Cloud-TM

Specific Targeted Research Project (STReP)

Contract no. 257784

Companion document for deliverable D3.1: Prototype of the Workload Monitor

Date of preparation: 10 June 2011
Start date of project: 1 June 2010
Duration: 36 Months

Contributors

Francesco Quaglia, CINI
Roberto Palmieri, CINI
Pierangelo Di Sanzo, CINI
Bruno Ciciani, CINI
Paolo Romano, INESC-ID

Contents

1 Introduction

This document is a companion of the software release associated with deliverable D3.1 of the Cloud-TM project, namely the prototype implementation of the Workload and Performance Monitor (WPM). The document presents design/development activities carried out by the project's partners in the context of WP3 (Task 3.1).

More in detail, the goal of this document is twofold:

- providing an overview of the WPM architecture, including a description of (i) the technologies used for the implementation of the prototype and (ii) its core mechanisms/functionalities;
- reporting a README associated with the release of the WPM prototype.

It is important to clarify that the design/implementation decisions taken in relation to deliverable D3.1 are not meant as conclusive and inalterable. Conversely, they express the results of activities performed during the first year of the project, which will need to be validated (and possibly be subject to changes) while progressing with the research and development work to be carried out during the future phases of the project.

The structure of this document is the following. In Section 2 we discuss what kind of existing technologies have been leveraged to build the WPM, and how they have been extended and integrate to achieve efficient interoperability with the ecosystem of components forming the Cloud-TM platform. Section 3 highlights the key architectural choices underlying the development of the WPM. Finally, in Section 4, we provide instructions on how to install and test the WPM.

1.1 Relationship with other deliverables

The prototype implementation of the WPM has been based on the user requirements gathered in the deliverable D1.1 "User Requirements Report", and taking into account the technologies identified in the deliverable D1.2 "Enabling Technologies Report".

The present deliverable has also a relation with the deliverable D2.1 "Architecture Draft ", where the complete draft of the architecture of the Cloud-TM platform is presented.

2 WPM Technology

The Workload and Performance Monitor (WPM) is a building-block component of the Autonomic Manager of the Cloud-TM platform. WPM must implement monitoring functionalities and mechanisms aimed at gathering statistics related to differentiated layers belonging to the Cloud-TM platform. As stated in the "User Requirement Report (D1.1)", these layers entail the hardware level (i.e. physical or virtualized hardware resources) as well as logical resources, such as data maintained by the data platform. The set of User Requirements directly impacting the structure of WPM are those in the interval [38-43] and [45]. Generally speaking, these requirements entail indications on:

- the type of resources whose usage must be monitored/profiled;
- the nature (e.g. in terms of standardization or de-facto reference) of the APIs or applications on top of which monitoring tasks should operate;
- the on-line/real-time operative mode of the monitoring system.

Anyway, WPM is also in charge of monitoring the performance parameters which will be selected as representative for the negotiated QoS levels.

2.1 The WPM skeleton

The skeleton of our prototype implementation of WPM is based on the Lattice framework [1], which offers a conceptually very simple means for instantiating distributed monitoring systems. This framework relies on a reduced number of interacting components, each one devoted (and encapsulating) a specific task in relation to distributed data-gathering activities. In terms of interaction abstraction, the Lattice framework is based on the producer-consumer scheme, where both the producer and consumer components are, in their turn, formed by sub-components, whose instantiation ultimately determines the functionalities of the implemented monitoring system. A producer contains data sources which, in turn, contain one or more probes. Probes read data values to be monitored, encapsulate measures within measurement messages and put them into message queues. Data values can be read by probes periodically, or as a consequence of some event. A message queue is shared by the data source and the contained probes. When a measurement message is available within some queue, the data source sends it to the consumer, which makes it available to reporter components. Overall, the producer component injects data that are delivered to the consumer. Also, producer and consumer have the capability to interact in order to internally (re)configure their operating mode.

Three logical channels are defined for the interaction between the two components, named:

- data plane;
- info plane;
- control plane.

The data plane is used to transfer data-messages, whose payload is a set of measures, each kept within a proper message-field. The structure of the message (in terms of amount of fields, and meaning of each field) is predetermined. Hence, message-fields do not need to be explicitly tagged so that only data-values are really transmitted, together with a concise header tagging the message with very basic information, mostly related to source identification and timestamping. Such a structure can be anyway dynamically reconfigured via interactions supported by the info plane. This is a very relevant feature of Lattice since it allows minimal message footprint for (frequently) exchanged data-messages, while still enabling maximal flexibility, in terms of on-the-fly (infrequent) reconfiguration of the monitoring-information structure exchanged across the distributed components within the monitoring architecture.

Finally, the control plane can be used for triggering reconfiguration of the producer component, e.g., by inducing a change of the rate at which measurements need to be taken. Notably, the actual transport mechanism supporting the planes is decoupled from the internal architecture of producer/consumer components. Specifically, data are disseminated across these components through configurable distribution mechanisms ranging from IP multicast to publish/subscribe systems, which can be selected on the basis of the actual deploy and which can even be changed over time without affecting other components, in term of their internal configuration. The framework is designed to support multiple producers and multiple consumers, providing the chance to dynamically manage data source configuration, probe-activation/deactivation, data sending rate, redundancy and so on.

2.2 Portability issues

The Lattice framework is based on JAVA technology, so that producer/consumer components encapsulate sub-components that are mapped onto a set of JAVA threads, each one taking care of specific activities. Some of these threads, such as the data-source or the data-consumer constitute the general purpose backbone of the skeleton provided by Lattice. Other threads, most notably the probe-thread and the reporter-thread, implement the actual logic for taking/reporting measurement samples. The implementation of these threads can be seen as the ad-hoc portion of the whole monitoring infrastructure, which performs activities tailored to specific measurements to be taken, in relation to the context where the monitoring system operates.

By the reliance on JAVA, portability issues are mostly limited to the implementation of the ad-hoc components. As an example, a probe-thread based on direct access to the "proc" file system for gathering CPU/memory usage information is portable only across (virtualized) operating systems supporting that type of file system (e.g. LINUX). However, widening portability across general platforms would only entail reprogramming the internal logic of this probe, which in some cases can even be done by exploiting, e.g., pre-existing JAVA packages providing platform-transparent access to physical resource usage.

The aforementioned portability considerations also apply to reporter-threads, which can implement differentiated, portable logics for exposing data to back-end applications (e.g. by implementing logics that store the data within a conventional database).

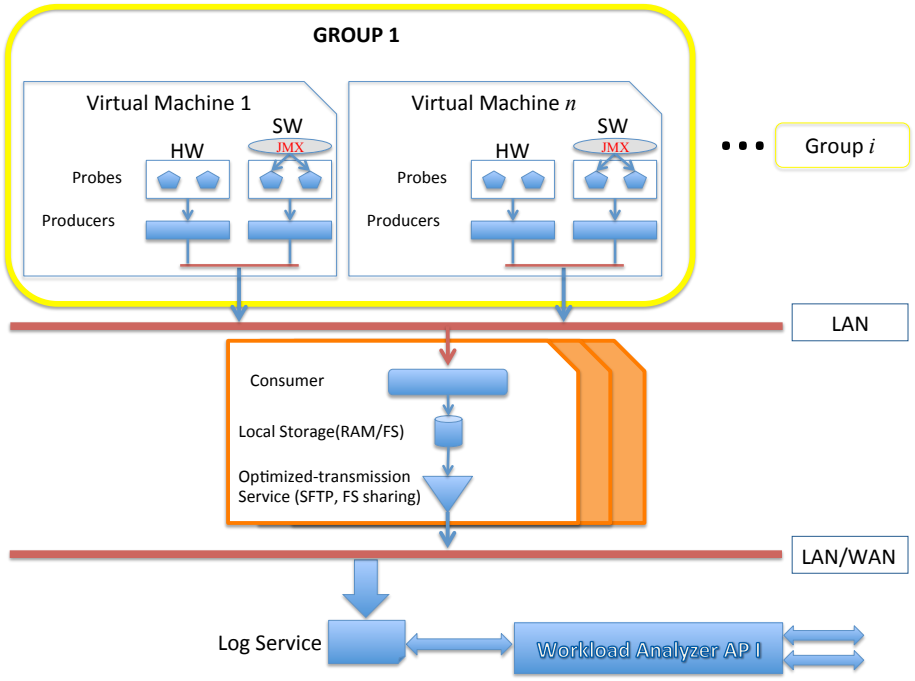

Figure 1: WPM Architectural Organization.

3 WPM Architectural Organization

Figure 1 shows the general architectural organization we have devised for WPM. It has been defined according to the need for supporting the following two main functionalities:

- statistical data gathering (SDG);
- statistical data logging (SDL).

The SDG functionality maps onto an instantiation of the Lattice framework, with Cloud-TM specific probes and collectors. In our instantiation, the elements belonging to the Cloud-TM infrastructure, such as Virtual Machines (VMs), can be logically grouped, and each group will entail per-machine probes targeting two types of resources: (A) hardware/virtualized and (B) logical. Statistics for the first kind of resources are directly collected over the Operating System (OS), or via OS decoupled libraries, while statistics related to logical resources (e.g. the data-platform) are collected at the application level by relying on the JMX framework for JAVA components.

The data collected by the probes are sent to the producer component via the facilities natively offered by the Lattice framework. Each producer is coupled with one or many probes and it is responsible of managing them. The consumer is the Lattice

component that receives the data from the producers, via differentiated messaging implementations, which could be selected on the basis of the specific system deploy. We envisage a LAN based clustering scheme such that the consumer is in charge of handling one or multiple groups of machines belonging to the same LAN. Anyway, in our architectural organization, the number of consumers is not meant to be fixed, instead it can be scaled up/down depending on the amount of instantiated probes/producers. Overall, the consumer can be instantiated as a centralized or a distributed process. Beyond collecting data from the producers, the consumer is also in charge of performing a local elaboration aimed at producing a suited stream representation to be provided as the input to the Log Service, which is in turn in charge of supporting the SDL functionality.

We have decided to exploit the file system locally available at the consumer side to temporarily keep the stream instances to be sent towards the Log Service. The functional block which is responsible for the interaction between SDG and SDL is the so called optimized-transmission service. This can rely on top of differentiated solutions depending on whether the instance of SDL is co-located with the consumer or resides on a remote network. Generally speaking, with our organization we can exploit, e.g., SFTP or a locally shared File System. Also, stream compression schemes can be actuated to optimize both latency and storage occupancy.

The Log Service is the logical component responsible for storing and managing all the gathered data. It must support queries from the Workload Analyzer so to expose the statistical data for subsequent processing/analysis. The Log Service could be implemented in several manners, in terms of both the underlying data storage technology and the selected deployment (centralized vs distributed). As for the first aspect, different solutions could be envisaged in order to optimize access operations depending on, e.g. suited tradeoffs between performance and access flexibility. This is also related with the data model ultimately supported by the Log Service, which might be a traditional relation model or, alternatively, a <key,value> model. Further, the Log Service could maintain the data onto a stable storage support or within volatile memory, for performance vs reliability tradeoffs. The above aspects are strictly coupled with the functionality/architecture of the Workload Analyzer, which could be designed to be implemented as a geographically distributed process in order to better fit the WPM deployment (hence taking advantage from data partitioning and distributed processing).

3.1 Implementation of infrastructure oriented probes

Infrastructure oriented probes have been developed to fulfil User Requirements [38-40]. The design and prototypal implementation of the infrastructure oriented probes has been tailored to the acquisition of statistical data in relation to (virtualized) hardware resources with no binding on a specific Operating System. This has been done by implementing the JAVA code associated with the probe prototypes on top of the SIGAR cross-platform JAVA based library (version 1.6.4) [2]. Infrastructure oriented probes are in charge of gathering statistical data on the usage of:

- CPU;
- RAM;

- Disk(s);
- Network Interfaces.

Below, we provide details on the whole set of monitored parameters for each resource of interest.
1) CPU (per core):

%user,

%system,

%idle.

2) RAM:

kB free memory,

kB used memory.

3) Network interfaces:

total incoming data bytes,

total outgoing data bytes,

inbound bandwidth usage,

outbound bandwidth usage.

4) Disk:

%Free space (kB),

%Used space (kB),

mountPoint or Volume.

For all of the above four resources, the associated sampling process can be configured with differentiated timeouts whose values can be selected on the basis of the time-granularity according to which the sampled statistical process is expected to exhibit non-negligible changes.

3.2 Implementation of data platform oriented probes

Data platform oriented probes are in charge of collecting statistical data in support of User Requirements [41-43] and [45]. The prototypal implementation of the data platform oriented probes has been extensively based on the JMX framework, which is explicitly oriented to support audit functionalities for JAVA based components. Essentially, each data platform oriented probe implements a JMX client, which can connect towards the JMX server running within the process where the monitored component

resides. Then via the JMX standard API, the probe retrieves the audit information internally produced by the monitored JAVA component in relation to its own activities.

Currently, we have developed a prototype data platform probe that accesses the internal audit system of single Infinispan caches, in order to sample the below reported list of parameters:

NumPuts - the number of requested put operations;

Commits - the number of committed transactions;

CommitTime - the average latency for commit operations;

Prepares - the number of prepare operations;

Rollbacks - the number of transaction rollbacks;

WriteTxCommits - the number of committed write transactions;

WriteTxInPrepare - the current amount of prepared write transactions;

WriteTxStarted - the current amount of started write transactions;

AvgLocalReadOnlyExecutionTime - the average latency for a read only transaction exclusively accessing locally cached objects;

AvgLocalWriteTxExecutionTime - the average latency for a write transaction entailing read operations exclusively accessing locally cached objects (commit phase excluded).

We underline that the list of the above reported parameters, which are currently collected by the JMX client embedded within the data platform oriented probe prototype, is not meant to exhaustively cover the set of data platform parameters to be finally monitored within Cloud-TM. Specifically, such a final set will be defined while finalizing the Autonomic Manager architecture, since it will depend on the specific optimization policies supported by the Autonomic Manager. As an example, statistical data related to layers underlying Infinispan (e.g. the Group Communication layer) might be needed in order to support optimization policies explicitly taking into account latency and scalability aspects across the whole stack of layers forming the Cloud-TM platform. Further, the final set of data platform parameters to be monitored by WPM will also depend on the specific QoS parameters supported by Cloud-TM, which will be ultimately defined by finalizing the instantiation of the QoS API exposed to the overlying customer applications. Nevertheless, the current structure of the WPM prototype will support the collection of additional parameters via trivial extensions, not impacting the architectural organization described within this document.

3.3 Startup and base message tagging rules

Particular care has been taken in the design of the startup phase of WPM components, in relation to the fact that each probe is requested to gather statistical data within a highly dynamic environment, where the set of monitored components (either belonging to the infrastructure or to the data platform) and the related instances can vary over time. By deliverable D2.1, the Cloud-TM Autonomic Manager will rely on a Repository of Tunable Components, where an XML description for each component currently taking part to the Cloud-TM platform is recorded at component startup time.

In the design of the WPM prototype, we rely on the Autonomic Manager repository, by exploiting it as a registry where each probe can automatically retrieve information allowing it to univocally tag each measurement message sent to the Lattice consumer with the identity of the corresponding monitored component instance, as currently maintained by the registry. This will allow supporting a perfect matching between the measurement message and the associated instance of component, as seen by the Autonomic Manger at any time instant. Such a process has been supported by embedding within Lattice probes a sensing functionality, allowing the retrieval of basic information related to the environment where the probe is activated (e.g. the IP number of the VM hosting that instance of the probe), which has been coupled with a matching functionality vs the registry in order to both:

(a) retrieve the ID of the currently monitored component instance;

(b) retrieve information needed to correctly carry out the monitoring task, in relation to the target component instance.

Such a behavior is shown in Figure 2, where the interaction with the registry is actuated as a query over specific component types, depending on the type of probe issuing the query (an infrastructure oriented probe will query the registry for extracting records associated with VM instances, while a data platform oriented probe will query the registry for extracting records related to the specific component it is in charge of).

As for point (b), data platform probes rely on the use of JMX servers exposed by monitored components. Hence, the information requested to correctly support the statistical data gathering process entails the address (e.g. the port number) associated with the JMX server instance to be contacted. The information associated with point (b) is a "don't care" for infrastructure oriented probes since they do not operate via any intermediary (e.g. JMX server) entity.

Given that the registry has not already been developed, and will be the object of future deliverables, the current prototype of the WPM emulates the registry access via stubs reading name/value records from the file system. The accessed files can be populated according to the specific needs while installing the prototype, as it will be specified via the README.

3.4 Implementation of the Optimized-transmission Service

In the current prototypal implementation, the optimized-transmission service has been implemented by relying on the use of zip and SSL-based file transfer functionalities.

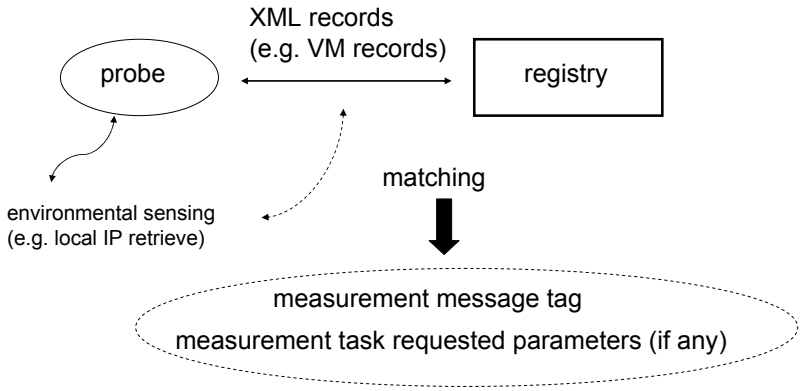

Figure 2: Interaction between the Probes and the Registry.

Each data stream portion assembled by the Lattice consumer is locally logged within a file, which is then zipped and sent towards the Log Service front-end via SSL. Exactly-once transmission semantic has been guaranteed via well known retransmission/filtering schemes, which have been based on a univocally determined name for each transmitted zipped file. Specifically, each Lattice consumer is univocally identified via a $consumer\_ID$, which has been used to generate unique file names in the form

$$consumer\_ID + start\_timestamp + end\_timestamp$$

where start and end timestamp values within the file name identify the time interval during which the statistical data have been gathered by the consumer. These timestamp values are determined by exploiting the local clock accessible at the consumer side via the `System.currentTimeMillis()` service.

3.5 Implementation of the Log Service

The Log Service prototype has been implemented by relying on Infinispan, specifically by instantiating it as an Infinispan application that parses the input streams received from the Lattice consumer, and performs put operations on top of an Infinispan cache instance. The keys used for put operations correspond to message tags, as defined by the Lattice producer and its hosted probes. In particular, as explained above, each probe tags measurement messages with the unique ID associated with the monitored component. This ID has been used in our implementation to determine a unique key, to be used for a put operation, formed by:

$$component\_ID + type\_of\_measure + measure\_timestamp$$

where the $type\_of\_measure$ identifies the specific measure carried out for that component (e.g. CPU vs RAM usage in case of a VM component), and the value expressed by $measure\_timestamp$ is again generated via the local clock accessible by the probe instance producing the message. According to this prototype implementation, the Log Service exposes to the Workload Analyzer Infinispan native <key,value> API, which does not prevent the possibility of supporting a different API in future releases, depending on the needs associated with the interaction with the Workload Analyzer.

4 Setting up the WPM prototype

In this section we describe the content of the package and the necessary steps to compile and run the WPM prototype.

4.1 Structure and Content of the Package

The content of the package is structured as follows

```
infinispan_test
  | _test_infinispan.jar
wpm
  | _config
  | _lib
  |  log
  | _src
        | _eu.cloudtm.wpm
              | _consumer
              | _hw_probe
              | _logService
              | _main
              | _producer
              | _sw_probe
        | _eu.reservoir.monitoring
doc
  | _D3.1-CompanionDocument.pdf
```

- The *infinispan_test* folder contains a simple test application that uses an Infinispan cache; this sample application exposes some relevant Key Performance Indicators of Infinispan via the MBean sever, which are acquired by WPM.
- The *wpm* folder contains the WPM system's source code, scripts and configuration files; the WPM system is composed by three modules
  - Log Service: this module logs the collected statistics within an Infinispan cache; its configuration file is *config/log_service.config*, while the one for the Infinispan instance is located within the *config* folder and its name is specified within the *log_service.config* file.
  - Consumer: its configuration file is *config/resource_consumer.config*.
  - Producer: its configuration file is *config/resource_controller.config*.
- The *doc* folder contains a textual document concerning the content of the package.

4.2 Compile the WPM prototype

1. The compile process requires the ANT program installed on the system. Decompress the zip file and, using the command line, locate the control within the folder *wpm*

2. In this folder run the command to clean the (possibly) previous project builds: *ant clean*

3. In order to compile the application, run the command: *ant compile*. The results of the execution should be the creation of the build folder that contains all the .class of the application.

4. The run scripts require the generation of an executable jar file. To do that run the command *ant jar*. If success, in the *wpm* folder a new jar file, called *wpm.jar*, should be appeared.

5. Now the application is compiled and ready to execute.

4.3 Setting up the WPM prototype

1. Since all the modules communicate via sockets, network firewall rules MUST BE configured in order not to drop the requests/packets through the ports specified by the configuration files.

2. For a correct startup, the modules should be activated in the following order: Infinispan_Test, Log Service,Consumer, Producer.

3. The command to run the application is:

   ```
   java -cp . -Dcom.sun.management.jmxremote.port=9999 -Dcom.sun.management.
   jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false
   -jar test_infinispan.jar
   ```

   This is contained within the script file *run_test_infinispan.sh*

4. The command to run the Log Service module is:

   ```
   java -cp . -Djavax.net.ssl.keyStore=config/serverkeys -Djavax.net.ssl.keyStorePassword=
   cloudtm -Djavax.net.ssl.trustStore=config/serverkeys -Djavax.net.ssl.trustStorePassword=
   cloudtm -jar wpm.jar logService
   ```

   This is contained within the script file *run_log_service.sh*

5. The command to run the Consumer module is:

   ```
   java -cp . -Djavax.net.ssl.trustStore=config/serverkeys -Djavax.net.ssl.trustStorePassword=
   cloudtm -Djavax.net.ssl.keyStore=config/serverkeys -Djavax.net.ssl.keyStorePassword=
   cloudtm -jar wpm.jar consumer
   ```

   This is contained within the script file *run_consumer.sh*

6. The command to run the Producer module is:

   java -cp . -Djava.library.path=lib/ -jar wpm.jar producer

   This is contained within the script file *run_producer.sh*

References

[1] http://www.reservoir-fp7.eu/index.php?page=open-source-code

[2] http://www.hyperic.com/products/sigar