Cloud-TM

Specific Targeted Research Project (STReP)

Contract no. 257784

Companion document for deliverable D2.3: Prototype of the RDSTM and RSS

Date of preparation: 15 Jan. 2013
Start date of project: 1 June 2010
Duration: 36 Months

Contributors

Emmanuel Bernard, RED HAT
João Cachopo, 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, INESC-ID
Manik Surtani, RED HAT

Table of Contents

1 Introduction

This document accompanies Deliverable D2.3, Prototype of the Cloud-TM Reconfigurable Distributed STM and Storage System, which we refer to as the Cloud-TM Data Platform, for the sake of brevity. The Cloud-TM Data Platform 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:

- overviewing the key functionalities provided by the various software layers composing the Cloud-TM data platform;
- providing references to detailed documentation on the usage and configuration of the various modules;
- describing the structure of the software package associated with D2.3;
- providing instructions on how to compile, deploy and run the example applications included in the software package.

The reader should note that, at the time of writing, the Cloud-TM Data platform is still being enhanced and extended to achieve smooth integration with the prototype of the Autonomic Manager (D3.4). During this last phase of the project, in fact, the effort invested in the consortium is focused on the integration, profiling and benchmarking of the various modules of the platform (most of which had to be developed independently to parallelize work in an effective way). Hence the current deliverable represents a snapshot of the current state of progress, rather than a final version of the Cloud-TM Data Platform prototype, whose delivery is instead planned for end of May 2013 (D4.5 - Final Cloud-TM Prototype).

The structure of this document is the following. We start, in Section 2, by overviewing the architecture of the Cloud-TM Data Platform, and illustrating the key functionalities provided by each of its main components. Next, in Section 3, we describe the content of the package containing the platform's prototype, and provide instructions on how to install it and test it by running the demo applications included in the package.

1.1 Relationship with other deliverables

The Cloud-TM Data Platform prototype is being integrated with the Cloud-TM Autonomic Manager Prototype (D3.4) in order to produce the Final Cloud-TM Prototype (D4.5). The Final Architecture Report (D4.7) will provide detailed information on the internals of the various modules of the Cloud-TM Platform.

This prototype builds on the results achieved with the following deliverables:

- D1.1 - User requirements report: the requirements gathered in this deliverable drove the design and implementation of the services provided by the Cloud-TM Data Platform.

- D2.1 - Architecture Specification Draft: the design choices formalized in this deliverable laid the foundations on top of which the Cloud-TM Data Platform was built.
- D2.3 - Preliminary Prototype of the RDSTM and of the RSS: the current prototype represents an evolution of the preliminary prototype delivered at the end of year 1.

2 Architectural Overview of the Cloud-TM Data Platform

The high level architecture of the entire Cloud-TM Platform is depicted in Figure 1. This deliverable represents the companion document of the Cloud-TM Data Platform, which is shown on the left side of Figure 1 (highlighted by a gray box).

In the following we briefly overview the main modules composing the prototype. For each module, we report the key functionalities it supports, and provide references to online documentation containing detailed information on how to configure and use it.

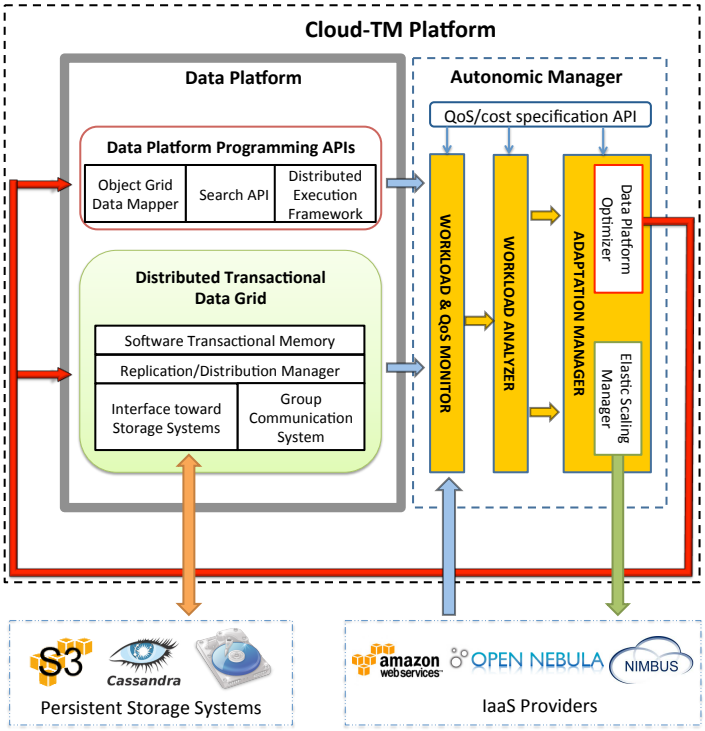

Figure 1: Architecture of the Cloud-TM platform.

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.

The main features that FF provides to application programmers are:

- The *Domain Modelling Language (DML)*;
- Transaction control via annotations and the standard Java Transaction API (JTA);
- Transparent two-way mapping between the object-oriented data model and the chosen underlying data representation.

The DML is a micro-language designed specifically to represent the structure of a domain model. It has constructs for specifying both entity types and associations between entity types. It has a Java-like syntax to be easy to learn by Java programmers.

By using this language, a programmer can describe the application's domain model without compromising to any specific implementation of the objects' structure. This decoupling can be used to adapt to different requirements, such as to change the objects' layout in memory, to efficiently load from persistent storage the most commonly accessed values, and to change the underlying storage and transaction technology in use.

The DML code generator operates at application's compile-time. This generator is responsible for creating a concrete implementation of the domain entities' structure. It provides standard APIs with accessors (`get/contains/...`) and mutators (`set/add/remove/...`) for the modelled entities' attributes and relations, whose internal implementation may vary, according to different needs.

During the past six months the FF has undergone a significant refactoring, in which we added the concept of a *Backend*. Each concrete backend is a replaceable sub-module of the framework that provides the bindings from the DML model to a concrete transaction system and a concrete data storage system. Above all, this allows for the programmer to substitute one backend with another without having to rewrite any part of the application code. This is possible because the code generators provided by each backend maintain a common API for the generated code.

For Cloud-TM, there are currently two backend implementations: the *Infinispan Direct backend*, and the *OGM backend*. Both backends provide bindings to the underlying storage provided by the Cloud-TM Transactional Data Grid (namely Infinispan, see Section 2.4). However, each backend provides a different mapping between the objects and their key-value representation.

The Infinispan Direct backend leverages on the fact that the Cloud-TM Transactional Data Grid is already embedded into the application's memory space, and thus uses very lightweight domain objects. These objects' attributes and relations to other objects don't take up any additional memory space. Whenever the value of any attribute or relation is requested, the generated code fetches the value directly from the corresponding cache entry in Infinispan. As such, the objects themselves, are *hollow*, in the sense that they only contain the reference to their primary key attribute.

The OGM backend relies on Hibernate Core's engine, stacked with the Hibernate Object/Grid Mapper (Hibernate OGM) to do the actual load/store operations from/to Infinispan. As such, the code generation for domain entities creates typical *POJO*s[1] together with a standard Java Persistence API mapping.

[1] Plain Old Java Objects.

Regardless of which backend the programmer uses, the FF additionally provides automatic management of bidirectional relations between domain entities modelled in DML (for example, when adding an object to a collection, both ends are updated). It also integrates with the Cloud-TM Search API module, by causing the updated entities to be indexed on successful commit and it supports the use of the Hibernate Search and Apache Lucene DSL query language over the domain entities.

Pointers to source code and additional documentation:

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

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

  or directly in the FF project's web page:

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

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

  ```
  ~cloudtm/cloudtm-data-platform/src/Fenix-Framework/
  ```

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

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

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

  ```
  ~cloudtm/cloudtm-data-platform/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/cloudtm-data-platform/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/cloudtm-data-platform/docs/Hibernate-OGM/
hibernate_ogm_reference.pdf
```

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).

Hibernate Search is built on top of Apache Lucene, and makes use of inverted indexes to offer full-text query support to various data sources. Note that support goes beyond pure text search, including also numeric and range queries as well as the recently added geospatial queries. Here are some examples of queries:

- find all users whose address ends with @redhat.com
- find all books whose price is below 20$ and talking about transactions
- find all libraries named Lincoln less than 1 km from the city hall.

The project is focused on ensuring ease of use, by handling transparently the low level and performance sensitive infrastructure code, and letting application developers focus on the business side of their queries. In particular, an object oriented fluent API is provided to express queries in a very readable way.

Distributing indexing and querying across the underlying distributed data platform is also a key component of Hibernate Search. This is achieved in different ways. Hibernate Search offers a way to send workload from slave nodes to master nodes via JMS or via the JGroups communication library. Index synchronization between nodes can be achieved in various ways including via delta file copies or by storing the index in the Cloud-TM data platform to make changes instantly visible to all nodes of the cluster.

Likewise, high throughput and low latency queries are at the core of the project and are supported by specific configurations as, for example, the one associated to the Near Real Time approach.

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/tree/
cloudtm
```

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/cloudtm-data-platform/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/cloudtm-data-platform/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.

The Distributed Execution Framework can be seen as an extension of the familiar java.util.concurrency framework, designed to transparently support, on top of the in-memory transactional data grid the execution of Java's Callable tasks, which can consume data from the underlying data grid. Unlike most other distributed frameworks, the Cloud-TM Distributed Execution Framework uses data from the transactional data platform's nodes as input for execution tasks. This provides a number of relevant advantages:

- The availability of the transactional abstraction for the atomic, thread safe manipulation of data allows drastically simplifying the development of higher level abstractions such as concurrent data collections or synchronization primitives.
- The Distributed Execution Framework capitalizes on the fact that input data in the transactional data grid is already evenly distributed (using a Consistent Hashing scheme [**?**]). Since input data is already evenly distributed execution tasks are likely to be automatically distributed in a balanced fashion as well; users do not have to explicitly assign work tasks to specific platform's nodes. However, our framework accommodates users to specify arbitrary subsets of the platform's data as input for distributed execution tasks.
- The mechanisms used to ensure the fault-tolerance of the data platform, such as redistributing data across the platform's nodes upon failures/leaves of nodes, can be exploited to achieve failover of uncompleted tasks. In fact, when a node F

fails, the data platform's failover mechanism will migrate, along with the data that used to belong to F, also any task T that was actively executing on F.

The main interfaces for distributed task execution are DistributedCallable and DistributedExecutorService. DistributedCallable is a subtype of the existing Callable from the java.util.concurrent package. DistributedCallable can be executed in a remote JVM and receive input from the transactional in-memory data grid. The task's main algorithm could essentially remain unchanged, only the input source is changed. Existing Callable implementations will most likely get their input in the form of some Java object/primitive while DistributedCallable gets its input from the underlying transactional data platform in the form of key/value pairs (see Section 2.4). Therefore, programmers who have already implemented the Callable interface to describe their task units would simply extend their implementation to match DistributedCallable and use keys from the data grid as input for the task. Implementation of DistributedCallable can in fact continue to support implementation of an already existing Callable while simultaneously be ready for distributed execution by extending DistributedCallable.

DistributedExecutorService is a simple extension of the familiar ExecutorService interface from the java.util.concurrent package. However, advantages of DistributedExecutorService are not to be overlooked. Existing Callable tasks, instead of being executed in JDK's ExecutorService, are also eligible for execution on the distributed Cloud-TM data platform. The Distributed Execution Framework would migrate a task to one or more execution nodes, run the task and return the result(s) to the calling node. Of course, not all Callable tasks will benefit from parallel distributed execution. Excellent candidates are long running and computationally intensive tasks that can run concurrently and/or tasks using input data that can be processed concurrently.

The second advantage of the DistributedExecutorService is that it allows a quick and simple implementation of tasks that take input from Infinispan cache nodes, execute certain computation and return results to the caller. Users would specify which keys to use as input for specified DistributedCallable and submit that callable for execution on the Cloud-TM platform. The Distributed Execution Framework's runtime would locate the appropriate keys, migrate DistributedCallable to target execution nodes and finally return a list of results for each executed Callable. Of course, users can omit specifying input keys in which case the Cloud-TM Data Platform would execute DistributedCallable on all keys for a specified data store.

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/
  core/src/main/java/org/infinispan/distexec
  ```

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

```
~cloudtm/cloudtm-data-platform/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/cloudtm-data-platform/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.

Infinispan architecture is extremely flexible and supports a range of operational modes: standalone (non-clustered) mode, or distributed mode. In the standalone mode, it works as a shared-memory (centralized) Software Transactional Memory (STM), providing support for transactions and a key/value store interface. This configuration is thought for small deployments and test environments, whereas in order to unlock the scalability potential of Infinispan applications should use it in either full or partial replication mode. In distributed mode, Infinispan will be able to transparently leverage the computational resources of a set of shared-nothing nodes. Communication among nodes and group membership is maintained via the JGroups' Group Communication Toolkit, which has also been enriched during the Cloud-TM project with additional group communication primitives.

In order to maximize efficiency in presence of workloads having different characteristics, Infinispan has been extended to support three alternative distributed data management strategies: Two Phase Commit (2PC) based replication, Total Order based Certification (TO) [**?**], and Primary Backup (PB) [**?**]. 2PC is optimized for workloads generating a significant number of update transactions and with low conflict probability. TO delivers optimal performance in write intensive scenarios with medium/high conflict probability. Finally, PB outperforms the other two protocols in read dominated scenarios, or, in extremely high contention scenarios, where it is beneficial to reduce the number of nodes concurrently updating the state of the platform. Infinispan has been extended to support on-line switching between the above replication protocols, leveraging innovative non-blocking schemes that minimize performance penalization during system's reconfiguration.

Protocol-wise, Infinispan has been extended with a novel, highly scalable distributed multi-versioning scheme, called GMU [**?**], which has the following unique characteristics:

- **strong consistency:** The consistency semantics of GMU adheres to the Extended Update Serializabilty criterion [**?**], which ensures the most stringent of the standard ISO/ANSI SQL isolation levels, namely the `SERIALIZABLE` level. Beyond that, it ensures that the snapshot observed by transactions, even those that need to be aborted, is equivalent to that generated by some serial execution of transactions. By preventing transactions from observing not-serializable states, application developers are spared from the complexity of dealing explicitly with anomalies due to concurrency that may lead to abnormal executions [**?**].

- **wait-free read-only transactions:** GMU ensures that read-only transactions can be committed without the need for any validation at commit time. Further, it guarantees that read-only transactions are never blocked or aborted. These properties are extremely relevant in practice, as most real-life workloads are dominated by read-only transactions.

- **genuine partial replication:** GMU is designed to allow achieving high scalability also in presence of write intensive workloads by ensuring that update transactions commit by contacting exclusively the subset of nodes that maintain data they read/wrote. Unlike other multi-versioning distributed consistency protocols, hence, GMU can commit update transactions without either relying on any centralized service (which is doomed to become the bottleneck of the system and ultimately limit its scalability) or flooding all nodes in the system (which would generate a huge amount of network traffic in large scale systems).

Another relevant, recent feature that has been introduced in Infinispan is the, so called *non-blocking state transfer*. This feature allows Infinispan to keep on processing transactions even when the scale of the platform is changing, i.e. whenever nodes are leaving/joining the system. This feature is particularly relevant in cloud environments, where it is highly desirable, in order to minimize operational costs, to elastically scale the platform depending on the actual load generated by user-level applications.

Within the Cloud-TM platform, in-memory data replication represents the reference mechanism to ensure fault-tolerance and data reliability without incurring in the overheads of synchronous logging to persistent storages. Nevertheless, data persistence (in particular asynchronous disk logging) may still represent an indispensable requirement for a large class of Cloud applications, in particular as a means to ensure disaster recovery capabilities. Infinispan supports persistence of data towards external persistent storage platforms via a modular, plug-in based, architecture that allows to neatly encapsulate the inherent peculiarities underlying the interactions with heterogeneous persistent storages via a homogeneous abstraction layer. Currently, Infinispan ships with plugins for the main alternative classes of persistent storage systems, such as local file-systems, DBMSs offering JDBC interfaces, and Cassandra.

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/tree/cloudtm/
  ```

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

  ```
  ~cloudtm/cloudtm-data-platform/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/cloudtm-data-platform/docs/Infinispan/
  infinispan_reference.pdf
  ```
  and
  ```
  ~cloudtm/cloudtm-data-platform/docs/Infinispan/
  infinispan_ext_reference.pdf
  ```

3 Setting up the prototype and the example applications

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

3.1 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:

```
cloudtm-data-platform
    |-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
    |    \-D2_3_Data_Platform_Prototype.pdf
    |-examples/
    |    |-bin/
    |    |    |-scenario1-ispn.zip
    |    |    |-scenario1-ogm.zip
    |    |    |-scenario2-ispn.zip
    |    |    |-scenario2-ogm.zip
    |    |    |-scenario3-ispn.zip
    |    |    |-scenario3-ogm.zip
    |    |    |-scenario4-ispn.zip
    |    |    \-scenario4-ogm.zip
    |    |-scenario1/
    |    |-scenario2/
    |    |-scenario3/
    |    \-scenario4/
    \-src/
         |-Fenix-Framework/
         |-Hibernate-OGM/
         |-Hibernate-Search/
         |-Infinispan/
         \-JGroups/
```

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

The *cloudtm-data-platform* folder contains the source code, documentation and examples of the preliminary prototype of the Cloud-TM 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.

3.2 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.

3.2.1 Preparing the Fénix Framework

The Fénix Framework is already shipped in the VM image, ready to be used by the example applications (as we will describe more in detail in Section 3.3). 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>
```

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

```
    <artifactId>fenix-framework-backend-infinispan</artifactId>
    <version>2.1-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>
      <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] `http://maven.apache.org/plugins/maven-dependency-plugin/`

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

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

3.3 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 scenarios 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 scenario we provide the instructions on how to run it, and we highlight the most relevant aspects shown.

3.3.1 Scenario 1: A simple Java application

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

```
scenario1
    |-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 scenario 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 scenario1 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).

3.3.2 Scenario 2: Using the Search API

This scenario 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:

```
scenario2
    \-src
        \-main
            \-java
                \-test
                    |-Author.java
                    |-Book.java
                    |-ComicBook.java
                    |-MainApp.java
                    |-Publisher.java
                    \-ScifiBook.java
```

And the following file has been added:

```
scenario2
    \-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 scenario 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 scenario2 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.

3.3.3 Scenario 3: Running on a cluster

This scenario extends the previous scenario 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:

```
scenario3
    \-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):

```
scenario3
    \-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 scenario 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 scenario3 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
```

3.3.4 Scenario 4: Persisting state

Finally, for the last scenario, 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 scenario to store the data in the local File-System. The only files changed were the following:

```
scenario4
    \-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 scenario4 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 scenario, additionally all the data created by the application is stored in /tmp/fs-store.

3.4 Running the pre-compiled examples

In Section 3.3 we demonstrated how to run the examples. However, the examples run in one machine (where scenarios 3 and 4 use two processes to simulate two machines) and, by following the steps mentioned in Section3.3, 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 scenarios 1 and 2, you only need to unzip and execute run.sh:

```
$ unzip <scenario1 or scenario2><-ispn or -ogm>.zip
$ cd <scenario1 or scenario2>
$ ./run.sh
```

For scenarios 3 and 4, 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 <scenario3 or scenario4><-ispn or -ogm>.zip
$ cd <scenario3 or scenario4>
$ 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 14: 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 scenario 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 15: 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>
```