
# Cloud-TM

Specific Targeted Research Project (STReP)

Contract no. 257784

## Deliverable D4.4:
## Prototype of the Cloud-TM Platform

**Date of preparation:** 31 July 2013
**Start date of project:** 1 June 2010
**Duration:** 38 Months

## Contributors

Emmanuel Bernard, RED HAT
Daniele Calisi, Algorithmica
João Cachopo, INESC-ID
Maria Couceiro, INESC-ID
Fabio Cottefoglie, ALGORITHMICA
Diego Didona, INESC-ID
Nuno Diegues, INESC-ID
Pierangelo Di Sanzo, INESC-ID
Sérgio Fernandes, INESC-ID
Sanne Grinovero, RED HAT
Mircea Markus, RED HAT
Sebastiano Peluso, CINI
Francesco Quaglia, CINI
Paolo Romano, INESC-ID
Pedro Ruivo, RED HAT
Manik Surtani, RED HAT
Francesco Zaratti, Algorithmica
Vittorio Ziparo, Algorithmica

---

# Table of Contents


**1 Introduction** **4**
- 1.1 Relationship with other deliverables . . . . . 4

**2 Structure of the Cloud-TM Data Platform Package** **5**
- 2.1 Object Grid Data Mapper . . . . . 5
- 2.2 Search API . . . . . 7
- 2.3 Distributed Execution Framework . . . . . 8
- 2.4 Distributed Transactional Data Grid . . . . . 8

**3 Structure of the Cloud-TM Autonomic Manager Package** **10**
- 3.1 WPM . . . . . 10
- 3.2 QoS Manager . . . . . 11
- 3.3 Workload Analyzer and Adaptation Manager . . . . . 12

**4 Setting up the prototype and the example applications** **15**
- 4.1 Structure and content of the package . . . . . 15
- 4.2 Installing and running the example applications . . . . . 17
  - 4.2.1 Preparing Fénix Framework . . . . . 17
- 4.3 Examples that demonstrate the use of the components . . . . . 20
  - 4.3.1 Demo 1: A simple Java application . . . . . 20
  - 4.3.2 Demo 2: Using the Search API . . . . . 22
  - 4.3.3 Demo 3: Running on a cluster . . . . . 24
  - 4.3.4 Demo 4: Persisting state . . . . . 26
- 4.4 Demo 5: Pluggable Collections and Concurrency-Friendliness . . . . 27
- 4.5 Demo 6: Co-location of Data . . . . . 28
- 4.6 Demo 7: Indexed Domain Relations . . . . . 30
- 4.7 Demo 8: L2 Cache . . . . . 31
- 4.8 Running the pre-compiled examples . . . . . 31
- 4.9 Installing and running the demo applications . . . . . 33
  - 4.9.1 Demo 1 - Automated Elastic Scaling (based on ANN) . . . . 33
  - 4.9.2 Demo 2 - Automatic switching among replication protocols (based on MorphR) . . . . . 39
  - 4.9.3 Demo 3 - What-if analysis (Based on TAS) . . . . . 41
  - 4.9.4 Demo 4 - QoS negotiation (based on DAGS) . . . . . 45
  - 4.9.5 Demo 5 - Self-tuning data placement (based on AUTOPLACER) 48

**5 Licensing** **52**

**References** **53**

# 1 Introduction

This document is the companion document of Deliverable D4.4, *Final Prototype of the Cloud-TM Platform*. The prototype is shipped as a ready-to-go virtual machine image, containing source code, pre-compiled binaries, a copy of the user documentation of the main components of the platform, as well as a set of examples aimed at allowing to test and get familiarized with the platform.

This document has the following main goals:

- providing references to detailed documentation on the usage and configuration of the various modules of the Cloud-TM Platform;
- describing the structure of the software packages presented in the architecture available in D4.6;
- providing instructions on how to compile, deploy and run the example applications included in the software package.

The structure of this document is the following. We start, in Sections 2 and 3 by providing references to interfaces and documentation manuals for, respectively, the Data Platform and Autonomic Manager. We then describe the contents of the packages in Section 4. This includes providing detailed instructions on how to install it and test it by running the demo applications included in the packages. Finally, in Section 5, we provide an overview and general guidelines concerning the licensing schemes associated with the use and distribution of the source code developed in the context of the Cloud-TM project.

## 1.1 Relationship with other deliverables

The Cloud-TM platform prototype represents the convergence of the results achieved throughout the project, and therefore as relationships with almost every deliverable of the project.

However, in order to understand the concepts mentioned in this deliverable, the reader should first refer, in particular, to Deliverable D4.6, *Final Architecture Report*, which describes in detail the various modules of which it is composed, as well as documents the APIs offered to programmers.

# 2 Structure of the Cloud-TM Data Platform Package

In this section we briefly recapitulate the architecture of the Cloud-TM Platform, presented in much more detail in D4.6. Figure 1 presents the high-level overview of the components composing it. In the following sections we provide references for the detailed documentation and manuals of the components in the Data Platform module. Section 3 shall analogously present other documentation for the Autonomic Manager.

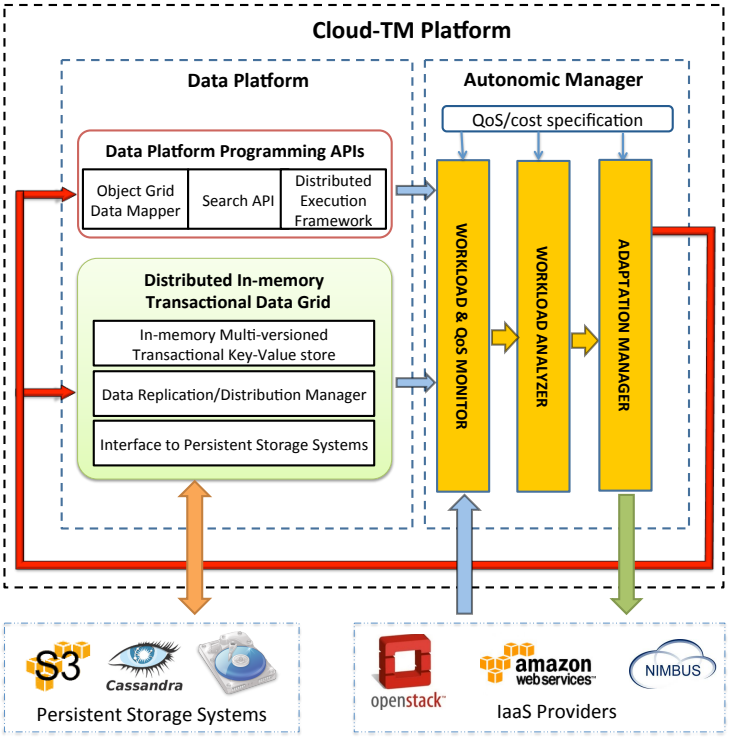


Figure 1: Cloud-TM high-level architecture.

## 2.1 Object Grid Data Mapper

The *Object Grid Data Mapper (OGDM)* module is responsible for handling the mapping from the object-oriented data model used by the programmer to the $< key, value >$ pair data model, which is employed by the underlying Distributed Transactional Data Grid platform. The *Fénix Framework (FF)* provides a concrete implementation of this module.

**Pointers to source code and additional documentation:**

- The source code of FF is available at the following URL:

  ```
  https://github.com/cloudtm/fenix-framework
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/DataPlatform/src/Fenix-Framework/
  ```

- The DML reference manual is available at the following URL:

  ```
  https://github.com/cloudtm/fenix-framework/blob/
  cloudtm/docs/dml-reference.md
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/DataPlatform/docs/Fenix-Framework/dml_
  reference.pdf
  ```

- The source code of Hibernate OGM is available at the following URL:

  ```
  https://github.com/cloudtm/hibernate-ogm
  ```

  or directly in the Hibernate OGM project's web page:

  ```
  https://github.com/hibernate/hibernate-ogm
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/DataPlatform/src/Hibernate-OGM/
  ```

- The Hibernate OGM reference manual is available at the following URL:

  ```
  http://docs.jboss.org/hibernate/ogm/4.0/reference/
  en-US/html/
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/DataPlatform/docs/Hibernate-OGM/hibernate_
  ogm_reference.pdf
  ```

We further highlight additional documentation to the following features in the OGDM:

```
http://fenix-framework.github.io/CloudTM.html
```

- Pluggable Collections:

  ```
  http://fenix-framework.github.io/Collections.html
  ```

- Indexation in the Domain Relations:

```
http://fenix-framework.github.io/Indexes.html
```

- Locality Hints:

```
http://fenix-framework.github.io/LocalityHints.html
```

- Programmer-defined Caching using the L2 Cache:

```
http://fenix-framework.github.io/L2Cache.html
```

- Data Access Patterns:

```
http://fenix-framework.github.io/DAP.html
```

## 2.2 Search API

The Cloud-TM Data Platform offers the programmers a Search API that allows to query the data maintained in the Data Platform using an object-oriented query language. To this end, the Cloud-TM Data Platform relies on Hibernate Search, an indexing and querying technology that offers APIs operating directly at the domain object level (aka POJO).

**Pointers to source code and additional documentation:**

- The source code of Hibernate Search is available at the following URL:

```
https://github.com/cloudtm/hibernate-search
```

  or directly in the Hibernate Search project's web page:

```
https://github.com/hibernate/hibernate-search
```

  as well as in the deliverable's virtual machine image at the path :

```
~cloudtm/DataPlatform/src/Hibernate-Search/
```

- The Hibernate Search reference manual is available at the following URL:

```
http://docs.jboss.org/hibernate/search/4.2/reference/
en-US/html/
```

as well as in the deliverable's virtual machine image at the path :

```
~cloudtm/DataPlatform/docs/Hibernate-Search/
hibernate_search_reference.pdf
```

## 2.3 Distributed Execution Framework

The Distributed Execution Framework aims at providing a set of abstractions to simplify the development of parallel applications, allowing ordinary programmers to take full advantage of the processing power available by the set of distributed nodes of the Cloud-TM platform without having to deal with low level issues such as load distribution, thread synchronization and scheduling, fault-tolerance.

**Pointers to source code and additional documentation:**

- The source code of Distributed Execution Framework is available at the following URL:

  ```
  https://github.com/cloudtm/infinispan/tree/cloudtm_
  v3/core/src/main/java/org/infinispan/distexec
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/DataPlatform/src/Infinispan/core/src/main/
  java/org/infinispan/distexec/
  ```

- The Distributed Execution Framework reference manual is available at the following URL:

  ```
  https://docs.jboss.org/author/display/ISPN/
  Infinispan+Distributed+Execution+Framework
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/DataPlatform/docs/Infinispan/infinispan_
  reference.pdf
  ```

## 2.4 Distributed Transactional Data Grid

The backbone of the Cloud-TM data platform is represented by Infinispan, a highly scalable, in-memory distributed key-value store with support for transactions. Born as an open-source project sponsored by Red Hat, Infinispan has been selected as the reference Distributed Transactional Data Grid platform for the Cloud-TM project. This choice has led to a close collaboration between the teams of Red Hat and of the academic partners of the Cloud-TM project, and to the integration in Infinispan of highly innovative data management algorithms and self-tuning mechanisms.

**Pointers to source code and additional documentation:**

- The source code of Distributed Transactional Data Grid is available at the following URL:

  ```
  https://github.com/cloudtm/infinispan
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/DataPlatform/src/Infinispan/
  ```

- The Distributed Transactional Data Grid reference manual is available at the following URL:

  ```
  https://docs.jboss.org/author/display/ISPN/User+Guide
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/DataPlatform/docs/Infinispan/infinispan_
  reference.pdf
  ```
  and
  ```
  ~cloudtm/DataPlatform/docs/Infinispan/infinispan_ext_
  reference.pdf
  ```

# 3 Structure of the Cloud-TM Autonomic Manager Package

In the following sections we provide references for detailed documentation and manuals of the components included in Autonomic Manager module (see Fig. 1 depicting the Cloud-TM platform).

## 3.1 WPM

The Workload and Performance Monitor (WPM) is the component aimed at gathering statistics provided by differentiated layers of the Cloud-TM platform; these layers include both the hardware level and the logical resources (e.g. Object Grid Data Mapper, Distributed In-memory Transactional Data Grid). In addition it implements functionality for statistics aggregation and it makes available gathered data to all the other components of the Autonomic Manager via both pull and push mechanisms. The latter is implemented by means of an adapter, namely the Workload Monitor Connector, designed as a plug-in for all the other components.
The WPM component is built on top of the Lattice framework [1].

**Pointers to source code and additional documentation:**

- The source code of the Workload and Performance Monitor (WPM) is available at the following URL:

  `https://github.com/cloudtm/wpm`

  as well as in the deliverable's virtual machine image at the path :

  `~cloudtm/AutonomicManager/src/WPM/`

- The source code of the Workload Monitor Connector is available at the following URL:

  `https://github.com/cloudtm/Workload_Monitor_Connector`

  as well as in the deliverable's virtual machine image at the path :

  `~cloudtm/AutonomicManager/src/WorkloadMonitorConnector/`

- The source code of the Lattice project is available at the following URL:

  `https://github.com/cloudtm/LatticeCloudTM`

  as well as in the deliverable's virtual machine image at the path :

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

```
~cloudtm/AutonomicManager/src/LatticeCloudTM/
```

- The Workload and Performance Monitor reference manual is available at the following URL:

  ```
  http://www.gsd.inesc-id.pt/~romanop/files/
  deliverables/D3_1.pdf
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/AutonomicManager/docs/wpm/D3_1_Prototype_of_
  the_Workload_Monitor.pdf
  ```

## 3.2 QoS Manager

The Quality of Service (QoS) Manager offers supports for QoS and cost management in the Autonomic Manager. It is used for allowing the customers to provide information about the application logic (to be deployed on top of the Cloud-TM platform): the associated expected workload, its features and the expected performance values to be matched within the Service Level Agreement (SLA). Furthermore it is responsible for informing the customers about the existence of an operating configuration of the Cloud-TM platform capable of satisfying that SLA, and at what cost.

**Pointers to source code and additional documentation:**

- The source code of the Quality of Service (QoS) Manager is available at the following URL:

  ```
  https://github.com/cloudtm/QoS
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/AutonomicManager/src/QoS/
  ```

- The Quality of Service (QoS) Manager reference manual is available at the following URL:

  ```
  http://cloudtm.ist.utl.pt/cloudtm-deliverables-pdf/
  Cloud-TM_Autonomic_Manager-D3.4.pdf
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/AutonomicManager/docs/qos/D3_4_Prototype_of_
  the_Autonomic_Manager.pdf
  ```

## 3.3 Workload Analyzer and Adaptation Manager

The Workload Analyzer and Adaption Manager (AM) is the main component of the Autonomic Manager aimed at implementing self-optimization mechanisms for the Cloud-TM Data Platform. In particular it offers solutions for (i) identifying the optimal values of a set of key configuration parameters/protocols, i.e. the scale of the Data Platform, the replication degree of data stored by the platform, the type of the replication protocol among the one offered by the Data Platform, and (ii) optimizing the data access locality of Cloud-TM applications, i.e. automatically identifying the optimal placement of data replicas across the instances of the Data Platform and defining locality-aware load distribution policies.

The self-optimization logic of the AM is hosted by a suit of performance prediction tools based on diverse prediction methodologies installable in the AM as plug-ins or directly integrated in the Data Platform and coordinated by the AM, e.g. the AUTOPLACER optimizer [1] or the MORPHR's run-time protocol switching mechanisms [2]. The ones offered by the Cloud-TM platform are the following:

- the Transactional Auto-Scaler (TAS) predictor;
- the MORPHR predictor;
- the Artificial Neural Network (ANN) predictor;
- the DAta Grid Simulation (DAGS) framework.
- the AUTOPLACER scheme.

**Pointers to source code and additional documentation:**

- The source code of the Workload Analyzer and Adaption Manager (AM) is available at the following URL:

  ```
  https://github.com/cloudtm/AdaptationManager
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/AutonomicManager/src/AdaptationManager
  ```

- The source code of the Transactional Auto-Scaler (TAS) is available at the following URL:

  ```
  https://github.com/cloudtm/TAS
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/AutonomicManager/src/TAS
  ```

- The Transactional Auto-Scaler (TAS) reference is available in the deliverable's virtual machine image at the path :

```
~cloudtm/AutonomicManager/docs/tas/tas_reference.pdf
```

- The source code of the MorphR is available at the following URL:

```
https://github.com/cloudtm/MorphR
```

  as well as in the deliverable's virtual machine image at the path :

```
~cloudtm/AutonomicManager/src/MorphR
```

- The MorphR reference is available in the deliverable's virtual machine image at the path :

```
~cloudtm/AutonomicManager/docs/morphr/morphr_
reference.pdf
```

- The source code of the Artificial Neural Network (ANN) is available at the following URL:

```
https://github.com/cloudtm/ANN
```

  as well as in the deliverable's virtual machine image at the path :

```
~cloudtm/AutonomicManager/src/ANN
```

- The ANN reference is available in the deliverable's virtual machine image at the path :

```
~cloudtm/AutonomicManager/docs/ann/ann_reference.pdf
```

- The source code of the DAta Grid Simulation (DAGS) framework is available at the following URL:

```
https://github.com/cloudtm/DAGS
```

  as well as in the deliverable's virtual machine image at the path :

```
~cloudtm/AutonomicManager/src/DAGS
```

- The DAGS reference is available in the deliverable's virtual machine image at the path :

```
~cloudtm/AutonomicManager/docs/dags/dags_reference.
pdf
```

- The source code of the DAGS Connector used as an adapter to plug the DAGS framework in the Adaptation Manager is available at the following URL:

  ```
  https://github.com/cloudtm/DAGSConnectorAM
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/AutonomicManager/src/DAGSConnectorAM
  ```

- The source code of the AutoPlacer is available at the following URL:

  ```
  https://github.com/cloudtm/infinispan/tree/cloudtm_v3
  ```

  as well as in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/AutonomicManager/src/Autoplacer
  ```

- The AutoPlacer reference is available in the deliverable's virtual machine image at the path :

  ```
  ~cloudtm/AutonomicManager/docs/autoplacer/autoplacer_
  reference.pdf
  ```

# 4 Setting up the prototype and the example applications

The prototype has been made publicly available both on the Cloud-TM website, at the URL:
`http://cloudtm.ist.utl.pt/cloudtm/cloudtm-image/`

as well as on the GitHub webpage of the Cloud-TM project, at the URL:
`https://github.com/cloudtm/final-prototype`

and on the project's website, in the deliverables section:
`https://www.cloudtm.eu/deliverables`

In this section we describe the content of the whole software package and the necessary steps to compile it, configure it and install it.

## 4.1 Structure and content of the package

The package is distributed as a virtual machine (VM) image of type qcow2, ready to be deployed on KVM-based IaaS platforms, such as OpenStack. Of course the image can be readily converted to run on other hypervisors (e.g., XEN) using open-source tools like qemu-image[2].

The VM comes with a default user (the `cloudtm` user), with a password-less account. The contents of this software package are stored in the home directory of the `cloudtm` user, and are organized according to the following directory structure:

The *DataPlatform* folder contains the source code, documentation and examples of the preliminary prototype of the Cloud-TM Data Platform. Specifically:

- The *docs* folder contains the manual reference for each module of the platform and this companion document.
- The *examples* folder contains the source code of the example applications and zip files with the compiled code ready to be deployed.
- The *src* folder contains all the source code of each module belonging to the Cloud-TM platform.

[2] `http://www.qemu.org`

```
DataPlatform
    |-docs/
    |    |-Fenix-Framework/
    |    |    | dml_reference.pdf
    |    |    \-fenix-framework_reference.pdf
    |    |-Hibernate-OGM/
    |    |    \-hibernate_ogm_reference.pdf
    |    |-Hibernate-Search/
    |    |    \-hibernate_search_reference.pdf
    |    |-Infinispan/
    |    |    |-infinispan_reference.pdf
    |    |    \-infinispan_ext_reference.pdf
    |    |-JGroups/
    |    |    \-jgroups_reference.pdf
    |-examples/
    |    |-Demo1/
    |    |-Demo2/
    |    |-Demo3/
    |    |-Demo4/
    |    |-Demo5/
    |    |-Demo6/
    |    |-Demo7/
    |    \-Demo8/
    \-src/
         |-Fenix-Framework/
         |-Hibernate-OGM/
         |-Hibernate-Search/
         |-Infinispan/
         \-JGroups/
```

The *AutonomicManager* folder contains the source code, documentation and demos of the preliminary prototype of the Cloud-TM Autonomic Manager. Specifically:

- The *docs* folder contains the reference for each components of the module.
- The *demos* folder contains the source code of the demos applications.
- The *src* folder contains all the source code of each component belonging to the Cloud-TM Autonomic Manager.

```
AutonomicManager
    |-docs/
    |    |-tas/
    |    |    \-tas_reference.pdf
    |    |-morphr/
    |    |    \-morphr_reference.pdf
    |    |-ann/
    |    |    \-ann_reference.pdf
    |    |-dags/
    |    |    \-dags_reference.pdf
    |    |-autoplacer
    |    |    \-autoplacer_reference.pdf
    |    |-qos/
    |    |    \-qos_reference.pdf
    |    |-wpm/
    |    |    \-wpm_reference.pdf
    |-demos/
    |    |-Demo1-ANN/
    |    |-Demo2-MorphR/
    |    |-Demo3-TAS/
    |    |-Demo4-DAGS/
    |    \-Demo5-AUTOPLACER/
    |
    \-src/
         |-AdaptationManager/
         |-TAS/
         |-MorphR/
         |-ANN/
         |-DAGS/
         |-DAGSConnectorAM/
         |-Autoplacer/
         |-QoS/
         |-WPM
         |-WorkloadMonitorConnector/
         |-LatticeCloudTM/
```

## 4.2 Installing and running the example applications

In this Section we provide instructions on how to compile, configure and install the example applications shipped with the virtual machine, which are meant to allow application developers to familiarize with the programming API and the main deployment alternatives supported by the Cloud-TM Data Platform.

### 4.2.1 Preparing Fénix Framework

The Fénix Framework (FF) is already shipped in the VM image, ready to be used by the example applications (as we will describe more in detail in Section 4.3). However, for the sake of self-containment, in the following we present the steps that should be performed in order to download, configure and prepare the framework to be used by applications from scratch.

This module requires:

- Java 6

- Maven 3.0.3
- Git 1.7+

The OGDM module is always part of the user's application. In the case of FF, it is provided as a set of JAR files, which should be made available to the application both during compile[3] and runtime. In the remainder of this section we describe the general procedure to setup the FF to develop an application. We use *you* to refer to the *reader in the role of an application developer*.

The FF is developed using Maven, so if you use Maven to build your application, you can just depend on the FF artifacts that you need, by adding them to your pom.xml:

**Listing 1: Dependency in pom.xml**

```
<dependencies>
  <!-- add dependencies for the desired backends -->
  <dependency>
    <groupId>pt.ist</groupId>
    <artifactId>fenix-framework-backend-infinispan</artifactId>
    <version>2.10-cloudtm-SNAPSHOT</version>
  </dependency>
</dependencies>
```

These artifacts are available via the Cloud-TM Nexus repository, so you need to add it to your configuration:

**Listing 2: Repository in pom.xml**

```
<pluginRepositories>
  <pluginRepository>
    <id>cloudtm-plugin-repository</id>
    <url>http://cloudtm.ist.utl.pt:8083/nexus/content/groups/public/</url>
  </pluginRepository>
</pluginRepositories>

<repositories>
  <repository>
    <id>cloudtm-repository</id>
    <url>http://cloudtm.ist.utl.pt:8083/nexus/content/groups/public/</url>
  </repository>
</repositories>
```

Additionally, you will probably want to hook the dml-maven-plugin to your build process, so that your domain classes get properly generated and post-processed. This can be achieved by adding the plugin to the build phase.

**Listing 3: DML Maven Plugin in pom.xml**

```
<build>
  <plugins>
    <plugin>
      <groupId>pt.ist</groupId>
      <artifactId>dml-maven-plugin</artifactId>
```

[3] At compile time, the FF is used to generate the backend-specific code of the domain entities.

```
      <version>${fenixframework.version}</version>
      <configuration>
        <codeGeneratorClassName>
          ${fenixframework.code.generator}
        </codeGeneratorClassName>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>generate-domain</goal>
            <goal>post-compile</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

The fenixframework.code.generator property, shown in the previous listing, could be set in your properties section of the POM file, as per the following example:

**Listing 4: Code Generator for DML Maven Plugin in pom.xml**

```
<properties>
  <fenixframework.code.generator>
    pt.ist.fenixframework.backend.infinispan.InfinispanCodeGenerator
  </fenixframework.code.generator>
<!-- alternative value could be
pt.ist.fenixframework.backend.ogm.OgmCodeGenerator -->
</properties>
```

Just make sure that fenixframework.version property is set in accordance with the version used for the other FF modules. This ensures that you use a plugin that matches the version of the framework you are using.

The steps described above are all that is necessary to be able to develop using the FF. The Maven build system will automatically download the required artifacts and the plugins will hook to the correct phases of your build.

Additionally, you may opt to compile the FF from source. It can be downloaded from GitHub and packaged with:

```
$ git clone git://github.com/cloudtm/fenix-framework.git
$ cd fenix-framework
$ git checkout cloudtm
$ mvn package
```

The previous sequence produces the artifacts, i.e. one JAR file for each FF submodule. To use the FF in your application, these packaged JAR files are required. They can be installed to the local Maven repository with:

```
$ mvn install
```

If you use any system other than Maven, first make sure that the jars resulting from the execution of mvn package are visible in your application's build and runtime

classpaths, and then check for any additional dependencies. You can check for dependencies using the Maven dependency plugin[4] (even if you don't use Maven in your application):

```
$ mvn dependency:list
```

At this point your application should be set up correctly to integrate the FF in its build dependencies. **If you are not using Maven** to build you application, then you need to take care of two additional steps. The first is to run the DML Compiler[5] to generate the source base classes before compiling your own code, and the second is to run the post-processor[6] on your compiled classes. Both tools come available with the FF code.

Finally, when deploying your application ensure that FF and all of its dependencies are available in the CLASSPATH.

## 4.3 Examples that demonstrate the use of the components

The demonstration application uses a fictitious domain that deals with publishers, authors and books.

This section demonstrates one possible step-by-step approach to developing an application using Cloud-TM's Platform. It is intended to provide a sample of how such development could evolve. We present a sequence of demos and we integrate some platform feature on each one. This is not intended to be a comprehensive display of the capabilities of each individual element of the platform. Rather, it aims at providing an example of how they can all be combined.

For each demo we provide the instructions on how to run it, and we highlight the most relevant aspects shown.

### 4.3.1 Demo 1: A simple Java application

This demo demonstrates a trivial Java application using the abstractions provided by the OGDM module. The contents of this demo are shown below:

[4] `http://maven.apache.org/plugins/maven-dependency-plugin/`

[5] Java program `pt.ist.fenixframework.DmlCompiler`.

[6] Java program `pt.ist.fenixframework.core.FullPostProcessDomainClasses`.

```
Demo1
    |-pom.xml
    |-runexample.sh
    \-src/
        \-main/
            |-dml/
            |    \-books.dml
            |-java/
            |    \-test
            |        |-Author.java
            |        |-Book.java
            |        |-ComicBook.java
            |        |-MainApp.java
            |        |-Publisher.java
            |        \-ScifiBook.java
            \-resources/
                 |-fenix-framework.properties
                 |-infinispan.xml
                 |-log4j.properties
                 \-run.sh
```

The main application (file test .MainApp) simply creates a few instances of the domain entities and connects them to create an object graph in memory. This initialization is done in one single transaction.

**Listing 5: test.MainApp**

```
public static void main(String[] args) {
    //...
    initDomain();
    //...
}

@Atomic
public static void initDomain() {
    //...
}
```

The application uses managed transactions (via the @Atomic annotation) to operate on the domain entities. These transactions are mapped to calls to the corresponding underlying transactional system, according to the backend in use.

As the result of a successful commit, all the domain entities manipulated during the transaction are transparently mapped to the underlying key/value store.

In this demo we make use of the DML to model the domain (file books.dml), thus abstracting its concrete implementation. The OGDM module is responsible for providing the concrete mapping, while the programmer can switch backends without having to change his application code.

The following is an excerpt from the DML file books.dml, showing part of the domain model.

**Listing 6: books.dml**

```
package test;

class Book {
```

```
    String bookName;
    double price;
}

class Publisher {
    String publisherName;
}

class Author {
    String name;
    int age;
}

(...)

relation PublisherWithBooks {
    Publisher playsRole publisher;
    Book playsRole booksPublished {
        multiplicity *;
    }
}

relation AuthorsWithBooks {
    Author playsRole authors {
        multiplicity *;
    }
    Book playsRole books {
        multiplicity *;
    }
}
```

To run this demo change to demo1 directory and execute:

```
$ ./runexample.sh
```

It is possible to change the default backend by running with:

```
$ ./runexample.sh -ogm
```

In this case, instead of using the Direct backend, the same application runs on top of the Hibernate OGM backend. No more changes are required. If different backends have different configuration requirements (e.g. a different Infinispan configuration file), then this configuration can be set in the resource file that configures the FF runtime for each backend (e.g. fenix−framework−ogm.properties).

### 4.3.2 Demo 2: Using the Search API

This demo extends the previous one by using the Search API. The structural contents of this demo are the same as before. The following files have been changed:

```
Demo2
    \-src
        \-main
            \-java
                \-test
                    |-Author.java
                    |-Book.java
                    |-ComicBook.java
                    |-MainApp.java
                    |-Publisher.java
                    \-ScifiBook.java
```

And the following file has been added:

```
Demo2
    \-src
        \-main
            \-resources
                \-fenix-framework-hibernate-search.properties
```

To use the Search API we first needed to mark the properties of the domain entities that should be indexed. Using such annotations causes the automatic indexing of the corresponding domain objects when their indexed properties are updated. The following is an excerpt showing the annotations placed in Book class:

**Listing 7: Book.java**

```
package test;

import org.hibernate.search.annotations.Field;
import org.hibernate.search.annotations.Indexed;
import org.hibernate.search.annotations.IndexedEmbedded;

@Indexed
public class Book extends Book_Base {
    (...)
    @Field @Override
    public String getBookName() { return super.getBookName(); }

    @Field @Override
    public double getPrice() { return super.getPrice(); }

    @Override @IndexedEmbedded
    public test.Publisher getPublisher() { return super.getPublisher(); }

    @Override @IndexedEmbedded
    public java.util.Set<test.Author> getAuthors() {
        return super.getAuthors();
    }
}
```

Note that indexing is triggered by committing a transaction, so when the initialization operation from the previous demo completes, a transaction is committed, which causes the indexing of those entities. Then, we perform some queries over the newly created indexes. Below is the code that performs such queries. This code was added to the test.MainApp application.

**Listing 8: MainApp.java**

```
@Atomic
public static void doQueries() {
    logger.debug("Doing example queries. Configured " + AUTH_COUNT +
            " authors, " + PUB_COUNT + " publishers, and " + BOOK_COUNT +
            " books");
    logger.debug("Find Book300: " + performQuery(Book.class, "bookName",
            "book300"));
    logger.debug("Find Book3*3: " + performWildcardQuery(Book.class,
            "bookName", "book3*3"));
    logger.debug("Find ScifiBook3*3: " + performWildcardQuery(
            ScifiBook.class, "bookName", "book3*3"));
    logger.debug("Find Scifi Books by Auth0: " + performQuery(
            ScifiBook.class, "authors.id", getAuthorByName("Auth0")
                    .getExternalId()));
}

(...)

public static <T> Collection<T> performQuery(Class<T> cls, String field,
        String queryString) {
    ArrayList<T> matchingObjects = new ArrayList<T>();

    QueryBuilder qb = HibernateSearchSupport.getSearchFactory()
            .buildQueryBuilder().forEntity(cls).get();
    Query query = qb.keyword().onField(field).matching(queryString)
            .createQuery();
    HSQuery hsQuery = HibernateSearchSupport.getSearchFactory()
            .createHSQuery().luceneQuery(query)
            .targetedEntities(Arrays.<Class<?>>asList(cls));
    hsQuery.getTimeoutManager().start();
    for (EntityInfo ei : hsQuery.queryEntityInfos()) {
        matchingObjects.add((T) FenixFramework.getDomainObject(
                (String) ei.getId()));
    }
    hsQuery.getTimeoutManager().stop();

    return matchingObjects;
}
```

The newly added file is just the configuration required for Hibernate Search.

To run this demo change to demo2 directory and execute:

```
$ ./runexample.sh
```

It is possible to change the default backend by running with:

```
$ ./runexample.sh -ogm
```

Besides initializing the domain objects as before, the application now performs some queries over the indexed properties.

### 4.3.3 Demo 3: Running on a cluster

This demo extends the previous demo by allowing to use OGDM in multiple machines sharing a consistent state of the data. To change from a centralized application to a

distributed application, you only need to adapt the configuration files. The modified configuration files are the following:

```
Demo3
    \-src
        \-main
            \-resources
                |-ispn-repl.xml
                |-ipsn-dist.xml
                \-fenix-framework-hibernate-search.properties
```

In addition, you need to add a JGroups configuration file in order to the OGDM machines to communicate among them. Two files were added, one for Infinispan cluster (jgroups.xml) and one for other modules cluster (hs−jgroups.xml):

```
Demo3
    \-src
        \-main
            \-resources
                |-jgroups.xml
                \-hs-jgroups.xml
```

To extend Hibernate Search to a distributed environment, we set Infinispan to store the index meta-data and JGroups as a master node election. The resulting file is the following:

**Listing 9: fenix-framework-hibernate-search.properties**

```
hibernate.search.default.directory_provider = infinispan
hibernate.search.default.worker.backend = jgroups
hibernate.search.services.jgroups.configurationFile=hs-jgroups.xml
```

Finally, we need to extended Infinispan to a distributed environment. Infinispan supports two clustering modes: replicated (each node has a copy of all the data) and distributed (each node has a copy of some data). In this demo we provided two configuration files, namely ispn−repl.xml for replicated cache and ispn−dist.xml for distributed. Common to both clustering modes, you need to set the transport to be used by Infinispan, as referred previously, JGroups. The following lines were added to the configuration file:

**Listing 10: Transport configuration in Infinispan**

```
(...)
<transport clusterName="scenario-3-repl-cluster">
    <properties>
        <property name="configurationFile" value="jgroups.xml" />
   </properties>
</transport>
(...)
```

For the replicated cache, you need to add the following:

**Listing 11: ispn-repl.xml**

```
(...)
<clustering mode="r">
    <sync replTimeout="15000" />
    <stateTransfer fetchInMemoryState="false" />
</clustering>
(...)
```

And for a distrbuted cache, you should add the following:

**Listing 12: ispn-dist.xml**

```
(...)
<clustering mode="d">
    <sync replTimeout="15000" />
    <hash numVirtualNodes="10" numOwners="1" />
    <stateTransfer fetchInMemoryState="false" />
</clustering>
(...)
```

To run this demo change to demo3 directory and execute **one** of the following commands:

```
$ ./runexample.sh              #Direct Backend and Replicated cache
$ ./runexample.sh -ogm         #OGM Backend and Replicated cache
$ ./runexample.sh -dist        #Direct Backend and Distributed cache
$ ./runexample.sh -dist -ogm   #OGM Backend and Distributed cache
```

### 4.3.4 Demo 4: Persisting state

Finally, for the last demo, we will show how to configure Infinispan to store the data in a persistence storage. As previously referred, Infinispan supports multiple persistence storage but for simplicity we configured this demo to store the data in the local File-System. The only files changed were the following:

```
Demo4
    \-src
        \-main
            \-resources
                |-ispn-repl.xml
                \-ipsn-dist.xml
```

The persistence storage configuration is common to both clustering modes (replicated and distributed). The following lines were added to both configuration files:

**Listing 13: ispn-repl.xml**

```
(...)
<loaders passivation="false" shared="true" preload="false">
    <loader class="org.infinispan.loaders.file.FileCacheStore"
            fetchPersistentState="false"
            purgerThreads="3"
            purgeSynchronously="true"
            ignoreModifications="false"
            purgeOnStartup="true">
        <properties>
            <property name="location" value="/tmp/fs-store"/>
        </properties>
```

```
        <async enabled="true"
               flushLockTimeout="15000"
               threadPoolSize="5"/>
    </loader>
</loaders>
(...)
```

To run this demo change to demo4 directory and execute one of the following commands:

```
$ ./runexample.sh              #Direct Backend and Replicated cache
$ ./runexample.sh -ogm         #OGM Backend and Replicated cache
$ ./runexample.sh -dist        #Direct Backend and Distributed cache
$ ./runexample.sh -dist -ogm   #OGM Backend and Distributed cache
```

Besides doing the same thing as the previous demo, additionally all the data created by the application is stored in /tmp/fs−store.

## 4.4 Demo 5: Pluggable Collections and Concurrency-Friendliness

This demo uses a subset of the previous domain applications to illustrate the benefits of plugging in specific collections as backing data-structures for the relations between domain objects. For that we use the following files:

```
Demo5
    |-pom.xml
    |-runexample.sh
    \-src/
        \-main/
            |-dml/
            |    \-books.dml
            |-java/
            |    \-test
            |        |-Book.java
            |        |-MainApp.java
            \-resources/
                 |-fenix-framework.properties
                 |-infinispan.xml
                 \-log4j.properties
```

The idea of this main application is to populate four different relations with the same number of Books. The difference is that each relation is backed by a different implementation of a domain collection. For this, we merely specify the *collection* attribute in the DML file, as the following snippet demonstrates:

**Listing 14: books-sc5.dml**

```
class Book {
    String bookName;
    double price;
}

relation DRWithBooksOptGhost {
    .pt.ist.fenixframework.DomainRoot playsRole parentOptGhostTree;
    Book playsRole booksOptGhostTree {
        multiplicity *;
```

```
        collection pt.ist.fenixframework.adt.bplustree.BPlusTreeArrayGhost;
    }
}
```

In this case we can see that we specify the Domain Root of the application to be related with an arbitrary number of Books. We used four of these relations with each of the following collections:

- `BPlusTree`: typical B$^+$Tree implemented on top of DML and using a Java `TreeSet` to hold the contents of each tree node;
- `BPlusTreeArray`: optimized tree for faster sequential executions; the main optimization is the usage of a highly efficient native array-based serialization of contents for each tree node;
- `BPlusTreeGhost`: corresponds to the `BPlusTree`, but rather using the technique of Ghost Reads available in FF;
- `BPlusTreeArrayGhost`: corresponds to the `BPlusTreeArray` and to the `BPlusTreeGhost` together

To compare each collection, we measure the time it takes to populate them with an equal number of elements. Furthermore, this demo also showcases the benefits of using a concurrency-friendly collection to implement the relation between domain entities. This corresponds to the technique of Ghost Reads available in FF. For that reason, we created multiple threads in this demo so that we have concurrent accesses to the relations generating concurrency. We then show the throughput obtained when using a Ghost collection versus a normal one.

To run this demo, change to directory demo5 and execute one of the following:

```
$ ./runexample.sh              #Direct Backend
$ ./runexample.sh -ogm         #OGM Backend
```

As a result, you will be presented with the performance results corresponding to the usage of each collection, thus evidencing the benefits obtained in each case.

## 4.5 Demo 6: Co-location of Data

This demo showcases the benefits of using co-location of data in the distributed flavor of the Data Platform. For this purpose, we created a distributed demo with replication degree one and two machines. This required modifying the MainApp.java file, as well as the configurations as described for Demos 3 and 4. Finally, we changed the DML to contain two relations using both a normal collection and it's Co-located counter-part.

```
Demo6
    |-pom.xml
    |-runexample.sh
    \-src/
        \-main/
            |-dml/
            |    \-books.dml
            |-java/
            |    \-test
            |        |-Book.java
            |        |-MainApp.java
            \-resources/
                 |-fenix-framework.properties
                 |-infinispan.xml
                 |-hs-groups.xml
                 \-log4j.properties
```

In the following DML file, please note that usage of the Colocated collection:

**Listing 15: books-sc6.dml**

```
package test;

class Book {
    String bookName;
    double price;
}

relation DRWithBooksRandom {
    .pt.ist.fenixframework.DomainRoot playsRole parentRandom;
    Book playsRole booksRandom {
        multiplicity *;
        collection pt.ist.fenixframework.adt.bplustree.BPlusTree;
    }
}

relation DRWithBooksColocated {
    .pt.ist.fenixframework.DomainRoot playsRole parentColocated;
    Book playsRole booksColocated {
        multiplicity *;
        collection pt.ist.fenixframework.adt.bplustree.ColocatedBPlusTree;
    }
}
```

Finally, the MainApp.java application exercises the collection for some time and outputs the throughput for both the normal collection and the co-located collection. This illustrates that, for small collections that fit in one machine, this data placement can help improve performance over the random placement from the consistent hashing used by default.

To run this demo, change to directory demo6 and execute one of the following:

```
$ ./runexample.sh              #Direct Backend with distribution
$ ./runexample.sh -ogm         #OGM Backend with distribution
```

## 4.6 Demo 7: Indexed Domain Relations

This demo presents the usage of indexes in relations between domain entities. For this, we identify in the DML file one relation with an index, and a similar one without it. In fact, a collection with indexes has no overheads when accessed normally; still, we present them separately to avoid any concern that the index could be hampering the performance of the baseline collection without index. For this demo, we modified only the two following files with respect to Demo 5: MainApp.java and books.dml.

Note the usage of the index in the following DML snippet:

**Listing 16: books-sc7.dml**

```
package test;


class Book {
    String bookName;
    double price;
}

relation DRWithBooks {
    .pt.ist.fenixframework.DomainRoot playsRole parent;
    Book playsRole books {
        multiplicity *;
        collection pt.ist.fenixframework.adt.bplustree.BPlusTree;
    }
}

relation DRWithBooksIndexed {
    .pt.ist.fenixframework.DomainRoot playsRole parentIndexed;
    Book playsRole booksIndexed {
        multiplicity *;
        indexed by bookName;
        collection pt.ist.fenixframework.adt.bplustree.BPlusTree;
    }
}
```

In the event that the Books could not be indexed by name, because it was not a unique value, we could use the multiple-value index by annotating with *indexed by bookName #(*)* instead.

Then, we do a search either with or without indexes, as shown next:

**Listing 17: MainApp.java**

```
    @Atomic
    public static void doRead(String mode) {
        String val = "book" + Math.abs(ran.nextInt(BOOK_COUNT));
        DomainRoot domainRoot = FenixFramework.getDomainRoot();
        if (mode.equals("non-indexed")) {
            for (Book b : domainRoot.getBooks()) {
                if (b.getBookName().equals(val)) return;
            }
        } else {
            domainRoot.getBooksIndexedByBookName(val);
        }
    }
```

Briefly, we seek an existing Book by its name either by traversing all known books until we find it, or by using the indexation scheme built-in the relation collections using the automatically generated method *getBooksIndexedByBookName*.

To run this demo, change to directory demo7 and execute one of the following:

```
$ ./runexample.sh              #Direct Backend
$ ./runexample.sh -ogm         #OGM Backend
```

## 4.7 Demo 8: L2 Cache

This demo illustrates the programmer-defined caching through L2 Cache in FF. For this purpose, we once again resort to automatically generated methods, which are part of the Object-Oriented API in FF, to govern the accesses to domain objects either by the cache, or without it (the normal usage). The idea here is to use a distributed demo, where some remote accesses can happen normally, to exploit the L2 Cache to maintain most of the execution flow local to the machine. With this intent, we changed only the following files with respect to Demo 6: MainApp.java and books.dml.

A brief inspection of the MainApp.java file reveals the invocation of the method *getBooksRandomCached(false)* over the Domain Root when using the L2 Cache. As a result, this method does not force a miss in the L2 Cache, and uses its contents when available. This contrasts with the normal usage of *getBooksRandom()*. We then show the difference in throughput in this distributed demo for both the machines executing it.

To run this demo, change to directory demo8 and execute one of the following:

```
$ ./runexample.sh              #Direct Backend distributed
$ ./runexample.sh -ogm         #OGM Backend distributed
```

## 4.8 Running the pre-compiled examples

In Section 4.3 we demonstrated how to run the examples. However, some examples run in one machine whereas others use multiple processes to emulate different machines and, by following the steps mentioned in Section4.3, it is required to download and compile a large number of all the software packages of the Cloud-TM Data Platform. This operation takes approximately 1 hour on a commodity PC, at the time of writing.

For the users, who wish to experiment with the example applications without going through the building of the entire Cloud-TM Data Platform, we provide also zip files with the pre-compiled code using both ODGM backends currently available, namely the Direct and OGM Backends. The files with the suffix −ispn are compiled to use the Direct Backend and the −ogm are compiled to use the OGM Backend.

For Demos 1, 2, 5, and 7 you only need to unzip and execute run.sh:

```
$ unzip <scenarioX><-ispn or -ogm>.zip
$ cd <DemoX>
$ ./run.sh
```

For Demos 3, 4, 6, and 8 it is possible to deploy them in two different options: i) one machine and multiple processes (each one simulating a physical machine); and ii)

multiple machines with one process each. In both cases, you need to unzip the zip file and choose the clustering mode you want to (distributed or replicated cache) use by creating a link to the correct configuration:

```
$ unzip <scenarioX><-ispn or -ogm>.zip
$ cd <DemoX>
$ ln -sf <ispn-repl.xml or ispn-dist.xml> infinispan.xml
```

Now, to running in option i), perform the following steps (assuming that you want to run with $N$ processes and each process simulates a machine):

```
$ ./gossiprouter.sh -start
$ for node in {1..N}; do ./run.sh N > out_${node} &; done
```

This will trigger N processes in background where the process $i$ is writing the output to the file out_i.

In the case where you opt for option ii), you need to configure JGroups in order to allow it to find all the machines in the network. First, pick one machine to be the Gossip Router and start it. The Gossip Router is a well known machine that each machine tries to communicate with in order to join the cluster.

```
$ ./gossiprouter.sh -start
```

After that, edit the files jgroups.xml and hs−jgroups.xml and set the Gossip Router hostname in the <TCPGOSSIP> tag by replacing jgroups.bind_addr with the hostname or IP. Assuming your machine IP is 192.168.0.1, the new configuration will look like this:

**Listing 18: Set Gossip Router hostname in JGroups**

```
(...)
<TCPGOSSIP
        initial_hosts="192.168.0.1[12001]"
        num_initial_members="1"
        break_on_coord_rsp="true"
        stagger_timeout="350"
        timeout="2000"
        />
(...)
```

Only for Demo 4, you have to configure if the persistence storage is shared among the machines (for example, a single data base server) or local (for example, the local file-system). In this particular case, the File-System storage is local to each machine so you need to set the attribute shared=false in both Infinispan configuration files:

**Listing 19: Set File-Storage not shared**

```
(...)
<loaders passivation="false" shared="false" preload="false">
(...)
```

Finally, copy the folder generated to all the machines and execute the following in each machine:

```
$ ./run.sh <number of machines>
```

## 4.9 Installing and running the demo applications

In this Section we provide instructions on how to compile, configure and install the demo applications shipped with the virtual machine, which are meant to demonstrate the functionalities and capabilities of the Cloud-TM Autonomic Manager.

### 4.9.1 Demo 1 - Automated Elastic Scaling (based on ANN)

In this section we discuss how to install and run an example application demonstrating the usage and the actual run-time dynamics produced by the ANN component. For this example we adopt a distributed benchmark, named RadarGun Client Server, that executes TPC-C [3] transactional profiles against the Cloud-TM data platform in such a way that the TPC-C transactions can run as clients on separate virtual machines wrt the machines hosting the data platform. In particular the data platform is formed by a clustered set of Cloud-TM instances, each one of them running as a server on a single virtual machine and accepting transactional requests from the clients threads.

Every virtual machine executing a data platform instance is monitored by means of WPM/WA, which is responsible for gathering (via Consumers and Producers instances) and aggregating statistics (via a Log Service instance) produced by the data platform during the execution of the benchmark, and making available those statistics to all the components of the Autonomic Manager. In particular, for this purpose we adopt a push mechanism according to which the ANN component can register a subscription to WPM/WA on a set of virtual machines $S$, and it is notified whenever the WPM's Log Service component logs a sample of statistics produced by the Cloud-TM instances running on $S$.

When the ANN instance receives new statistics, it can decide to change the actual configuration of the Cloud-TM data platform, namely the number of data platform instances and the data replication degree, on the basis of the current workload, i.e. the average number of active transactional threads executing on the Cloud-TM data platform, with the purpose of maximizing the overall system throughput, i.e. the average number of committed transactions per second, or minimizing the transactional response time, i.e. the average time spent to successfully execute a transactional request. This change in configuration is executed by an actuator module we refer to as Autonomic Manager Actuator, which is responsible for (i) stopping subsets of instances belonging to the Cloud-TM data platform, (ii) starting new data platform instances, (iii) triggering a change of the data replication degree within the Cloud-TM cluster.

**Configuring and running the example**

The current example is already shipped in the VM image and it is ready to be executed, for less than a configuration process. However, for the sake of self-containment, we list in what follows the steps that should be performed in order to download, compile, configure and run the application.

The application requires:

- Java 6
- Maven 3.0.3

- Ant 1.8.4

The package has the following structure

```
Demo1-ANN
 |-ann/
 |-AutonomicManagerActuator/
 |-DataGridClient/
 |-DataGridServer/
 |-RadargunClientServer/
```

As a prerequisite to run the example application we need the WPM runtime and a WPM Connector to be used by the ANN component for registering subscriptions on and acquiring new statistics from WPM/WA. The Connector's source code is available at the following URL

```
https://github.com/cloudtm/Workload_Monitor_Connector
```

and it can be compiled using the following commands:

```
$ cd Workload_Monitor_Connector
$ ant
```

The WPM's source code is available at the following URL

```
https://github.com/cloudtm/wpm
```

and it can be compiled using the following commands:

```
$ cd wpm
$ ant
```

In order to compile ANN, the AutonomicManagerActuator and the DataGridServer modules, we have to enter their main folders and to execute the ant command:

```
$ cd ann
$ ant
$ cd ../AutonomicManagerActuator
$ ant
$ cd ../DataGridServer
$ ant
```

Since the ANN module needs to connect to the actuator as well as to WPM, we have to move the output of the AutonomicManagerActuator compilation, namely the AutonomicManagerActuator/AutonomicManagerActuator.jar, and the output of the Workload_Monitor_Connector compilation, namely Workload_Monitor_Connector/WPMConnector.jar to the ann/lib folder.

Given that the adopted benchmark runs the transactional threads as clients of the data platform servers, the RadargunClientServer depends on a client stub used to transparently implement the communication protocol between a benchmark thread and a server thread of the data platform. For this reason, right before compiling the RadargunClientServer, we need to compile and install in the local maven repository the DataGridClient stub according to the following steps:

```
$ cd DataGridClient
$ ant
```

```
$ mvn install:install-file -Dfile=InfinispanClient.jar
-DgroupId=simutools.infinispan.client -DartifactId=infinispan-client
-Dversion=1.0.0 -Dpackaging=jar
```

Then RadargunClientServer can be compiled via maven by executing the following commands:

```
$ cd RadargunClientServer
$ mvn install
```

A run of the example application is divided into two phases. During the first phase we run and populate an instance of the Cloud-TM data platform by creating the initial TPC-C data-set. During the second phase we (i) start-up the remaining instances of the data platform that can join the same cluster and share the same initial data-set, and (ii) execute the TPC-C benchmark on the platform.

From now on we suppose to have two types of virtual machines with two different roles. On a virtual machine of type $master$ we will run the WPM's Log Service, the Autonomic Manager Actuator and the ANN module, while on a virtual machine of type $slave$ we will run a benchmark client, or a data platform instance monitored by WPM. Furthermore, in this example, we consider to run one machine of type $master$ having $IPmaster$ as IP address; on the other hand we consider to run two sets of machines of type $slave$: the first set $S1$ having IP addresses $IPslave1$, $IPslave2$, $IPslave3$, $IPslave4$, which will host the data platform, and the second set $S2$ having IP addresses $IPslave5$, $IPslave6$, $IPslave7$, $IPslave8$, which will host the benchmark clients.

To configure the connection between the benchmark clients and the data platform instances we need the following line in the cacheprovider.properties configuration file (see Listing 20) of the RadargunClientServer folder.

**Listing 20: cacheprovider.properties configuration file**

```
remote IPslave1 IPslave2 IPslave3 IPslave4
```

**A. Data platform population**

To run the population phase of the benchmark, we have to execute a population client thread on an instance of the data platform. To this end, we execute the following commands in order to run a data platform instance on the machine $IPslave1$:

```
$ cd DataGridServer
$ ./bin/runServer.sh
```

Then we change the benchmark.xml configuration file contained in RadargunClientServer/target/distribution/RadarGun-1.1.0-SNAPSHOT as shown in Listing 21, and we execute the following commands on machine $IPmaster$:

```
$ cd RadargunClientServer/target/distribution/RadarGun-1.1.0-SNAPSHOT
$ ./bin/benchmark.sh IPslave5
```

**Listing 21: Benchmark configuration for TPC-C population**

```
<bench-config>
   <master bindAddress="${127.0.0.1:master.address}"
```

```
                        port="${2103:master.port}"/>
    <benchmark initSize="1" maxSize="${1:slaves}" increment="1">
        ...
        <TpccPopulation
            numWarehouses="1"
            cLastMask="0"
            olIdMask="0"
            cIdMask="0"
            populateOnOneCache="true"
            enablePopulate="true"/>

        <CsvReportGeneration/>
    </benchmark>

    ...

</bench-config>
```

At the end, to complete the population phase on all the four instances of the data platform (set $S1$), we execute again the runServer.sh command on the machines $IPslave2$, $IPslave3$, $IPslave4$. In this way the four instances automatically join the same cluster and they share the just populated initial data-set.

**B. WPM execution**

The WPM layer is used to monitor the data platform instances running on the set of machines $S1$ and to log the gathered statistics on machine $IPmaster$. For this reason we first run the WPM's Log Service on the machine $IPmaster$ by executing

```
$ cd wpm
$ ./log_service.sh start
```

Then, on each machine in the set $S1$ we properly set the LogService_IP_Address attribute in the wpm/config/resource_consumer.config file (see Listing 22) to guarantee that WPM's Consumers are able to connect to the Log Service, and we run the pair Consumer/Producer in the following way:

```
$ cd wpm
$ ./consumer.sh start
$ ./producer.sh start
```

**Listing 22: Consumer configuration**

```
LogService_IP_Address=IPmaster
```

**C. TPC-C benchmark execution**

We change the benchmark.xml configuration file contained in RadargunClientServer/target/distribution/RadarGun-1.1.0-SNAPSHOT as shown in Listing 23, and we execute the following commands on machine $IPmaster$:

```
$ cd RadargunClientServer/target/distribution/RadarGun-1.1.0-SNAPSHOT
$ ./bin/benchmark.sh IPslave5 IPslave6 IPslave7 IPslave8
```

**Listing 23: Benchmark configuration for TPC-C execution of four machines**

```
<bench-config>
   <master bindAddress="${127.0.0.1:master.address}"
                port="${2103:master.port}"/>
   <benchmark initSize="4" maxSize="${4:slaves}" increment="1">
      ...
      <TpccPopulation
                numWarehouses="1"
                cLastMask="0"
                olIdMask="0"
                cIdMask="0"
                populateOnOneCache="false"
                enablePopulate="false"/>
      <TpccBenchmark
                numOfThreads="8"
                perThreadSimulTime="300"
                arrivalRate="0.0"
                backoffTime="1.0"
                thinkTime="1.0"
                paymentWeight="5.0"
                orderStatusWeight="70.0"/>

      <CsvReportGeneration/>
   </benchmark>
    ...
</bench-config>
```

In this way we will run four clients processes with eight benchmark threads per client, each one executing 70% of TPC-C Order-Status transactions, 5% of TPC-C Payment transactions and 25% of TPC-C New-Order transactions.

### D. Actuator execution

Since the Actuator has to connect to the machines executing the data platform instances and it has to accept requests on a given ip:port end-point, we need to set the infinispanNodes, ip and port attributes in the AutonomicManagerActuator/actuator-server.properties file on the machine $IPmaster$ as shown in Listing 24

**Listing 24: Actuator configuration**

```
ip=IPmaster
port=8889
infinispanNodes=IPslave1 IPslave2 IPslave3 IPslave4
```

Afterwards we can run the Actuator on machine IPmaster as follows:

```
$ cd AutonomicManagerActuator
$ ./bin/runActuatorServer.sh
```

### E. ANN execution

To train the neural network implemented within ANN we have to build an initial training set by monitoring preliminary runs of the benchmark and adopting a range of different configurations, e.g. number of data platform instances, number of client

threads, data replication degree. All the collected data have to be moved to a directory that will be accessed by the ANN module in order to execute the initial training phase and whose path has to be specified within the configuration file of the ANN module. In Listing 6, we show and example of configuration file for the ANN module, which is used for specifying the following set of parameters:

- $clientNormalization$: it is the maximum number of client that send requests to the system plus one.
- $serverNormalization$: it is the maximum number of available server plus one.
- $replicationNormalization$: it is the maximum level of replication that can be used inside the system (N.B serverNormalization <= replicationNormalization).
- $throughputNormalization$: it must be greater than the maximum throughput that can be obtained by the system.
- $responseNormalization$: it must be greater than the maximum response time that can be obtained by the system.
- $trainingFilesPath$: it is the relative path where the logging files used for the ANN training phase must be copied.
- $minimalReplication$: it is the minimal replication degree that we want to use inside the system (N.B. $minimalReplication < replicationNormalization$, $0 < minimalReplication \leq maximumNodeNumber$).
- $maximumNodeNumber$: it is the maximum number of server nodes that we want to use inside the system (N.B.: $maximumNodeNumber = serverNormalization - 1$).
- $optimizationTarget$: the values of this parameter have to be chosen in the domain $\{Throughput, Response\}$, where
  - $Throughput$ is specified if we want to maximize the system throughput;
  - $Response$ is specified if we want to minimize the response time.
- $monitoredNodes$: the list of server nodes that can be used by the system (the cardinality of this list must be equal to $maximumNodeNumber$'s value).
- $actuatorServer$: the ip address of the machine that hosts the Actuator server process.
- $actuatorServerPort$: the port number on which the Actuator server process is accepting connections.

**Listing 25: ANN configuration**

```
clientNormalization = 33
serverNormalization = 5
```

```
replicationNormalization = 5
throughputNormalization = 2000
responseNormalization = 17000
trainingFilesPath = ./trainingFiles
minimalReplication = 2
maximumNodeNumber = 4
optimizationTarget=Throughput
monitoredNodes =IPslave1 IPslave2 IPslave3 IPslave4
actuatorServer=IPmaster
actuatorServerPort=8889
```

After starting up the ANN module, it will automatically build the training set, it will train the neural network and will start to control the system. The number of nodes and the replication degree of the platform will be automatically controlled by the actuator process, which receives new predictions each time new monitoring samples are available and then, if needed, it triggers the commands aimed at changing the platform configuration. To run the ANN module we have to execute the following commands:

```
$ cd ann
$ ./runANN.sh
```

**Outcome**

Figure 2 shows the system throughput, i.e. transactions committed per time unit, achieved by the Cloud-TM data platform during the execution of the TPC-C benchmark. In particular we run 32 clients threads on 4 different machines (i.e. IPslave5, IPslave6, IPslave7, IPslave8) that execute the TPC-C transactional logic on a cluster of 4 Cloud-TM data platform instances with data replication degree equals to 4 and under the ANN control. After 200 seconds since the start of the demo, the ANN component detects the current configuration to be sub-optimal and it triggers a configuration change to the Actuator process. In particular, ANN forecasts that the configuration with 2 data platform instances and data replication degree equals to 2 is optimal under the current workload and it forces that Actuator process to stop two of the running Cloud-TM instances and reduce the replication degree to 2. The result obtained after this reconfiguration is an increase by 43% of the overall system throughput.

### 4.9.2 Demo 2 - Automatic switching among replication protocols (based on MorphR)

In this section we discuss how to install and run an example application demonstrating the usage of MorphR when subjected to three very distinct workloads, showing that the system is able to recognize it is no longer using the optimal replication protocol and switch to the most appropriate one. We use the Radargun framework to generate these synthetic benchmarks, whose dynamics have already been explained in the beginning of Section 4.9.1.

The example starts with the system running a very low contention workload, especially suited for 2PC. Six minutes later, the workload is changed by increasing the contention, favouring the TOB protocol. Finally, in minute twelve we change the workload to the one with the highest contention, an optimal scenario for PB.

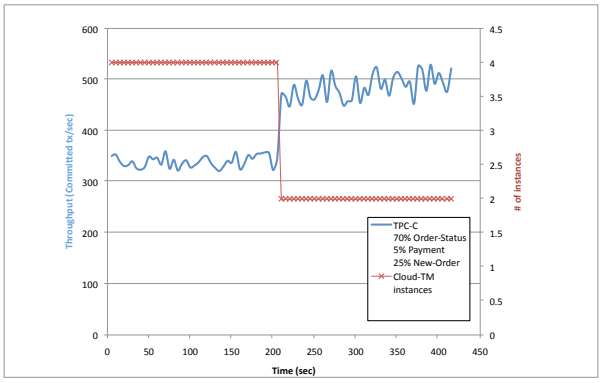


Figure 2: Execution of the example application under the ANN control.

Like ANN, MorphR also relies on WPM to gather statistics and to be notified when new data is collected from all the nodes in the system. When MorphR receives new statistics, it queries the machine learning instance on whether the current replication protocol guarantees the highest throughput possible and, if not, switches the system's replication protocol to the optimal one. The machine learning instance relies on models built a priori; in this example the models were built by varying the parameters of the benchmark and running on a cluster with 10 nodes.

**Configuring and running the example**

As before, current example is already shipped in the VM image and it is ready to be executed after a brief configuration process. For self-containment, we list the steps needed to download, compile, configure and run the application.

The package has the following structure

```
Demo2-MorphR
 |-MorphR/
 |-RadargunClientServer/
 |-run-test.sh
 |-environment.sh
 |-beforeBenchmark.sh
```

MorphR also requires the WPM runtime and a WPM Connector, both already covered in the previous example.

MorphR can be compiled by running the following commands:

```
$ cd MorphR
$ ant
```

Then RadargunClientServer can be compiled via maven by executing the following commands:

```
$ cd RadargunClientServer
$ mvn install
```

Before running this example you may need to edit the Morphr/buildPropertiesFile.sh configuration file, whose fields are listed below:

- namingHosts = the IPs of the slaves
- namingPort = the port where the JMX connection is open
- runDuration = the amount of time the MorphR should be active (in milliseconds)
- toQuery = if set to true the machine learner is queried, otherwise MorphR only prints information to build models
- outputFileName = file where the model is stored
- modelFilename = the name of the file with the model
- modelUsesTrees = model uses trees or rules
- forceStop = if set to false Stop and Go method is used, otherwise the fast switch is activated whenever possible
- abort = transactions are aborted during switches or not

The cluster parameter in the environment.sh must also be edited to include the names of the nodes in the system.

In order to run this example one simple has to execute the run-test.sh script, as it launches all the necessary components provided the directory has the following structure:

```
Demo2-MorphR
 |-MorphR/
 |-RadargunClientServer/
 |-wpm/
 |-Workload_Monitor_Connector/
 |-run-test.sh
 |-environment.sh
 |-beforeBenchmark.sh
```

**Outcome**

Figure 3 compares the throughput of MorphR with that of the three standalone replication protocols. This was obtained by executing the previously mentioned run.sh script. MorphR is able to recognise the changes in the workloads and switches the replication protocol to the optimal, based on the Machine learning output, which is queried every 70 seconds.

### 4.9.3 Demo 3 - What-if analysis (Based on TAS)

The purpose of the TAS' demo is to show the capabilities of the TAS' performance forecasting model. To this end, like for demo in Sec. 4.9.1 we will use the Radargun framework in order to run the TPC-C benchmark on top of Infinispan instances. Unlike demo in Sec. 4.9.1, however, the deployment of the application represents a use-case in which application tier and the data tier are collocated. This means that TPC-C transactions are generated directly on the Infinispan instances that will execute them. In

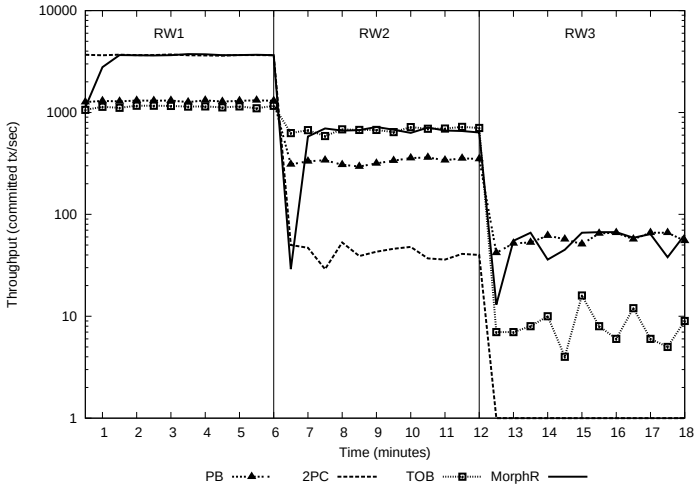

Figure 3: Comparision of the performance of MorphR with the static configurations.

this section we illustrate how to download, install and run the TAS' demo application, located in the `cloudtm-tas-demo` folder.

The VM image shipped with this document is already set up to run the application. However, we will describe all the necessary steps to perform a brand new installation on another machine.

**System requirements** The application requires the following packages to be installed

- Java 6 or higher
- Ant 1.8.4 or higher
- gnuplot
- git

**Download and installation** The package comes in the form of self-installer script that can be downloaded at the following URL

```
https://github.com/cloudtm/Demo3-TAS.git
```

and it has the following structure

```
Demo3-TAS
 |-build.sh
```

To download and install the last version of the package, simply run

```
$ ./build.sh
```

Upon completion, the structure of the package is the following:

```
TAS_Demo
 |-ControllerTas
 |-RadargunTASDemo
 |-LatticeCloudTM
 |-wpm
 |-Workload_Monitor_Connector
```

### A. Configuring and running the example

In this section, we will show how to configure and run the demo application.

**Configuring wpm** The configuration files relevant to wpm are in the wpm/config folder. The main parameters are already set up in order to match the requirements of the demo application. The only information that has to be provided at deployment time is the ip address of the machine hosting the LogService process. To this end, simply set

**Listing 26: LogService configuration**

```
...
LogService_IP_Address=IP_master
...
```

in the `log_service.config` file.

**Configuring ControllerTas** The configuration file of the ControllerTas is specified in the conf/ controller .xml file.

**Listing 27: Controller configuration**

```
<TasControllerConfiguration>
<ScaleConfig
     minNumNodes="2"
     maxNumNodes="10"
     minNumThreads="2"
     maxNumThreads="10"
     initNumNodes="2"
     initNumThreads="2"/>
<PlatformConfig
     numCores="8"/>
<GnuplotConfig
     exec="/usr/bin/gnuplot"/>
<DemoTransitoryConfig
    transtitoryTime="90"/>
</TasControllerConfiguration>
```

The semantic of these parameters is hereby explained

- minNumNodes, maxNumNodes, minNumThreads, maxNumThreads are the minimum and maximum values for nodes and threads considered in the what-if analysis
- initNumNodes, initNumThreads is the deployment configuration of the TPC-C benchmark, which must match the corresponding fields in the Radargun's configuration file.

- numCores in the number of cores per VM
- exec is the path to the gnuplot executable
- transitoryTime is the number of sec that the ControllerTas waits before considering stable the statistics collected from the slaves

The values for the numCores attribute has to be accordingly updated also in `conf/tas2.xml`.

**Listing 28: TAS configuration**

```
...
<PhysicalConfig
   numCores="8"/>
...
```

The other configuration parameters for the TAS module are not supposed to be modified, thus their meaning is not provided in this document.

**Configuring Radargun** The configuration parameters for Radargun are specified inRadargunTASDemo/conf/benchmark.xml are a subset of the ones already described in Sec. 4.9.1; therefore their explanation is omitted in this section.

Nevertheless, for the sake of clarity, here we remind that the following tags must match the values of the attributes initNumNodes and initNumThreads in the ControllerTas' configuration file.

**Listing 29: Radargun configuration**

```
<benchmark initSize="2"  maxSize="2" increment="1">
...
<TpccBenchmark
...
   numOfThreads="2"
...
/>
```

Please not that the values for initSize and maxSize have to coincide.

### B. Running the application

To run the demo simply execute the following command on the master node.

```
$ ./run.sh "IP_slave1 IP_slave2 ... IP_slaveN"
```

This will start the LogService, the ControllerTas and the Radargun's master processes on the master node; at the same time, it starts the Radargun's slave processes and Wpm's producer and consumer processes in the slave nodes.

Periodically, the Wpm instances deployed over the slave nodes will push statistics to the LogService. Upon the receipt of these statistics, the LogService will trigger the what-if analysis process of the ControllerTas. Specifically, the ControllerTas will query TAS in order to obtain a forecast about the throughput that the application would deliver if deployed over different scales (in terms both of nodes in the platform and

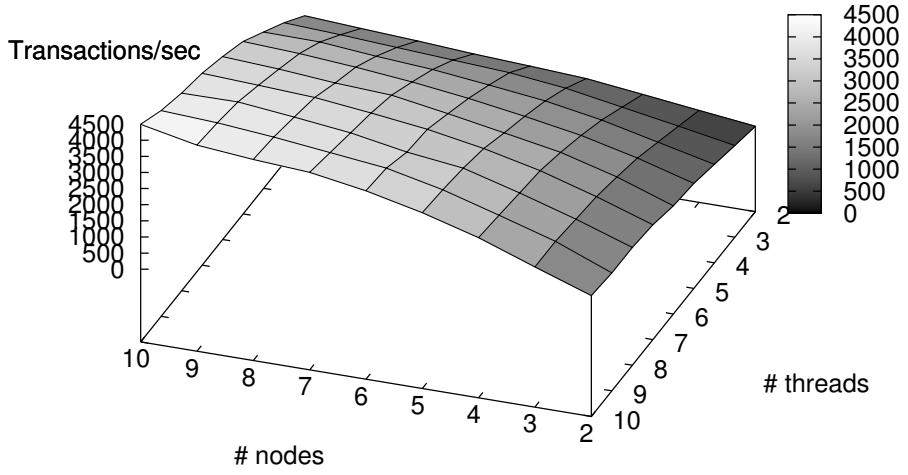


Figure 4: Outcome of TAS' what-if analysis

processing threads per node). The result of this analysis is stored both as a text file and as an image, stored respectively in the gnuplot/data and gnuplot/plot folders. Upon completion, a message is printed to notify the production of new results.

```
New plot produced and stored in gnuplot/plots/Throughput_1359906064803.eps
```

An example of produced plot is provided in Fig. 4

#### 4.9.4 Demo 4 - QoS negotiation (based on DAGS)

In this example we demonstrate how the DAGS simulation framework can be exploited in order to support the QoS negotiation phase, and allow developers of Cloud-TM applications, and providers of the Cloud-TM platform (or of PaaS services embedding the Cloud-TM platform) to reach a mutual agreement on the SLAs, and corresponding costs, to be established.

To this end, the QoS/cost management framework of the Cloud-TM platform can rely on DAGS (or on other predictos, such as TAS) to predict main system performance indexes, including average transaction response time, throughput and transaction abort ratio. These indexes are predicted executing simulation runs while varying the number of clients, the number of nodes within the platform and the degree of data replication. The number of clients and the number of servers are varied within two intervals. Fixed the number of clients and servers, for each simulation run the degree of data replication is varied between 1 and the number of servers (thus including scenarios entailing both partial and full data replication). In this demo we exploit the application

AutoDAGS, whose package is available in :

```
Demo4-DAGS
 |-AutoDAGS/
```

AutoDAGS automatizes the whole simulation process by executing all simulation runs varying the above mentioned system configuration parameters, and providing final results in a single output file. The output file is compliant with the file format required by the web application included in the QoS package allowing to visualize predicted system performance indexes with respect the SLA defined throught the WML templates. When executed, AutoDAGS performs the following computational steps:

- it parses the aggregated statistics included in the files provided by the WPM via the Log Service, in order to calculate the average computational resources demands of the different types of operations executed within the data platform (e.g. the average cpu service demand for get/put operations);
- it generates a configuration file containing all the configuration parameters needed for the simulation (also including the above-mentioned resource service demands);
- it executes e set of simulation runs varying the system configuration parameters, and eventually produces the output file.

**Configuring and running the example**

DAGS and AutoDAGS can be compiled using the GCC compiler version 4.4.3 or later, and using the makefiles included in both DAGS and AutoDAGS packages. To use AutoDAGS, the executable file generated by compiling DAGS must be placed within the same folder of the executable file generated by compiling AutoDAGS. AutoDAGS requires the following input parameters:

- m and M : (mandatory) they specify the minimum and the maximum, respectively, timestamp of data generated by the WPM to be parsed.
- f : (mandatory) it specifies the full path of the file (or the list of files) that contains data generated by the WPM to be parsed. Multiple files must be separated with a comma;
- s and S: they specify the minimum and the maximum, respectively, number of nodes to be used in the simulation .
- c and C: they specify the minimum and the maximum, respectively, number of clients to be used in the simulation (default 1).
- e: it specifies the number of transactions to be executed by each client in a simulation run (default: 10000).

- d: it specifies the number of the data objects to be used in the simulation (default 100000)

An example command line to execute AutoDAGS is:

```
$ ./AutoDAGS -m 1 -M 100000 -f data1.log,data2.log -s 1 \
    -S 4 -c 10 -C 20 -e 100000 -d 500000
```

where $data1.log$ and $data2.log$ are assumed to be files produced by the WPM.

At the end of the simulation process, the file output.txt will contain a set of rows (each one as a result of a simulation run), each one including (separated by comma):

- the number of clients used in the simulation run;
- the number of servers used in the simulation run;
- the number of clients used in the simulation run
- the predicted average transaction response time;
- the predicted transaction throughput;
- the predicted transaction abort ratio.

Figure 5 shows a graph depicting the average transaction response calculated using AutoDAGS. Predictions are calculated for a scenarions with 40 concurrent clients, varying the number of modes between 8 and 16, and, for a give number of server, varying the data replication degree between 1 and the number of server.

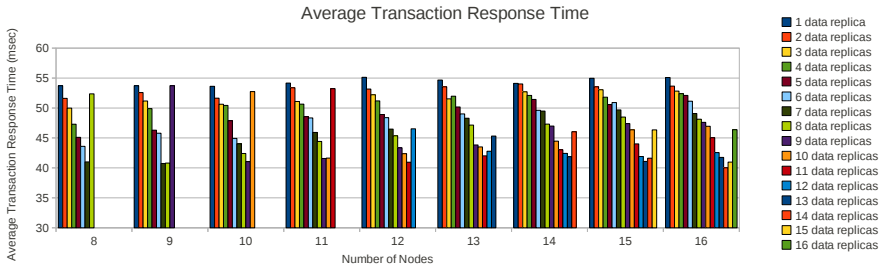


Figure 5: Example of the predicted average transaction response time

Figure 6 shows a print-screen of the web console supporting the SLA negotiation phase, and highlighting how the predictions generated by DAGS can be exploited, by both the platform provider and the application developer, in order to reach an agreement on the QoS levels to be guaranteed, as well as on the corresponding operational costs (computable based on the amount and type of computation resources necessary to ensure pre-determined QoS levels). Charts highlight using red color predicted values which violate the SLA defined by the user throught the WML templates. In the example depicted in the figure, system configurations which are expected to provide

a throughput greater than 80 transaction per seconds are expected violate the average transaction response time as defined by user, while system configurations which are expected to provide a throughput greater than 90 transaction per seconds are expected to violate the maximum accettable abort rate.

### 4.9.5 Demo 5 - Self-tuning data placement (based on AUTOPLACER)

In this section we demonstrate the AutoPlacer optimizer. For this demo, we use a synthetic benchmark named RadarGun, which was configured to achieve some data locality over the cluster. Each machine in the cluster performs a read dominant workload and it has a disjoint subset of the total number of keys that is most frequently accessed by each machine. In more detail, 90% of the time, each machine chooses a key from it local subset and the remaining for other machines' subsets.

Another demo is also implemented to demonstrate the AutoPlacer effectiveness in the case of real applications too by using the Geograph Pilot application. But since the deployment of that pilot requires a more complex software architecture (e.g. JBoss Application Server, TorqueBox platform) it is not shipped with the VM associated to this deliverable. For further details on that demo, refers to the deliverable D5.7 "Pilot Demonstration".

The RadarGun is composed by a master node, referred to hereinafter as master_node, and by a set of slaves nodes, referred to hereinafter as slave_nodes. The master_node is responsible to coordinate all the benchmark phases (such as cluster creation, population, benchmark, etc.) which are executed by all the slave_nodes. All the steps described below must be performed in the master_node.

The VM image contains the following software package:

```
Demo5-AUTOPLACER
|-Csv-reporter/
|-Machine-learner/
|-Radargun/
|-www/
\-dist/
```

The dist/ provides the compiled code ready to be executed, after some configurations. However, it is possible to download the source code and compile it yourself.

#### Download and Compile the source code

All the source code is available on GitHub in the following url:

```
https://github.com/cloudtm/cloudtm-auto-placer
```

In order to download and compile the source code, perform the following steps:

```
$ git clone git://github.com/cloudtm/cloudtm-auto-placer.git
$ cd cloudtm-auto-placer
$ ./dist.sh
```

The ./dist.sh creates the folder dist/ similar to the one provided in the VM image. The next steps is to configure the demo.

Configure CSV Reporter

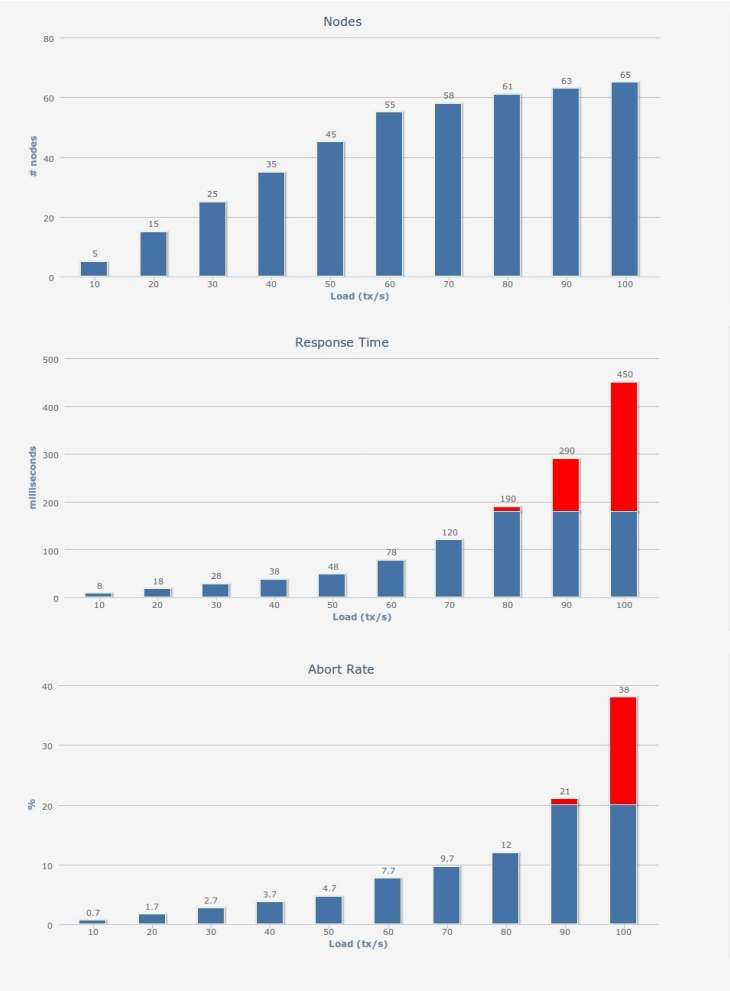


Figure 6: Example illustrating how the predictions generated by DAGS can be exploited in the context of the SLA definition phase.

The CSV Reporter is a lightweight application that collects the Infinispan statistics over time and export them in a .csv file. In addition, we provide a PHP web page that analyses the .csv file generated and plots this information. The PHP web page is already installed in the VM image web server, so the next step is not necessary.

The PHP web page is *optional*, but if you want see the system's performance over time you have to copy the folder www/ for your web server folder (the web server installed must support PHP) and you should have write permissions in that folder. Assuming you are located in cloudtm−auto−placer folder and your web server folder is /var/www/html, perform the following steps

```
$ mkdir /var/www/html/example
$ # needed to save the .csv file generated
$ mkdir /var/www/html/example/current
$ cp -r www/* /var/www/html/example
$ chmod -R +w /var/www/html/example
```

The configuration for CSV Reporter needs a little changes. The file can be found in dist/conf/config.properties and the properties that may need to be modified are the following:

- reporter.ips: refers to a comma separated list of hostname:port of all the slave_nodes. The port value is the JMX port defined in dist/conf/environment.sh in property named JMX_SLAVES_PORT which value default to 9998;
- reporter.output_file : refers to the file path where the CSV Reporter will save the statistics. The default value is /var/www/html/example/current/reporter.css, but if you don't have the PHP web page, you can set the location to another folder, for example, /tmp/reporter.css;
- reporter.updateInterval : sets the update interval in seconds in which the CSV Reporter will collect the statistics. Each time the statistics are collected, a new line is added in the file set by reporter.output_file.

### Configure Data Placement and Machine Learner

The data placement configuration can be found in the file dist/plugins/infinispan52cloudtm/conf/ispn.xml and the default configurations are the following: (note that the configurations does not need any changes when the VM image is used)

**Listing 30: Data Placement configuration**

```
<dataPlacement
   enabled="true"
   maxNumberOfKeysToRequest="1000"
   objectLookupFactory="org.infinispan.dataplacement.c50.
                                        C50MLObjectLookupFactory">
 <properties>
```

```
  <property
    name="keyFeatureManager"
    value="org.radargun.cachewrappers.RadargunKeyFeatureManager"
    />
  <property
    name="location"
    value="/tmp/ml"
    />
  <property
    name="bfFalsePositiveProb"
    value="0.001"
    />
  </properties>
</dataPlacement>
```

The Machine Learner executable location should match with value defined in the location property. You may copy the folder in dist/ml to /tmp or change the property value to the new location. Note that the Machine Learner location must match in all the slave_nodes.

Before trying this demo, you need to check if the compiled Machine Learner runs in your system. If not, you can try to compile the C5.0[7] (the Machine Leaner used) to your system. If it is not possible, you can set the objectLookupFactory to the value org.infinispan.dataplacement.hm.HashMapObjectLookupFactory. This option will save the new key mapping in a Java Hash Map instead of the space efficient Machine Learner.

**Running the Demo**

After finishing all the previous steps, you only have to copy the dist/ folder to all the slave_nodes. After everything is copied, go to the master_node and perform the following command:

```
$ ./bin/benchmark.sh -i [number of nodes] [list of slaves_nodes]
```

Then you can check the system performance in the file set by reporter.output_file or in your browser in the link:

```
http://master_node/example
```

The expected output is shown also in Figure 7

[7] http://www.rulequest.com/see5-info.html

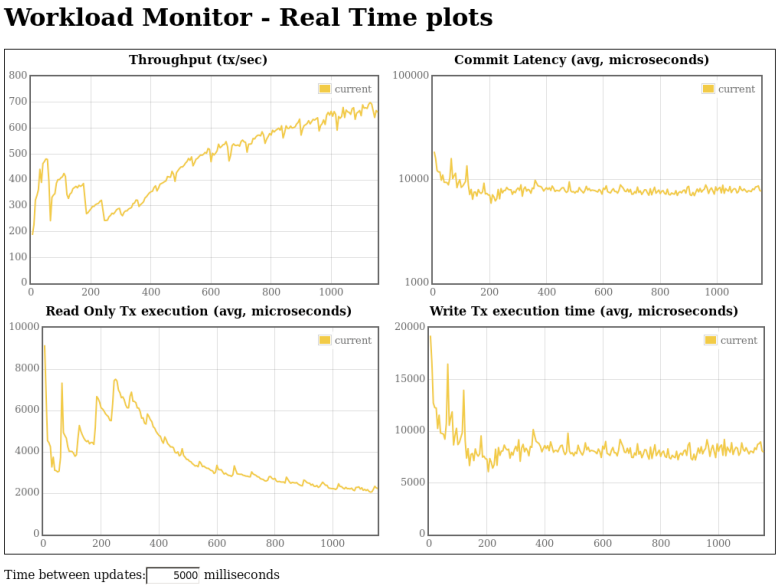


Figure 7: The VM ships with a lightweight statistics viewer that alows monitoring a set of key performance indicators, which allow monitoring the impact on performance of the self-tuning of the data placement in the Cloud-TM Data Platform.

# 5 Licensing

The Cloud-TM project encompasses a large ecosystem of open-source modules developed by multiple partners, often in collaboration. The general rule concerning licensing of code developed in the scope of the Cloud-TM project is that the GNU Lesser General Public License (GNU LGPL), Version 3, 29 June 2007, applies.

However, individual modules of the Cloud-TM platform, despite being open-source, may ship with different licensing schemes (such as GPL, AGPL or Apache).

Hence, it is sole responsibility of the users of any software artifact developed in the context of the Cloud-TM project to verify the exact licensing scheme (and copyright ownership) of any module of the Cloud-TM platform that she may want to use or integrate in her software projects.

# References

[1] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues, "Autoplacer: scalable self-tuning data placement in distributed key-value stores," in *Proc. ICAC 2013*, 2013.

[2] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues, "Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation," in *DSN*, pp. 1–12, 2013.

[3] TPC Council, "TPC-C Benchmark, Revision 5.11," Feb. 2010.