before, Cave Canem provides a set of XML files to customize its behavior—two general configuration files and one configuration file for each plugin. This section describes their contents, specifying the function of each parameter.

3.1 cavecanem.xml

cavecanem.xml defines the general properties of the execution of Cave Canem. It is located at the *config/* directory within *cavecanem-0.2.0/*.

- `<general>`
  - `<publishing_period_sec>`—Cave Canem polls for new plugins willing to publish at the indicated rate.
- `<dds_properties>`
  - `<dds_domain_id>`—DDS domain in which the application publishes the information.
  - `<dds_qos_file>`—XML file where QoS libraries and profiles are defined (relative path).
  - `<dds_qos_default_library>`—QoS library for the DDS Domain Participant and the DDS Publisher.
  - `<dds_qos_default_profile>`—QoS profile for the DDS Domain Participant and the DDS Publisher. If the default profile is indicated—the profile's name is `default`—Cave Canem uses the QoS default policies without taking into account the selected QoS library. Therefore, Cave Canem does not support the definition of QoS profiles called `default` in the QoS XML configuration file.
- `<plugins>`
  - `<plugin_library dir="">`—Defines a new directory containing plugins. Each plugin has to be located in a folder with same name as the plugin within that directory. At least, the folder must include the dynamic library containing the plugin's class and an its XML configuration file.
    - `<plugin>`—Name of a plugin to be loaded from the plugin library.

3.2 cavecanem_qos.xml

Located at *config/* as well, it contains a set of QoS libraries and profiles. However, a custom QoS configuration file may be specified in the `<dds_qos_file>` parameter of *cavecanem.xml*.

It defines two QoS libraries and QoS profiles for testing and deployment purposes, respectively. The *deployment* profile is designed to keep information available for late-joiner subscriber applications, whereas the *testing* profile only ensures reliable communication.

More information on QoS configuration can be found in the RTI Data Distribution Service User's Manual, as well as in the QoS reference guide[1].

3.3 Plugin configuration files

Each plugin must provide a XML configuration file according to the structure described bellow. The XML file must have the same name as the plugin and be located within the plugin's directory.

- `<plugin name="">`—Name of the plugin. By default, the name of the plugin defines the DDS Topic's name as well.
  - `<dll>`—Name of the dynamic library where the plugin is located. The name must be

1 Available at the RTI Community Portal: http://community.rti.com/sites/default/files/RTI_DDS_QoS_Reference_Guide.pdf.

independent of the platform, i.e., if we set `<dll>` to `cpu`, Cave Canem will try to open *libcpu.so* in Linux and Solaris, whereas in Windows, it will try to open *cpu.dll*.

- `<create_function>`—Name of the function that creates the plugin by returning an instance of the plugin's class (see Section 4.2).
- `<publishing_period_sec>`—Publication rate of the plugin.
- `<dds_properties>`—To define the QoS policies regarding the DDS Data Writer that publishes the information collected by the plugin, the user must either indicate a pair QoS library QoS profile or define the DDS Data Writer's policies within the `<datawriter_qos>` tag using the RTI DDS syntax for defining QoS.
  - [Optional] `<dds_qos_library>`—QoS library for the DDS Data Writer.
  - [Optional] `<dds_qos_profile>`—QoS profile for the DDS Data Writer. As stated before, if `default` is indicated, Cave Canem will use the QoS default policies for the DDS Data Writer, without taking into account the QoS library defined.
  - [Optional] `<datawriter_qos>`—Defines—using the RTI DDS syntax for QoS XML configuration files—the QoS policies for the DDS Data Writer .
  - [Optional] `<dds_topic_name>`—Defines the DDS Topic name. It allows the specification of a DDS Topic name different from the plugin's name.
- `<plugin_config>`—Allows the inclusion of custom parameters to configure the plugin.
  - [Optional] `<plugin_element name="">`—Each parameter must be defined with a name and a value—the contents of the tag.
- `<type_definition type_name="">`—Definition of the data type associated with the DDS Topic published by the plugin. It avoids the need to use *rtiddsgen*.

4 Plugins

4.1 Plugins included

Cave Canem includes seven plugins for basic system and network monitoring using the Hyperic Sigar library[2]. It also includes one plugin to perform intrusion detection using the Intrusion Detection System (IDS) Snort[3].

Cave Canem plugins may be used as an example for the development of new plugins for custom needs. A brief description of each plugin is provided below.

4.1.1 CPU

CPU plugin publishes CPU usage and load of the host where Cave Canem is running.

4.1.2 Memory

Memory plugin gathers information about the usage of memory and swap.

2 The Hyperic Sigar library is available, under the Apache License v2.0, at: http://sourceforge.net/projects/sigar/
3 Snort may be downloaded from: http://www.snort.org/.

4.1.3 Disk

Disk plugin provides information regarding the usage of the different file systems and network file systems the host is using.

4.1.4 Net Load

Net Load plugin publishes information about the configuration, usage, and errors of the different network interfaces.

4.1.5 Host Info

Host info plugin provides information about the operating system running, as well as the uptime of the host.

4.1.6 Proc

Proc plugin gathers information about each process running on the host. It maintains one instance per process that includes information about its CPU and memory usage, the user, group, etc.

4.1.7 Proc Stat

Proc Stats plugin provides information regarding the general status of the processes running on a host. This information includes the number of processes running, sleeping, the number of threads...

4.1.8 Snort

The Snort plugin gathers and publishes alerts generated by the IDS Snort. For that purpose, it uses its CSV output plugin in its default configuration. Therefore, the user must add the following line to use that output in the Snort's configuration file —usually located at */etc/snort/snort.conf*—to configure the output needed:

```
output alert_csv: /var/log/snort/alert.csv default
```

To configure the Cave Canem's Snort plugin to read alerts from */var/log/snort/alert.csv* the user has to specify a new plugin element, `log_file`, within the `<plugin_config>` tag in its configuration file.

```
<plugin_element name="log_file">/var/log/snort/alert.csv</plugin_element>
```

To provide a common interface for the integration of IDS alerts, allowing the subscriber to receive and process information from different IDS, the Snort plugin publishes a topic called *ids_alert*—using the `<dds_topic_name>` tag in its XML configuration file. Therefore, new IDS-related plugins should use the same data type and topic name to provide interoperability in the dissemination of IDS alerts.

4.2 How to develop a new plugin

One of the main goals of this manual is to provide a guide for extending Cave Canem's functionality. This section explains the steps of the development process of a new plugin.

As stated in Section 2.2, plugins are built according to common interface. It is specified by the base class `cc_plugin`, defined in *main/plugin.hpp*. To create, for instance, a *Hello World* plugin[4] it is necessary to define a new C++ class implementing the virtual methods specified by `cc_plugin`. The definition of the `hello_world` class regarding our example should be implemented in a file called *hello_world.hpp* as described below.

```
/* hello_world plugin (plugins/hello_world/hello_world.hpp)
 */

#include <plugin.hpp>

/* First, it is necessary to inherit from cc_plugin */
class hello_world : public cc_plugin {
public:
      /* Then, create the constructor, which must get the plugin_id
       * and a map of properties by default */
      hello_world(std::string plugin_id,
                  std::map<std::string,std::string> properties);

      /* The virtual destructor */
      virtual ~hello_world();

      /* Main method of the plugin. The collection and publication of
       * information must be implemented within it */
      virtual bool generate_and_publish_information(DDSDynamicDataWriter *writer,
                                                    DDS_DynamicData *data);

      /* This method returns just a string with the plugin class name */
      virtual std::string plugin_class()

      {

            return "hello_world";

      }
private:

      /* Method to configure stuff needed to initialize the plugin */
      bool initialize_plugin(std::map<std::string, std::string> properties);

      /* Data to be published on the DDS Global Data Space */
      long i_;

};


/* Defines the C create function of the Hello World plugin, therefore
 * it defines the class factory */
extern "C" cc_plugin* create_hello_world(std::string plugin_id,
                           std::map<std::string,std::string> properties) {
    return new hello_world(plugin_id,properties);
}
```

Each plugin must provide an XML configuration file. It defines the data type related to the DDS Topic the plugin will publish, as well as some information related to the dynamic library where the plugin is stored (see Section 3.3). In our example, *hello_world.xml* should look as follows:

4 The Hello World plugin will publish a long value that will be incremented in each publication.

```
<!-- hello_world plugin (plugins/hello_world/hello_world.xml)-->

<plugin name="hello_world">
 <!--Name of the library. In Linux and Solaris it will be named after
   the compilation as libhello_world.so - or as libhello_worldd.so if debugging.
   In Windows it will be called hello_world.dll-->
  <dll>hello_world</dll>

  <!--Name of the 'create' function defined in hello_world.hpp that returns
      an object of the plugin class-->
  <create_function>create_hello_world</create_function>

 <!--The plugin will publish every 3 seconds when called by the plugin manager-->
  <publishing_period_sec>3</publishing_period_sec>

  <!-- Defines some DDS properties-->
  <dds_properties>
    <dds_qos_library>deployment</dds_qos_library>
    <dds_qos_profile>deployment</dds_qos_profile>
   <!--It is possible to define the DataWriter QoS adding a <datawriter_qos>
        tag here and removing <dds_qos_library> and <dds_qos_profile>-->
   <!--It is also possible to change the name of the topic published by the
        plugin adding a <dds_topic_name> tag here-->
  </dds_properties>

  <!-- Some parameters needed by the plugin may be added here-->
  <plugin_config>
      <!--<plugin_element name="parameter_name">value</plugin_element>-->
  </plugin_config>

  <!-- Definition of the data type associated with the DDS Topic published by the
   plugin. See the definition of other plugins for further information-->
  <type_definition type_name="hello_world">
    <struct name="hello_world">
      <member name="hello_counter" type="long"/>
    </struct>
  </type_definition>

</plugin>
```

Finally, the plugin must implement the methods defined above in *hello_world.hpp*. Those methods should be included then in a third file, called *hello_world.cpp.*

```
/* hello_world plugin (plugins/hello_world/hello_world.cpp)
 */

#include "hello_world.hpp"

using namespace std;

/* Constructor of the class. Everything to initialize will be implemented by
 * initialize_plugin(). In case of failure, the constructor throws an exception
 * that will be handled by the plugin manager */
hello_world::hello_world(string plugin_id,
                         map<string,string> properties)
{
    if(!initialize_plugin(properties))
      throw runtime_error("hello_world plugin could not be initialized");
}

/* Destructor of the class */
hello_world::~hello_world(void)
{
    // Customize if needed
}

/*  In the initialize_plugin() method we initialize some things related to the
 *  plugin. In this case, we initialize the long that will be published to 0 */
bool hello_world::initialize_plugin(map<string,string> properties)
{
    i_ = 0;
    return true;
}

/* This method extracts and publishes the information. It takes a
 * DDS_DynamicData structure given as a parameter and must fill the structure
 * according to the definition of the data type in the XML configuration file.
 *
 * It uses the DDS Dynamic Data API, that provides methods to set the members
 * depending on their data type. For further information check the implementation
 * of other Cave Canem plugins or the RTI DDS API documentation.
 *
 * In this case, the structure will be filled with a long that represents the
 * number of times that generate_and_publish_information() has been called.
 * When finished filling data, the method will call publish_information -- defined
 * in the base class -- that will take care of the publication of the information
 * in the DDS Global Data Space */
bool hello_world::generate_and_publish_information(DDSDynamicDataWriter *writer,
                                                   DDS_DynamicData *data)
{
    // Sets the value of i to the member of the structure using the DDS API
    data->set_long("hello_counter", DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED, i_);
    // Increments the value of i for next call.
    i_++;
    // Then let the base class do the job of publishing
    if(!publish_information(writer, data))
      return false;
    return true;
}
```

To use the recent defined plugin, it is necessary to add the compiled dynamic library and the XML configuration file in a folder with the plugin's name within an existent or new plugin library directory. Plugin libraries are specified in *config/cavecanem.xml* (see Section 3.1).

All the plugins included with Cave Canem provide their own *makefile* for Unix-like systems, or *project* for Visual Studio in Microsoft Windows. The Hello World plugin's *makefile* is provided as an example above. For further information on Unix-like and Microsoft Windows plugin compilation, check the implementation of the Cave Canem plugins.

```
# HELLO WORLD Makefile (plugins/hello_world/makefile)

# Detect both the platform (Linux, SunOS...) and the machine (i686,x86_64...) to
# be able to choose the right FLAGS and libraries.
PLATFORM := $(shell uname -s)
MACHINE := $(shell uname -m)

# Provide the path to config.mk. Config.mk provides all the flags and libraries
# needed to compile everything in Cave Canem for Linux and Solaris, given the
# platform and machine.
CONFIG_MK_PWD = ../../shared

# Define the name of the library.
ifeq ($(DEBUG),1)
LIB_NAME = libhello_worldd.so
else
LIB_NAME = libhello_world.so
endif

# Include config.mk -- using the previously defined path to the file.
include $(CONFIG_MK_PWD)/config.mk

# Define the source files and the objects to be used.
SOURCES = hello_world.cpp
OBJECTS = $(SOURCES:%.cpp=%.o)

# Define the compilation targets. In this case, it is necessary to add
# include and ld flags for Sigar -- the monitoring library. However, usually
# it is only necessary to add include and ld common flags, that include all the
# DDS compilation stuff needed.
all: $(OBJECTS)
      $(CC) -o $(LIB_NAME) $(OBJECTS) $(LDFLAGS_SIGAR) $(LDFLAGS_COMMON)

%.o : %.cpp
      $(CC) $(CFLAGS) $(INCLUDES_COMMON) $(INCLUDES_SIGAR) -o $@ -c $<

clean:
      rm -f *.o *.so *~
```

5 RTI Real-Time Connect

RTI RTC provides bidirectional integration between RTI DDS and relational databases. It means, it can be used to collect information from disparate sensors, process and analyze it, and then propagate the

results to remote entities. It is designed to integrate existing systems with minimal modification, using a daemon that subscribes to DDS applications according to a configuration based on XML files. In those files it is possible to specify the location of the database and the properties of the subscription—including content-filtering, QoS definitions, etc.

Cave Canem uses RTI RTC to get the monitoring data published by different hosts running the publisher application and to store it on a relational database for online and offline processing—e.g., the visualization of the information using the Web UI. For that purpose, it provides an XML configuration file with everything prepared to start using the application.

The XML configuration file defines two QoS profiles—testing and deployment—complementary to the profiles described in Section 3.2:

- *testing::testing*
  - `<datareader_qos>`
    - `<reliability>`
      - `<kind>RELIABLE_RELIABILITY_QOS</kind>`—Samples not received will be repaired by the middleware infrastructure.
    - `<history>`
      - `<kind>KEEP_LAST_HISTORY_QOS</kind>`–The DDS DataReader will keep only the number of samples defined in `<depth>` in its cache.
      - `<depth>1</depth>`

- *deployment::deployment*
  - `<participant_qos>`
    - `<discovery>`—Sets DDS discovery properties. It allows to define a set of initial peers that the DDS Participant has to take into account, and whether it will accept the discovery of new unknown peers while executing.
      - `<initial_peers>`
      - `<accept_unknown_peers>`
  - `<datareader_qos>`
    - `<reliability>`
      - `<kind>RELIABLE_RELIABILITY_QOS</kind>`—Samples not received will be repaired by the middleware infrastructure.
    - `<durability>`
      - `<kind>TRANSIENT_DURABILITY_QOS</kind>`—As DDS publishers keep a set of samples in memory for late-joiner subscribers, RTI RTC, as a subscriber, will be able to receive data produced in the past.
    - `<history>`
      - `<kind>KEEP_ALL_HISTORY_QOS</kind>`—The DDS DataReader will keep all the data, so it may be able to collect all the information in case of late-joining.

Moreover, the file defines a real-time connection called *cavecanem* to create the subscriptions. In particular, it defines a MySQL connection with Database Source Name (DSN) *cavecanem* and an user and password. The DSN should be configured using the *~/odbc.ini* file or any equivalent method. For each DDS Topic the user wants to subscribe the application to (each plugin) a new `<subscription>`

tag must be added. By default, the XML configuration file includes eight subscriptions, one for each plugin. The subscription to the memory plugin is described below:

```
…
<subscription>
      <!--User owner of the table-->
      <table_owner>cavecanem</table_owner>
      <!--Name of the database table where the samples will be kept.-->
      <table_name>memory</table_name>
      <!--Domain ID of the subscription-->
      <domain_id>127</domain_id>
      <!--Topic RTI RTC will subscribe to-->
      <topic_name>memory</topic_name>
      <!--Type name of the suscription-->
      <type_name>memory</type_name>
      <!--Number of samples that will be stored for each instance.
          By default, it is set to 9216 -- 32 days of samples produced
          every five minutes (300 seconds)-->
      <table_history_depth>9216</table_history_depth>
      <!--Qos library::Qos profile used in the suscription-->
      <profile_name>deployment::deployment</profile_name>
      <!--Forces the filtration of duplicated samples-->
      <filter_duplicates>1</filter_duplicates>
      <!--Forces the persistence of the state of the database-->
      <persist_state>1</persist_state>
</subscription>
…
```

Once the subscription is configured properly, RTI RTC may be run using the following command:

```
$ rtirtc_mysql -noDaemon -cfgfile ~/RTI/RTI_Real-Time_Connect_4.5d/resource/xml/
RTI_REAL_TIME_CONNECT_cavecanem.xml -cfgname cavecanem -queueDomainId 125
```

6 Web User Interface

Cave Canem provides a Web User Interface (UI) that presents a textual and graphical description of the monitoring scenario and its evolution. The Web UI is based on PHP, JavaScript and JQuery and uses an open source library, called Flot[5], to produce graphical plots.

The Web UI follows the Model-View-Controller (MVC) pattern in the definition of its architecture. Therefore, it provides a set of classes to control the behavior and data of the application domain—the model—a set of classes to deal with the user interface—the view—and a set of classes to accept the input from the user and perform the actions based on that input—the controller.

6.1 Model

As stated before, the model manages the behavior and data of the application domain. The data that the Web UI handles is stored on the database controlled by RTI RTC. As a result, the application must provide a set of classes to get the information stored in the database and process it for its visualization.

5 Flot is available under the MIT License at: http://code.google.com/p/flot/

In particular, the Web UI provides eight classes (`cpu`, `disk`, `host_info`, `ids`, `memory`, `net_load`, `proc`, and `proc_stat` located in separate files in the the *plugins/* directory) to handle the information related to each plugin defined in Cave Canem. *plugin.php* defines the interface that the plugin-related classes have to implement (each plugin-related class may implement only a subset of the interface, therefore the developer has to choose which methods are suitable for the plugin's nature), it includes the following methods:

- `get_all_data_from_table($history,$point_distance)`—Takes the monitoring data stored in the database table associated with the plugin.
  - Parameters:
    - `$history`—specifies that the data collected from the database must have a *timestamp* greater or equal than a given value. Therefore, for instance, to get the information stored from the last hour, the method must be called using a `$history` equal to the current *timestamp* minus 3600 seconds.
    - `$point_distance`—specifies the time distance that the values taken from the database must have. That is, if the method is told to get information stored from the last hour, it must get a subset of values—taking sometimes one value from every few values—so that it does not return more than a certain amount of data. This amount of data is associated with the width number of pixels of the graph to be drawn.
  - The method returns an array with the following elements:
    - `$data_map`—A map structure with the list of values collected from the database, indexed by host and by column—each column represents a metric from the plugin. That is, in PHP syntax, `$data_map['hostname']['column_name'][]`.
    - `$numeric_fields_array`—is an array with the numeric fields extracted from the database (columns). Each element of the array is another array with the name of the numeric field and its unit, e.g., `array("swap_used","KB")`.
- `get_all_data_from_table_single_query($history, $point_distance)`—Depending on the value of `$history` and the nature of the plugin, it is more efficient to get information from the database in one single query or to get it using multiple queries. As a result, most part of the plugin-related classes of the Web UI provide more than one implementation of `get_all_data_from_table()`. The controller must decide then which method is more efficient for each case. In particular, this implementation has demonstrated to be more efficient with higher values of `$history`, i.e., information closer in time.

- `get_all_data_from_table_multiple_query($history,$max_points)`—Many plugin-related classes include also a multiple query implementation of `get_all_data_from_table()`. It has demonstrated to be more efficient with smaller values of `$history`, i.e., information further in time.

- `get_host_data_from_table($hostname,$history)`—Takes the monitoring information from a table corresponding to a given host. It is mainly used in plugin-related classes with only not numeric fields, as the process does not involve taking into account the number of points to be drawn in a graph and, as as result, the point distance.
  - Parameters:
    - `$hostname`—Name of the host the method has to get information from.

        - `$history`—The method must get data with timestamps greater than the value of this parameter.
        - [Optional] `$limit`—Number of rows to be collected from the database. Sometimes it involves getting the top `$limit` values for a parameter, e.g., top fifteen cpu-consumer processes.
    - It returns an array with:
        - A map of the values got indexed by column name, i.e., `$data_map['column_name'][]`.
        - An array with the name of the not numeric fields (columns).

- `get_host_data_from_table_for_graph($hostname,$history,$point_distance)`—In contrast to the last method, `get_host_data_from_table_for_graph()` is implemented by plugin-related classes dealing with numeric fields to be drawn in a graph (although, these classes may include also not numeric fields). Therefore, it includes a point distance parameter that indicates the distance in time there has to be between the values got from the database—for a given host in a time interval.
    - Parameters:
        - `$hostname`—Name of the host the method has to get information from.
        - `$history`—The method must get data with timestamps greater than the value of this parameter.
        - `$point_distance`—Distance in time that the values got from the database must have.
    - It returns an array with:
        - A map of the values got indexed by column name, i.e., `$data_map['column_name'][]`.
        - An array with the name of the numeric fields (columns) and their unit—each element will be, as a result, a two element array.
        - An array with the name of the not numeric fields (columns).
        - A string with the name of the key column, in case it is needed, e.g., disk and net_load are organized by name (file system name), and device, respectively. The key will be used to identify the element in the graph.

The information about the database, including its location, user and password is defined in *db/db_manager.php*. This file includes a function that returns those values in an array, and is therefore used by all the plugin-related classes.

6.2 View

The View is the element that deals with the user interface. In the Cave Canem Web UI, the role of the View is done by a single PHP class called `ui`.
`ui`—defined in *ui/ui.php*—includes all the HTML and JavaScript code that the application uses, providing several methods to print textual and graphical information:

- `print_header()`—Prints the headers of the web page and opens the `<body>` tag. It includes some JavaScript code to get the time zone of the user and calculate the time offset that must be taken into account when converting the UTC timestamps to human readable dates. It also prints the button to select the history to be shown in the Web UI.

- `print_pagebody()`—Opens the `page_body` div.

- `print_pathway($hostname,$sensor)`—Prints the pathway, depending on the scenario being shown (see Section 6.3). If no sensor or host are being shown, the parameters have to be filled with empty strings.
- `print_bottom()`—Prints the bottom of the web page closing all opened tags.
- `init_script()`—Opens a JavaScript piece of code.
- `end_script()`—Closes a JavaScript piece of code.
- `draw_plugin_with_devices_host_info($table_name,$data_map, $numeric_fields_array,$all_history,$start_time,$end_time)`—Draws a graph for a set of data from multiple devices running on the same host, e.g., the packets dropped by the network interfaces on a host.
    - Parameters:
        - `$table_name`—Name of the plugin or table from the database.
        - `$data_map`—Map with the data to be drawn, indexed by device name and metric.
        - `$numeric_fields_array`—Array of the numeric fields (metrics) and their unit.
        - `$all_history`—Earliest *timestamp* that will be drawn. It sets the left limit of the graph when the user drags to pan. Therefore, it can be different from the `$start_time` parameter.
        - `$start_time`—Minimum value of the time axes when the graph is printed.
        - `$end_time`—Maximum value of the time axes—current *timestamp*.
- `draw_plugin_without_devices_overall_info($table_name,$data_map, $numeric_fields_array,$all_history,$start_time,$end_time)`—Draws a graph for a set of data from multiple hosts, e.g., the CPU load of four different hosts in last five minutes.
    - Parameters:
        - `$table_name`—Name of the plugin or table from the database.
        - `$data_map`—Map with the data to be drawn, indexed by device name and metric.
        - `$numeric_fields_array`—Array of the numeric fields (metrics) and their unit.
        - `$all_history`—Earliest *timestamp* that will be drawn. It sets the left limit of the graph when the user drags to pan. Therefore, it can be different from the `$start_time` parameter.
        - `$start_time`—Minimum value of the time axes when the graph is printed.
        - `$end_time`—Maximum value of the time axes—current *timestamp*.
- `draw_plugin_without_devices_host_info($table_name,$data_map, $numeric_fields_array,$all_history,$start_time,$end_time)`—Draws a graph for a set of data from one single host, e.g., the CPU load of a laptop.
    - Parameters:
        - `$table_name`—Name of the plugin or table from the database.
        - `$data_map`—Map with the data to be drawn, indexed by device name and metric.

        - `$numeric_fields_array`—Array of the numeric fields (metrics) and their unit.
        - `$all_history`—Earliest *timestamp* that will be drawn. It sets the left limit of the graph when the user drags to pan. Therefore, it can be different from the `$start_time` parameter.
        - `$start_time`—Minimum value of the time axes when the graph is printed.
        - `$end_time`—Maximum value of the time axes—current *timestamp*.

- `open_table($table_name,$fields_array)`—Creates a new HTML table and fills the table header with a set of fields given as a parameter.
    - Parameters:
        - `$table_name`—Name of the table.
        - `$fields_array`—Array with the fields that will appear in the table header.

- `open_host_table_single_data($table_name,$numeric_fields_array, $not_numeric_fields_array,$hostname)`—Creates a new HTML table to print the latest values of the metrics related to a plugin in a given host. The table is usually printed above the graph where the evolution of data is shown. Its caption is linked to a different page that shows the whole list of values collected by the plugin for that host—the behavior of the link is a task of the Controller.
    - Parameters:
        - `$table_name`—Name of the table.
        - `$numeric_fields_array`—Array with the numeric fields that will appear in the table header.
        - `$not_numeric_fields_array`—Array with the numeric fields that will appear in the table header.
        - `$hostname`—Name of the host the table is showing information from.
- `open_host_table_all_data($table_name,$numeric_fields_array, $not_numeric_fields_array)`—Creates a new HTML table to show the data collected by a plugin in a given host.
    - Parameters:
        - `$table_name`—Name of the table.
        - `$numeric_fields_array`—Array with the numeric fields that will appear in the table header.
        - `$not_numeric_fields_array`—Array with the numeric fields that will appear in the table header.

- `open_not_numeric_table($table_name,$not_numeric_fields_array,$hostname)`—Creates a new HTML table that will only be filled with not numeric data.
    - Parameters:
        - `$table_name`—Name of the table.
        - `$not_numeric_fields_array`—Array with the not numeric fields that will appear in

the table header.
        - `$hostname`—Name of the host the table shows information from.

- `fill_host_table_single_data($table_data,$numeric_fields_array, $not_numeric_fields_array, $key_name)`—Fills a table with the latest data collected by a plugin for a given host.
    - Parameters:
        - `$table_data`—Data the table will be filled with. The data is indexed by column name —values of the `$numeric_fields_array` and the `$not_numeric_fields_array`.
        - `$numeric_fields_array`—Array with the numeric fields that will appear in the table body.
        - `$not_numeric_fields_array`—Array with the not numeric fields that will appear in the table body.
        - `$key_name`—Name of the key column, if it does exists, e.g., the name of a file system or the name of a network interface. If the table does not have a key, the parameter should be an empty string.

- `fill_host_table_all_data($table_data,$numeric_fields_array, $not_numeric_fields_array,$key_name,$time_offset)`—Fills a table with all the data collected by a plugin for a given host.
    - Parameters:
        - `$table_data`—Data the table will be filled with. The data is indexed by column name —values of the `$numeric_fields_array` and the `$not_numeric_fields_array`.
        - `$numeric_fields_array`—Array with the numeric fields that will appear in the table body.
        - `$not_numeric_fields_array`—Array with the not numeric fields that will appear in the table body.
        - `$key_name`—Name of the key column if it does exists.
        - `$time_offset`—Usually, *timestamp* is one of the columns printed in this table. Therefore, if we want to print the date and hour according to the user's time zone, the time offset of that zone from UTC—in seconds—is required. If not, the parameter should be equal to `0`.

- `fill_not_numeric_table_all_data($table_data,$not_numeric_fields_array, $key_field,$time_offset)`—Fills a table with all the data collected by a plugin for a given host when it does not have to include numeric information.
    - Parameters:
        - `$table_data`—Data the table will be filled with. The data is indexed by column name —values of the `$not_numeric_fields_array`.
        - `$not_numeric_fields_array`—Array with the not numeric fields that will appear in the table body.
        - `$key_field`—Name of the key column if it does exists.

        - `$time_offset`—Usually, *timestamp* is one of the columns printed in this table. Therefore, if we want to print the date and hour according to the user's time zone, the time offset of that zone from UTC—in seconds—is required. If not, the parameter should be equal to `0`.
- `fill_not_numeric_table_host_data($table_data,$not_numeric_fields_array, $hostname,$time_offset)`—Fills a table with the latest data collected by a plugin for a given host if it does not include numeric information, e.g., IDS alerts.
    - Parameters:
        - `$table_data`—Data the table will be filled with. The data is indexed by column name —values of the `$not_numeric_fields_array`.
        - `$not_numeric_fields_array`—Array with the not numeric fields that will appear in the table body.
        - `$key_name`—Name of the key column if it does exists.
        - `$time_offset`—Usually, *timestamp* is one of the columns printed in this table. Therefore, if we want to print the date and hour according to the user's time zone, the time offset of that zone from UTC—in seconds—is required. If not, the parameter should be equal to `0`.
- `print_not_numeric_table_all_data($table_data,$not_numeric_fields_array, $key_field, $table_name)`—Does all the job of printing a table filled with not numeric data using JavaScript. This method should be used by the Controller when there is not a way of getting the time offset of the user's time zone in PHP—the user's time zone offset is got using a JavaScript library instead of using PHP and therefore we need to pass get the value somehow. That is the case of the IDS alerts printed in the home page (see Section 6.3 for further information).
    - Parameters:
        - `$table_data`—Data the table will be filled with. The data is indexed by column name —values of the `$not_numeric_fields_array`.
        - `$not_numeric_fields_array`—Array with the not numeric fields that will appear in the table body.
        - `$key_field`—Name of the key column if it does exists.
        - `$table_name`—Name of the table to print.

- `close_table()`—Closes a table.

6.3 Controller

The Controller receives user input and performs actions based on that input. In our scenario, this will involve using the Model and View classes depending on the actions of the user within the web page.

Our Web UI reproduces three different scenarios:

- *Main page*—Shows all the information collected from different hosts running Cave Canem. It allows the user to select the hosts must be drawn in each graph, as well as the history, e,g., last hour, last month, etc.

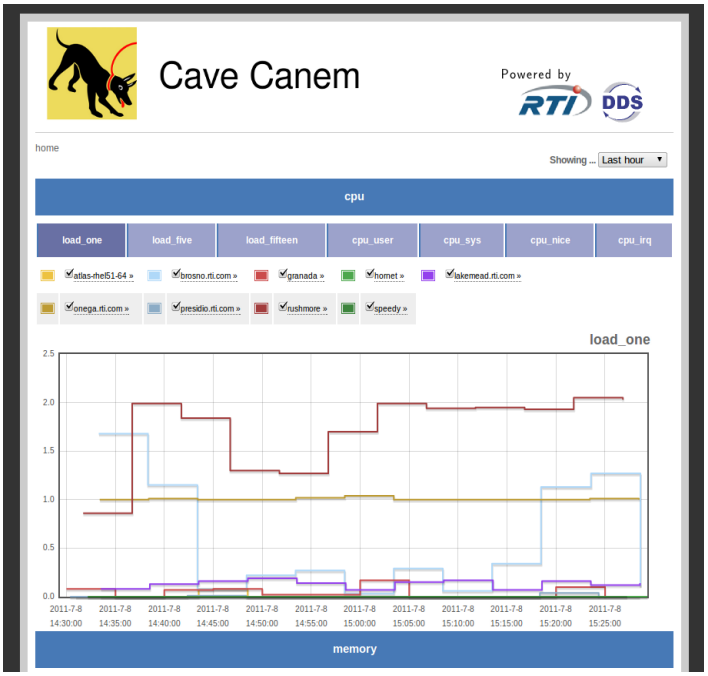

Figure 3: Main page

- *Host information page*—Shows both graphical and textual information collected from a host.

Figure 4: Host information page

- *Historical plugin information page*—Shows all the information collected by a plugin from a given host.

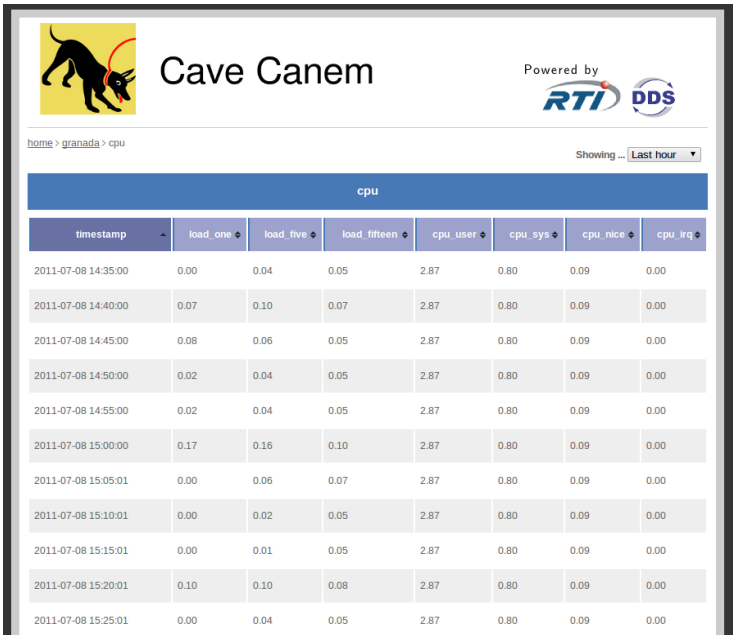

Figure 5: Historical plugin information page.

The Controller is implemented in *index.php* and *page_body.php*. *index.php* uses the View to print the header and the bottom of the web page, leaving the page body empty.

The contents of the page body will be filled according to the user's choices by the functions defined in *page_body.php*. In particular, this is done by checking the contents of the HTTP REQUEST variable:

- `$_REQUEST['history']`—Indicates the history the Web UI has to show, e.g., last hour, last month, etc.
- `$_REQUEST['offset']`—Indicates the time offset in seconds of the user's time zone from UTC. As stated before, the Web UI detects the user's time zone using JavaScript[6]. As a result, the value of the variable containing the time offset must be passed via HTTP POST to the PHP side. In case it is not possible—when the page is loaded for the first time—the View provides a method to fill the data using JavaScript accessing directly to the value of the JavaScript time offset variable (see Section 6.2).
- `$_REQUEST['host']`—Indicates whether the information to be shown is related to a host. If this variable is set, then the Web UI must show either the *Host information* page or the *Historical plugin information* page, depending on the value of `table`. On the other hand, if `host` is not set or is empty, the *Main* page must be shown.
- `$_REQUEST['table']`—Indicates the name of table (plugin) the *Historical plugin information* page scenario will show information from.

6 Cave Canem gets the user's time zone and the offset of that time zone using jsTimeZoneDetect. It is available at: https://bitbucket.org/pellepim/jstimezonedetect/wiki/Home.