
# Cloud-TM

Specific Targeted Research Project (STReP)

Contract no. 257784

# D2.1: Architecture Draft

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

## Contributors

Emmanuel Bernard, Red Hat
Joao Cachopo, INESC-ID
Bruno Ciciani, CINI
Diego Didona, INESC-ID
Francesca Giannone, ALGORITHMICA
Mark Little, Red Hat
Sebastiano Peluso, INESC-ID
Francesco Quaglia, CINI
Luis Rodrigues, INESC-ID
Paolo Romano, INESC-ID
Vittorio A. Ziparo, ALGORITHMICA

# Table of Contents

# 1 Introduction

This deliverable is the result of the activities carried out by the project's partners in the context of WP2 (Task 2.1) and WP3 (Task 3.1), and describes the planned architecture of the Cloud-TM platform.

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

- defining the platform's architecture, describing relations and data flows among the various modules of the system, as well as the internal design of the main components of the Cloud-TM platform;
- specifying, whenever this is already possible, the external interfaces among the main components of the Cloud-TM platform, in order to decouple them and allow parallel development by the various members of the Cloud-TM project's team.

It is important to clarify that the design decisions discussed in this deliverable are not meant as conclusive and inalterable. Conversely, they represent a commonly agreed starting point, which summarizes the results of the design activities performed during the first year of the project, and which will need to be validated (and possibly be subject to changes) while progressing with the research and development work carried out during the future phases of the project.

## 1.1 Relationship with other deliverables

The architecture defined in this deliverable has been developed on the basis of the user requirements gathered in the deliverable D1.1 "User Requirements Report", and taking into account the technologies identified in the deliverable D1.2 "Enabling Technologies Report".

This deliverable has also a relation with the deliverable D2.2 "Preliminary Prototype of the RDSTM and the RSS", which includes a preliminary (non self-tunable) version of the components of the Cloud-TM Data Platform, whose architecture is described in the following.

Also, section 4.2, which describes the architecture of the Workload Monitor, is clearly related to the deliverable D3.2 "Prototype of the Workload Monitor".

Finally, the architecture defined hereafter will serve as a reference to guide the development of the deliverables associated with the prototypes of the Cloud-TM platform, namely the deliverables D2.3 "Prototype of the RDSTM and of the RSS", D3.2 "Prototype of the Workload Analyzer", D3.4 "Prototype of the Autonomic Manager", and, finally, D4.5 "Final Cloud-TM Prototype".

# 2 Architectural overview

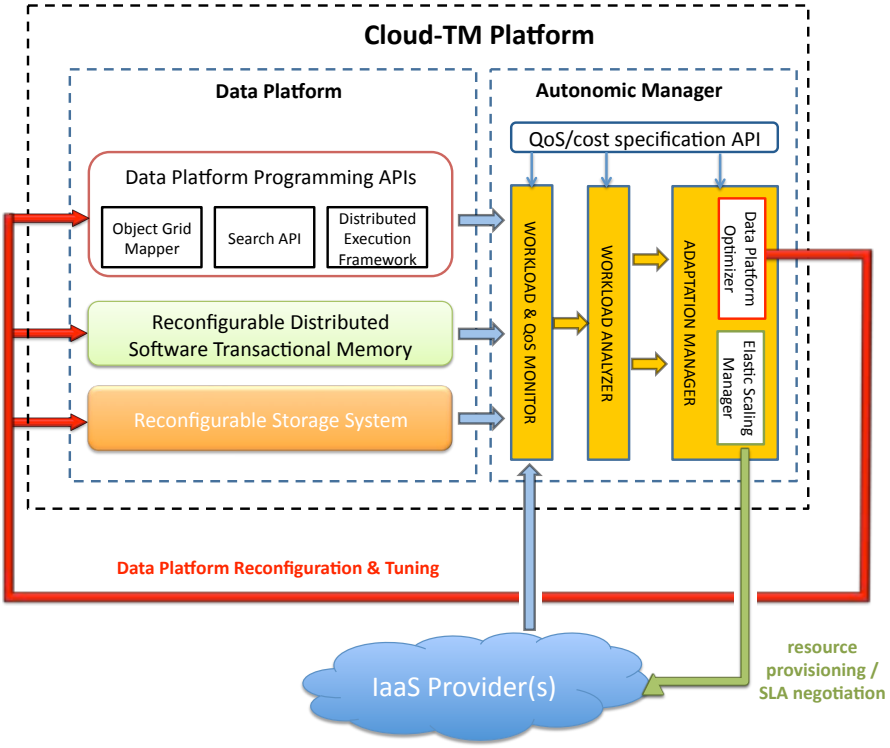



Figure 1: Architectural Overview of the Cloud-TM Platform.

Figure 1 presents the high level architecture diagram of the Cloud-TM platform. This will be used as a starting point to derive, in a top down fashion, more refined versions of the platform's main building blocks, which will be described in detail in the remainder of this document.

This diagram represents a refinement of the architectural diagram shown in Figure 1 of Annex I - "Description of Work", which has been updated to incorporate the results achieved so far in the project.

As already discussed in deliverable D1.1 "User requirements", the Cloud-TM platform will be formed by two main parts: the Data Platform and the Autonomic Manager.

The Data Platform will be responsible for storing, retrieving and manipulating data across a dynamic set of distributed nodes, elastically acquired from the underlying IaaS Cloud provider(s). It will expose a set of APIs, denoted as "Data Platform Programming APIs" in Figure 1, aimed at increasing the productivity of Cloud programmers from a twofold perspective:

1. To allow ordinary programmers to store and query data into/from the Data Platform using the familiar and convenient abstractions provided by the object-oriented

paradigm, such as inheritance, polymorphism, associations.

2. To allow ordinary programmers to take full advantage of the processing power of the Cloud-TM platform via a set of abstractions that will hide the complexity associated with parallel/distributed programming, such as load balancing, thread synchronization and scheduling, fault- tolerance.

Lower in the stack we find the backbone of the Data Platform, namely a highly scalable, elastic and dynamically Reconfigurable Distributed Software Transactional Memory (RDSTM).

In order to maximize the visibility, impact and the chances of future exploitation of the Cloud-TM project, the consortium agreed to use Red Hat's Infinispan, namely one of the leading open-source projects in the area, as the starting point for developing this essential component of the Cloud-TM platform. In the remainder of this document, we will therefore consider Infinispan as synonymous of RDSTM or, more precisely, as the base component that implements the RDSTM.

Infinispan is a recent in-memory transactional data grid designed from the ground up to be extremely scalable. Infinispan will be extended during the project with innovative algorithms (in particular for what concerns data replication and distribution aspects), and real-time self-tuning schemes aimed at guaranteeing optimal performance even in highly dynamic Cloud environments.

At its lowest level, the Data Platform will support the possibility to persist its state over a wide range of heterogeneous durable storage systems, ranging from local/distributed filesystems to Cloud storages (such as Amazon's S3 or Cassandra).

The Autonomic Manager is the component in charge of automating the elastic scaling of the Data Platform, as well as of orchestrating the self-optimizing strategies that will dynamically reconfigure the data distribution and replication mechanisms to maximize efficiency in scenarios entailing dynamic workload.

Its topmost layer will expose an API allowing the specification and negotiation of QoS requirements and budget constraints.

The Autonomic Manager will leverage on pervasive monitoring mechanisms that will not only track the utilization of heterogeneous system-level resources (such as CPU, memory, network and disk), but will also characterize the workload sustained by the various subcomponents of the transactional Data Platform (local concurrency control, data replication and distribution mechanisms, data contention level) and their efficiency.

The stream of raw data gathered by the Workload and Performance Monitor component will then be filtered and aggregated by the Workload Analyzer, which will generate distilled workload profiling information and alert signals that will serve as input for the Adaptation Manger.

Finally, the Adaptation Manager will host a set of optimizers that will rely on techniques of different nature, ranging from analytical or simulation-based models to machine-learning-based mechanisms, which will self-tune the various components of the Data Platform and control the dynamic auto-scaling mechanism with the ultimate goal of meeting QoS/cost constraints.

# 3 Data Platform

This section is devoted to discuss the architectural aspects associated with the development of the key building blocks of the Cloud-TM Data Platform.

Given that self-tuning and self-optimization represent a cross-layer concern in the Cloud-TM platform, we start by introducing, in Section 3.1, the set of standardized interfaces and mechanisms that will be adopted throughout the vertical stack of layers composing the Data Platform for achieving remote monitoring and dynamic reconfiguration.

The remainder of the section is then structured in a top-down fashion.

We start by describing, in Section 3.2, the APIs that will be exposed by the Data Platform to the programmers, and the architectural organization of the main building blocks that will expose these APIs.

In Section 3.3, we discuss the internal organization of the RDSTM module. To this end, we start by providing an overview of the internal architecture of Infinispan, a popular open-source project (whose development is led by one of the partners the Cloud-TM consortium, namely Red Hat) that has been selected as the reference platform to incorporate the research results achieved throughout the project. Next we focus on how the architecture of Infinispan will be extended in order to meet dynamic reconfiguration and self-tuning requirements.

Finally, Section 3.4 presents the architecture of the Reconfigurable Storage System, focusing on the issues of how to achieve interoperability with a large ecosystem of heterogeneous external data stores.

## 3.1 Generic Tunable Component Interfaces

The Cloud-TM platform aims at achieving self-tuning in a pervasive fashion, across multiple layers of the Data Platform. In order to decouple the development of the Data Platform and of the Autonomic Manager, it has been decided to standardize the mechanisms to:

1. allow the Autonomic Manager to monitor and trigger the reconfiguration of the Data Platform components;
2. allow the developers of each module/component of the Data Platform to expose meta-data aimed at guiding the reconfiguration logic implemented by the Autonomic Manager.

Figure 2 provides an overview of the software architecture designed to achieve these goals.
Given that we plan to use Java technology for the Data Platform components, it has been agreed to rely on the standard Java Management Extensions (JMX) for exposing:

1. the attributes to be monitored by the Autonomic Manager;
2. the methods used to trigger the reconfiguration of the components, which will range from relatively simple alterations of the setting of a numerical parameter, to more complex run-time switches among alternative algorithmic implementations of a given functionality.

Each component of the Data Platform will externalize statistical information via standard JMX MXBeans, and expose them via a common JMX MBean Server. As it will be further discussed in Section 4, in order to maximize scalability and efficiency, the monitoring/tuning information will not be propagated in the platform using point-to-point communication (as mandated by the standard, RMI-based JMX connectors [1]). Conversely, this information will be convoyed, via a protocol adaptor, by relying on the Lattice framework [2], offering specialized data distribution mechanisms to propagate information across the so called:

- data plane, which is dedicated to disseminate the monitoring data;
- information plane, which is dedicated to the advertise for meta-data concerning the attributed of tunable components;
- control plane, which is dedicated to the propagation of the reconfiguration signals generated by the Autonomic Manager.

While the reliance on JMX ensures the homogeneity of the interactions between the Autonomic Manager and the Java components of the Data Platform, it is on the other hand desirable to also include non-Java components within the control loop, such as the virtual machines or the operating systems on top of which the Data Platform components are run. Forcing these components to externalize a JMX interface would

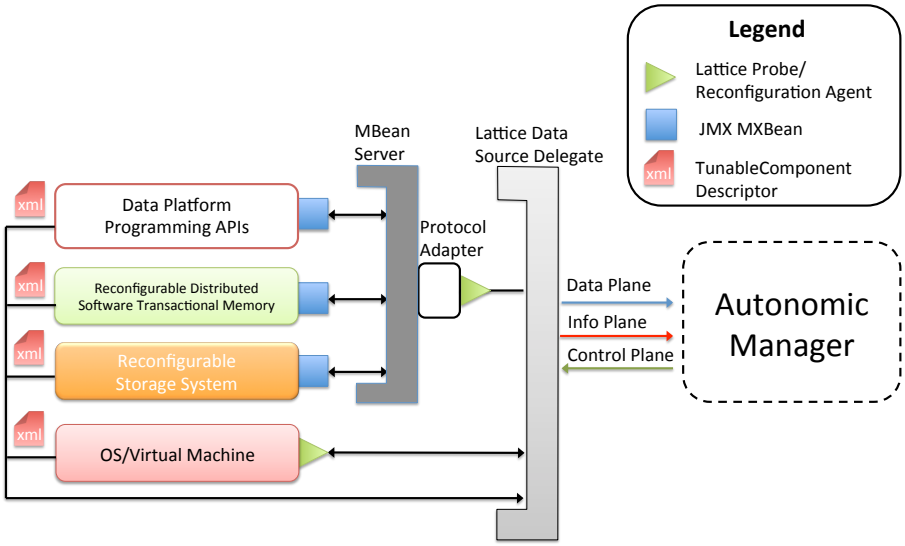



Figure 2: Standardized Interfaces for Monitoring/Tuning the Data Platform Components.

demand their encapsulation into Java modules, and force the adoption of expensive Java Native Interfaces (JNI) invocations in order to access information/services not directly available within a JVM. At the light of these considerations, in order to maximize efficiency, it has been decided to use native Lattice probes/reconfiguration-agents to monitor/reconfigure non-Java components.

The Data Platform consists of a large ecosystem of heterogeneous, tunable, modules, each one possibly exposing a large number of monitored attributes/tunable parameters, and whose self-tuning could be driven by arbitrary utility functions. Another critical issue to be tackled is therefore the definition of a standardized mechanism to let the developers (or system administrators) externalize sufficient meta-data for each module of the Data Platform, in order to allow effective control by the Autonomic Manager. In particular, it has been decided to standardize a set of mechanisms to cope with the following issues/functional requirements:

1. To inform in a standard way the Autonomic Manager about which monitored attributes a tunable component exports, and which of its (configuration) options should be automatically tuned, also specifying what are their types and their admissible ranges (min,max) in case these have numeric domains.

2. To allow the Autonomic Manager to learn about which ones, among the monitored attributes exposed by a component, need to be considered Key Performance Indicators (KPI) for that component.

3. To advertise which ones, among the monitored attributes (possibly exposed by other components of the platform) are expected to affect the KPIs of a specific component of the Data Platform. These meta-data will allow, for instance, to increase the effectiveness of the so called feature selection phase [3] when adopting model free/black box control techniques (such as reinforcement learning techniques [4]) and to minimize the risks of over-fitting and the learning time.

4. To allow the specification of a set of utility functions that the autonomic manager should use to define the autonomic tuning strategy of a given component of the Data Platform.

5. To define a unique namespace for the attributes monitored at different layers of the platform to avoid naming clashes.

6. To ensure correctness of cross-referencing among attributes/KPIs/utility functions.

In order to fulfil the above requirements, it has been decided to standardize a XML-based meta-data encoding by defining the XML Schema reported in Appendix A. By merging the XML file descriptors of the various modules of the Data Platform, the Autonomic Manager will gather global knowledge of the ecosystem comprised by the tunable components of the Cloud-TM platform. Further, by validating the XML file descriptors against the pre-established XSD, it will be possible to safely detect integrity violations (such as collisions in the namespace and invalid cross-references).

Figure 3 provides an overview of the main elements of the XML file encoding the meta-data of a generic tunable component:

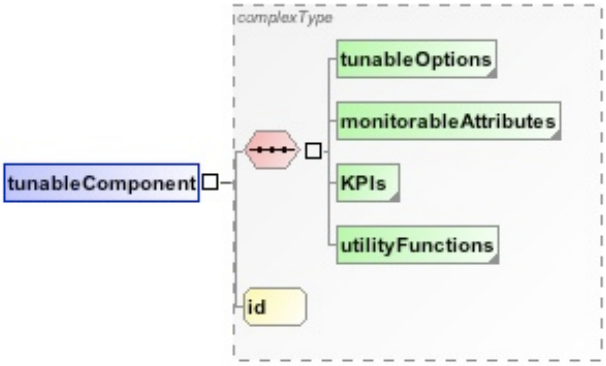



Figure 3: Main XML Elements of the File Descriptor of a Generic Tunable Component.

- a root tunableComponent element, which uniquely identifies the tunable component within the Autonomic Manager;
- a sequence of tunableOption elements, uniquely identified within the scope of their enclosing tunableComponent element and providing information on the nature of the configurable option (algorithm vs parameter) and on their valid domains;
- a sequence of monitorableAttribute elements, uniquely identified within the scope of their enclosing tunableComponent element and providing information on the type and valid domains of the monitored attributes;
- a sequence of KPIs, each of which:
  - is uniquely identified by a name attribute that refers to one of the enclosing tunableComponent's monitorableAttribute;
  - is associated with a sequence of relevantMonitorableAttribute, whose attributes id and componentName allow to define a cross-reference to a monitorable attribute possibly exposed by a different component of the platform;
- a sequence of relevantTunableOption, which are expected to affect the value of a KPI, and whose attributes id and componentName allow to define a cross-reference to a tunable parameter possibly exposed by a different component of the platform;
- a sequence of utilityFunction elements, where each of these elements specifies:

1. the utility function to be maximized by the Autonomic Manager class, identifying the class implementing the utility function's logic by the classpath attribute;
2. the input parameters for the utility function;
3. the priority of each utility function, which allows to define the relative weight of each utility function in case multiple (and possibly contrasting) optimization goals are defined.

An example of an XML descriptor advertising meta-data for the Replication Manager Component is provided in Listing 1 and Listing 2. This component informs the Autonomic Manager that it provides two tunable options: the replication algorithm and the command batching level. Further, it exports five monitorable attributes, namely AbortRate, AvgWritesPerTransaction, AvgCommitDuration, Throughput and AvgBytesSentPerTransaction. Next, two of these monitorable attributes, namely Throughput and AvgBytesSentPerTransaction are specified as KPIs, and, for each of them, the corresponding relevant monitored attributes and tunable parameters are defined. For instance, in order to optimize the Throughput KPI, the Autonomic Manager should monitor the AbortRate, CommitDuration and CPU_Utilization attributes (note that CPU_Utilization is an attribute exposed by a different component of the Cloud-TM platform, namely the underlying Virtual Machine). The Autonomic Manager is also informed about the fact that possible reconfiguration strategies that affect the Throughput KPI adjust the ReplicationAlgorithm, the BatchingCommandLevel and the CPU_Clock_rate of the underlying Virtual Machine.

Finally two utility functions are defined on the specified KPIs, namely maximizing the Throughput KPI and minimizing the AvgBytesSentPerTransaction KPI, giving higher priority to the former utility function.

**Listing 1: Example XML file descriptor for a Generic Tunable Component (part 1/2)**

```
<tunableComponent id="ReplicationManager">


  <tunableOptions>
    <tunableOption id="ReplicationAlgorithm">
      <algorithm>
        <!— describe available implementations of this Component—>
        <implementations>
          <implementation name="2PC"/>
          <implementation name="Primary Backup"/>
        </implementations>
      </algorithm>
    </tunableOption>

    <tunableOption id="BatchingCommandLevel">
      <parameter>
        <Type>Integer</Type>
          <minValue>1</minValue>
           <maxValue>100</maxValue>
      </parameter>
    </tunableOption>
  </tunableOptions>

  <monitorableAttributes>
    <monitorableAttribute>
      <name>AbortRate</name>
      <type>Double</type>
      <minValue>0</minValue>
      <maxValue>1</maxValue>
    </monitorableAttribute>
    <monitorableAttribute>
      <name>AvgWritesPerTransaction</name>
      <type>Double</type>
      <minValue>0</minValue>
      <!— not having specified a maxValue this means it is not upper
      bounded —>
    </monitorableAttribute>
    <monitorableAttribute>
      <name> AvgCommitDuration </name>
      <type> Double </type>
      <minValue> 0 </minValue>
    </monitorableAttribute>
    <monitorableAttribute>
      <name> Throughput </name>
      <type> Double </type>
      <minValue> 0 </minValue>
     </monitorableAttribute>
    <monitorableAttribute>
      <name>AvgBytesSentPerTransaction</name>
      <type>Double</type>
      <minValue>0</minValue>
    </monitorableAttribute>
  </monitorableAttributes>
 ...
```

**Listing 2: Example XML file descriptor for a Generic Tunable Component (part 2/2)**

```
...
<KPIs>
  <KPI name="Throughput">
    <relevantMonitorableAttributes>
      <relevantMonitorableAttribute id="AbortRate"
          componentName="ReplicationManager"/>
      <relevantMonitorableAttribute id ="AvgCommitDuration"
          componentName="ReplicationManager"/>
      <relevantMonitorableAttribute id ="CPU_Utilization"
          componentName="VirtualMachine"/>
    </relevantMonitorableAttributes>
    <relevantTunableOptions>
      <tunableOption id="ReplicationAlgorithm"/>
      <tunableOption id="BatchingCommandLevel"/>
      <tunableOption id="CPU_Clock_Rate" componentName="VirtualMachine"/>
     </relevantTunableOptions>
  </KPI>
  <KPI name="AvgBytesSentPerTransaction">
    <relevantMonitorableAttributes>
      <relevantMonitorableAttribute id ="AvgWritesPerTransaction"
          componentName="ReplicationManager"/>
      <relevantMonitorableAttribute id ="Net_UP/DOWN"
          componentName="VirtualMachine"/>
    </relevantMonitorableAttributes>
    <relevantTunableOptions>
      <tunableOption id="ReplicationAlgorithm"/>
    </relevantTunableOptions>
  </KPI>
</KPIs>

<utilityFunctions>
  <utilityFunction classpath="eu.cloudtm.optimizationFunctions.Maximize"
      priority="MAX">
    <argument>
      <type>KPI</type>
      <value>Throughput</value>
    </argument>
  </utilityFunction>
  <utilityFunction classpath="eu.cloudtm.optimizationFunctions.Minimize"
      priority="NORMAL">
    <argument>
      <type>KPI</type>
      <value>AvgBytesSentPerTransaction</value>
    </argument>
  </utilityFunction>
</utilityFunctions>

</tunableComponent>
```

## 3.2 Data Platform APIs

As depicted in Figure 1, the APIs exposed by the Data Platform will entail:

- The Object Grid Mapper. A key element of the Cloud-TM platform is that it should support the development of object-oriented programs based on object domain models, that is programs that maintain their states as sets of entities, which are represented by instances of various classes with relationships among them.

  As we will discuss more detailedly in Section 3.3, the Transactional In-Memory Data Grid component of the Cloud-TM architecture is a key-value store and its API is not the most adequate for a programmer that wants to store a complex graph of entities from many different types. So, the proposed architecture of the Cloud-TM platform includes a layer on top of the Transactional In-Memory Data Grid that is responsible for providing the higher-level API needed to develop an application that is based on an object-oriented domain model.

- The Searcher. Any complex data-centric application requires supporting ad-hoc queries to retrieve and manipulate portions of the state that it manages. Given that we want the Data Platform to provide support for development of object-oriented applications, the module in charge of implementing the querying functionality should be able to deal with some intrinsic aspects of the object-oriented model, e.g. supporting notions such as polymorphism and inheritance.

  These functionalities will be implemented by the Search API component. This component will expose to the programmer the Java Persistent API - Query Language (JPA-QL) interface, which represents, at the time of writing, the industry standard for encoding queries on an object-oriented data base at least for what concerns the Java platform. This same API will be used also to support advanced full-text queries, supporting notions such as ranked searching, search by fields, proximity queries, phrase queries etc.

  Under the hood, this component will rely on an innovative design strategy that will integrate some of the leading open-source projects in the area of data management and indexing, namely Hibernate OGM and Lucene, with a fully fledged distributed query engine providing support for complex data manipulation/queries (such as joins or aggregate queries).

- The Distributed Execution Framework. This framework will provide a set of APIs aimed at simplifying the development of parallel applications running on top of the Cloud-TM platform. It will essentially consist of two main parts:

  1. An adaptation of the java.util.concurrency framework, providing a set of building blocks for the development of classic imperative parallel applications deployed on the Cloud-TM platform. These will include abstractions such as task executors and synchronizers (e.g. countdown latches) and transaction-friendly concurrent data collections.

2. An adapted version of the popular Google's MapReduce framework, that will allow developers to transparently parallelize their tasks and execute them on a large cluster of machines, consuming data from the underlying in-memory data grids rather than using input files as it was defined by the original proposal.

In the following subsections, we describe in more detail each of these modules.

### 3.2.1 Object grid mapper

The Object Grid Mapper module is responsible for implementing a mapping from an object-oriented domain model to the Infinispan's key-value model.

Thus, a programmer that is developing an application to execute in the Cloud-TM platform will use the API provided by this module not only to develop his application's object-oriented domain model, but also to control the life-cycle of the application's entities.

**3.2.1.1 Two different APIs for the Object Grid Mapper** Given the maturity of the Java Persistence API (JPA) and its wide acceptance by Java software developers as the API to use to access persistent objects, we decided that the Cloud-TM platform should provide an implementation of the JPA standard as one of its options for mapping an object-oriented domain model. The adoption of JPA will make the Cloud-TM platform standards compliant, easing its adoption by the masses of programmers already familiar with JPA, and providing a smoother migration path for those applications already built on top of JPA. Hibernate OGM, discussed below, is our implementation of JPA for the Cloud-TM platform.

But within the consortium, we have also discussed the possibility of having a second implementation for the Object Grid Mapper module, based on INESC-ID's previous work on the Fénix Framework. More specifically, an implementation that relies on the Domain Modeling Language (DML) to specify the domain model. This will allow us to experiment with a different way of exposing to the application-level programmer a programming model that allows the creation of a transactional, distributed, replicated, persistent domain model, even considering that having two different APIs has its costs (at least, making it more confusing to the application programmer). But having more than one API is not uncommon: for instance, in J2EE a programmer may use either JDBC or JPA to access a relational database; another example is the use of transactions in JPA, where there is more than one way to do it.

The main reason for having an alternative is because JPA imposes some constraints on the implementation of the mapping from objects to Infinispan that may difficult or completely prevent some of the approaches that we would like to explore in the Cloud-TM project. To illustrate why, let us consider the self-tuning mechanisms that constitute a central theme of Cloud-TM.

We envision that some of those self-tuning mechanisms may be present at the mapping-level also. For instance, by changing the schema of the classes that implement a domain model so that less frequently used data is put on helper classes whose instances do not need to be loaded by a certain node, or by using different types of collection classes, depending on the size of those collections or on how those collections are accessed by the application.

These changes are difficult to implement in a JPA setting because the structure of the domain model is fixed by the application programmer, who implements the Java classes that represent the application's entities. For instance, imagine that the programmer creates a class Person with 80 slots, but that, among these 80 slots, only 3 are often used. In that case, we would like to be able to implement the Person class with those 3 slots only and another slot pointing to an instance of a helper class containing the other

77 slots. This way, by way of lazy loading, we could spare loading all of the slots of a Person on a certain node of a cluster. Unfortunately, this kind of adaptation is not easy (if at all possible) to do in JPA.

Like this example, there are several others that we would like to explore in the Cloud-TM platform and that led us to decide having a second API for the application programmer.

We will start with what we have now in the Fénix Framework, but extend it in the directions that may seem better suited to a cloud computing platform. Ideally, this may even be the foundations of a future standard, much as Hibernate laid the path to JPA.

To sum it up, we would like to explore the following directions with this alternative implementation of the mapping are the following:

- different layouts for classes implementing the domain model's entities, as discussed above;
- different implementations for collections representing associations;
- using an STM-based approach for ensuring isolation among threads accessing the same object, rather than the copy-on-read approach currently used by Hibernate;
- different models of nested transactions, including the possibility of having parallel nesting;
- layering a different transactional system on top of Infinispan, so that we may have different consistency guarantees than those provided by Infinispan (for instance, strict serializability for certain more critical operations, without forcing Infinispan to support more costly consistency levels);
- using an STM for node-local fast consistency checks that may use knowledge about the structure of the domain model to reduce the conflicts among transactions, which may be more difficult or impossible to do when the objects are split into cells of Infinispan (a common example of this is operations on collections representing associations, but there are others).

To conclude, we point out that some of these things may be possible to implement in Hibernate OGM, but we believe that they are much harder to prototype there than in the Fénix Framework. Partly because some of these things are already available in the Fénix Framework, and partly because trying to prototype some of the above ideas in a JPA implementation would be much more difficult because we will need to be concerned with many more restrictions imposed by JPA.

Instead, we intend to use the Fénix Framework implementation as a vehicle of rapid prototyping of these ideas to test them and see if they give good results. If so, then they may be incorporated into Hibernate OGM, to benefit also the JPA users.

**3.2.1.2 Hibernate Object Grid Mapper** Hibernate OGM is the aggregation of a few key components (see Figure 4):

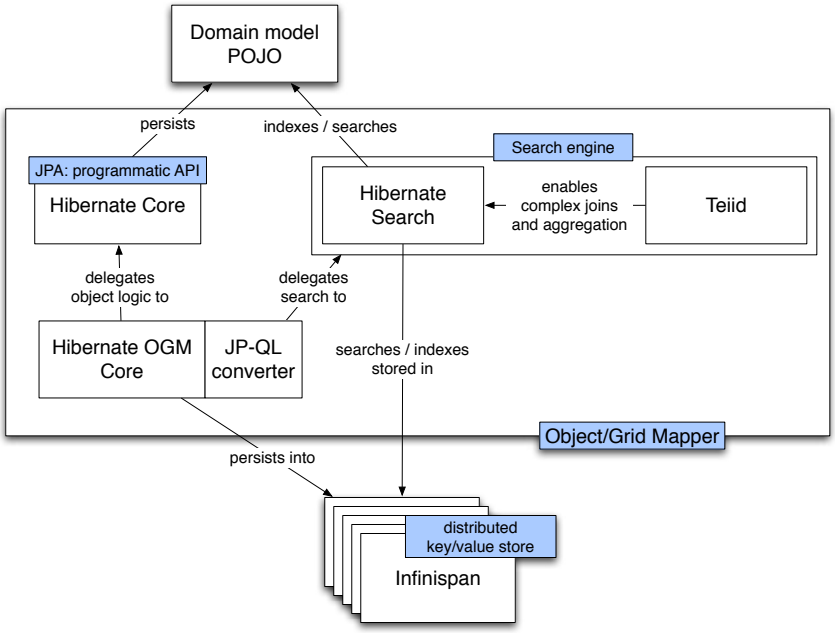



Figure 4: Architectural Overview of Hibernate Object Grid Mapper

- Hibernate Core for JPA support
- Hibernate Search for indexing purposes
- Teiid (future) for advanced queries
- Infinispan for the data grid
- Infinispan's Lucene directory to store indexed in Infinispan itself
- Hibernate OGM itself

Hibernate OGM reuses as much as possible from the Hibernate Core infrastructure to limit the number of youth bugs and the massive investment inherent to writing a JPA implementation from scratch.

Hibernate Core exposes the notion of:

- Persisters: responsible for executing Create Update and Delete operations related to a given entity or a given collection. These operations are triggered by the higher levels of the Hibernate Core execution engine.
- Loaders: responsible for executing Read operations related to a given entity or a given collection. These operations are triggered by the higher levels of the Hibernate Core execution engine.

These two elements are the core piece interacting with JDBC (i.e. the relational interface) in Hibernate Core.

The Persisters and the Loaders have been rewritten in Hibernate OGM to persist data using Infinispan instead of JDBC. These implementations are the core of Hibernate OGM. We will see later how the data is structured. Other than that, all higher level logics for Create/Read/Update/Delete (CRUD) operations are implemented by Hibernate Core.

To see how JP-QL queries are implemented, the reader should check section 3.2.2.

Hibernate OGM best work in a JTA environment. The easiest solution is to deploy it on a Java EE container. Alternatively, you can use a standalone JTA TransactionManager.

Let's now see how and in which structure data is persisted in the NoSQL data store.

**How is data persisted** Hibernate OGM tries to reuse as much as possible the relational model concepts (at least when they are practical and make sense). For very good reasons, the relational model brought peace in the database landscape over 30 years ago. In particular, Hibernate OGM inherits the following philosophy traits:

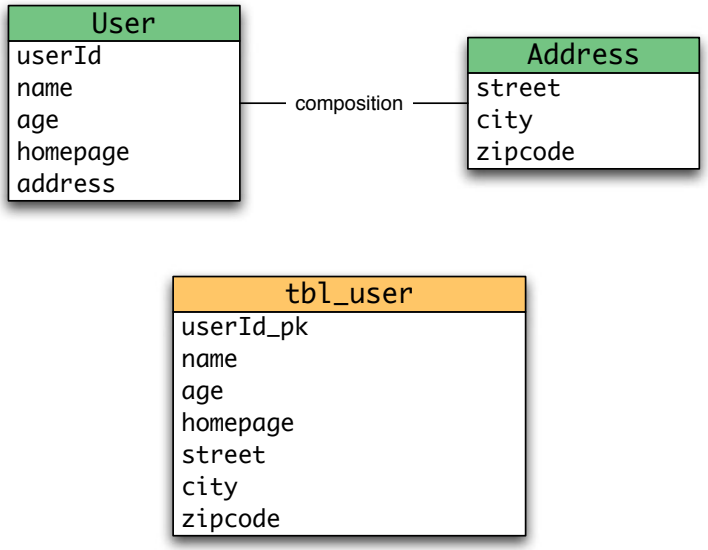
- abstraction between the application object model and the persistent data model;
- persist data as basic types;
- keep the notion of primary key to access an entity;
- keep the notion of foreign key to link two entities.

If the application data model is too tightly coupled with the corresponding persistent data model a few issues arise including:

- any change in the application object hierarchy / composition must be reflected in the persistent data;
- any change in the application object model will require a migration at the persistent level;
- any access to the data by another application instantly tied both applications;
- any access to the data from another platform become somewhat more challenging.

Entities are stored as tuples of values by Hibernate OGM (see Figure 5). More specifically, each entity is represented by a Map<String,Object> conceptually where the key represents the column name (often the property name but not always) and the value represents the column value as a basic type. Hibernate OGM favours basic types over complex ones to increase portability (across platforms and across type / class schema evolution over time). For example a URL object is stored as its String representation.

The key where a given entity instance is stored is composed of:




| key | value |
|---|---|
| tbl_user,userId_pk,1 | {userId_pk=1,<br>name="Emmanuel",<br>age=33,<br>homepage="http://emmanuelbernard.com",<br>address.street="rue de la Paix",<br>address.city="Paris",<br>address.zipcode="75008"} |

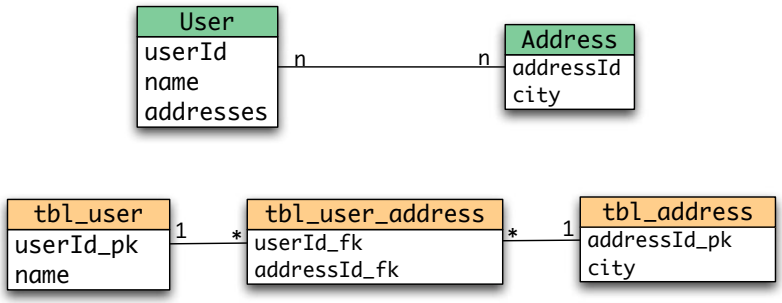
Figure 5: Example of mapping of entities and unidirectional associations in a data grid

- the table name
- the primary key column name(s)
- the primary key column value(s)

An entity is addressable by a single lookup.

Associations are also stored as tuples, or more precisely as a set of tuples. To ensure that associations are reachable via key lookups from the entity hosting them, Hibernate OGM stores one copy of the association data, allowing Hibernate OGM to resolve the navigation. In case of bidirectional relationship (see Figure 6), Hibernate OGM will duplicate information.

The key in which association data are stored is composed of:




| key | value |
|---|---|
| tbl_user,userId_pk,1 | {userId_pk=1,name="Emmanuel"} |
| tbl_user,userId_pk,2 | {userId_pk=2,name="Caroline"} |
| tbl_address,addressId_pk,3 | {addressId_pk=3,city="Paris"} |
| tbl_address,addressId_pk,5 | {addressId_pk=5,city="Atlanta"} |
| tbl_user_address,userId_fk,1 | { {userId_fk=1, addressId_fk=3}, {userId_fk=1, addressId_fk=5} } |
| tbl_user_address,userId_fk,2 | { {userId_fk=2, addressId_fk=3} } |
| tbl_user_address,addressId_fk,5 | { {userId_fk=1, addressId_fk=5} } |
| tbl_user_address,addressId_fk,3 | { {userId_fk=1, addressId_fk=3}, {userId_fk=2, addressId_fk=3} } |

Figure 6: Example of mapping of entities and bidirectional associations in a data grid

- the table name;
- the column name(s) representing the foreign key to an entity;
- the column value(s) representing the foreign key to an entity.

Using this approach, Hibernate OGM favours fast read and (slightly slower writes).

Note that this approach has benefits and drawbacks:

- it ensures that all CRUD operations are doable via key lookups;
- it favours reads over writes (for associations);
- but it duplicates data.

Infinispan is a key/value store, which means that per se it has no notion of a schema. Likewise, the tuple stored by Hibernate OGM is not tied to a particular schema: the

tuple is represented by a Map, not a typed Map specific to a given entity type. Nevertheless, JPA does describe a schema thanks to:

- the class schema;
- the JPA physical annotations like @Table and @Column.

Even though this approach requires explicitly encoding the data schema into the application, it offers some robustness and explicit understanding when the schema is changed as the schema is right in front of the developers' eyes. This is an intermediary model between the strictly typed relational model and the totally schema-less approach pushed by some NoSQL families.

**3.2.1.3 Fénix Framework Object Grid Mapper** The Fénix Framework was originally designed with the goal of simplifying the development of Java-based applications that need a transactional and persistent rich domain model. To do this, the Fénix Framework hides the details of accessing an external database management system, while providing transparent persistence and strong consistency guarantees to the various threads executing concurrently over the domain model's entities.

The two key elements of the Fénix Framework are the JVSTM, a Java library implementing a multi-version Software Transactional Memory, and the Domain Modeling Language (DML), a domain-specific language created to specify the structure of an object-oriented domain model.

Programmers using the Fénix Framework specify the structure of the application's domain model using the DML. Namely, they declare the value types that they want to use in their domain model, describe the domain's entities in terms of properties, and specify which relationships exist among entities. The DML compiler transforms a domain model specification written in DML to a set of Java classes that implement the structure of that domain model and that may be further extended by the programmer, as shown in Figure 7. The key idea here is that the classes generated from the DML code incorporate all the code needed to make their instances persistent and transactional-safe.

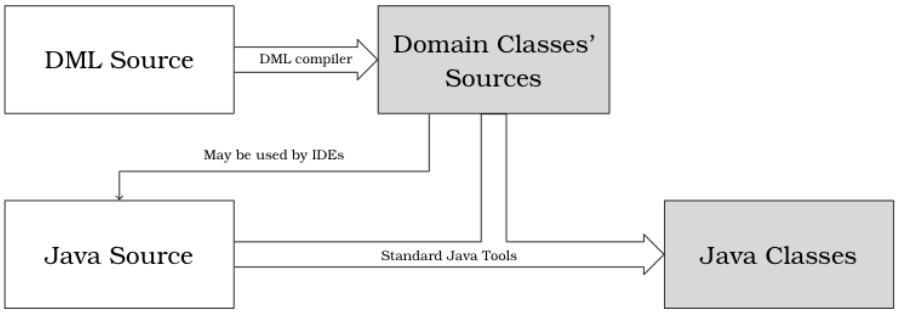



Figure 7: Data Flow Among the Main Components of the Fènix Framework Object Grid Mapper.

Besides using the DML to implement the structure of the domain model, programmers using the Fénix Framework API just need to specify which methods must execute atomically.

The Fénix Framework API is, thus, quite minimalist, and that is one of its strengths for the Cloud-TM project, as it gives us great freedom to implement the domain model in many different ways, depending on the applications characteristics.

In fact, programmers using the Fénix Framework cannot assume anything about how classes representing domain entities are implemented, except that they provide a certain interface determined by the semantics of the DML language. For instance, specifying the User entity of the previous section in DML will generate a class that provides in its interface the usual getters and setters for each of its properties (namely, the properties name, age, homepage, and address), as illustrated in Figure 8. That does not mean, however, that the class will have actual slots for each of those properties. That is one possible implementation. Another is to bundle several properties into an auxiliary inner class, for instance. Still, the programmer will always access the entity in the same way.

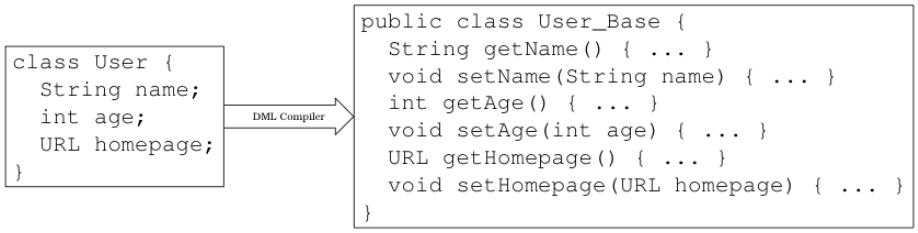



Figure 8: Example Transformation of a Simple Entity by the DML Compiler.

**Classes layout** In our current prototype of the Fénix Framework for the Cloud-TM platform, we still rely on the standard behaviour of the DML compiler, which generates Java classes that use the vboxes of the JVSTM to store the mutable state of each entity.

However, we have support for two different layouts for the classes representing entities:

- each property of an entity corresponds to a slot in the class, wrapped by a vbox, as shown in Figure 9;
- all properties are kept into a single vbox for the entire object, which contains a instance of a static class containing all of the properties of the object, as shown in Figure 10.

The choice of the layout used affects the memory usage and the concurrency control granularity.

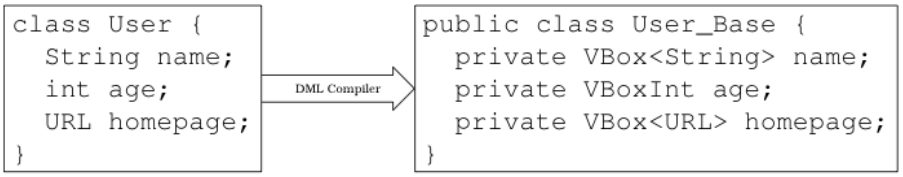



Figure 9: One-to-one mapping between object's properties and JVSTM's VBoxes

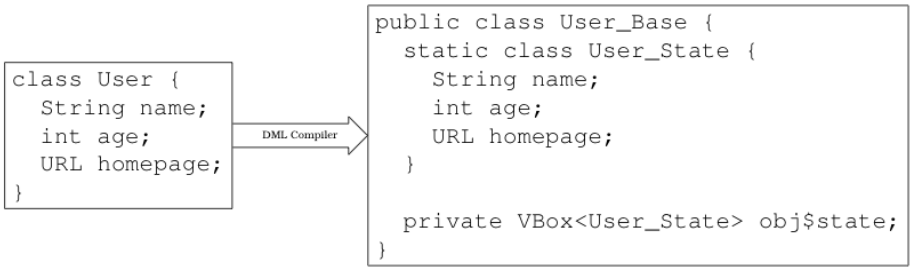



Figure 10: Mapping the whole state of an entity into a single JVSTM's VBox

**Mapping objects to Infinispan** The mapping currently done by the Fénix Framework is from vboxes to keys in Infinispan. Each vbox is written to Infinispan into a key that is composed by the slot name containing the box and the OID of the object to which the vbox belongs to. So, for example, using the multi-box layout, the name of a User with OID 1 will be stored under the key "name:1".

As all mutable data is stored within a vbox, including collections representing relationships among objects, the mapping from an object-oriented domain model to Infinispan is uniquely determined by how vboxes are used to represent the domain model's state. So, when we change the layout of the classes that represent entities, we change also the mapping done to the underlying Infinispan.

**Direct mapping to Infinispan** Even though we currently rely on the vboxes to hold the mutable state of each entity, thereby making use of the JVSTM to ensure the transactional properties of the operations accessing the domain objects, an alternative is to map objects directly into Infinispan. This approach has the advantage of depending only on the transactional support provided by Infinispan, accessing the Infinispan cache whenever an object is accessed within a transaction. We intend to explore this approach as an alternative and compare it to the current approach.

### 3.2.2 Search APIs

A couple of search APIs will compose Cloud-TM. They address different needs and can be implemented at a different rate in Cloud-TM minimizing the project risks.

The first supported API is support for full-text search queries as provided by Hibernate Search. The second API is support for JP-QL (initially a subset and in a second step, a more complete subset).

#### Full-text queries and Hibernate Search

Hibernate Search is an Object/Index Mapper: it binds Apache Lucene full-text index technology to the Object Oriented world.

Hibernate Search consists of an indexing and an index search component. Both are backed by Apache Lucene.

Each time an entity is inserted, updated or removed in/from the database, Hibernate Search keeps track of this event (through the Hibernate Core event system) and schedules an index update. All index updates are handled without you having to use the Apache Lucene APIs. This makes the data in the datastore inlined with the index information and thus makes queries accurate.

To interact with Apache Lucene indexes, Hibernate Search has the notion of DirectoryProviders. A directory provider will manage a given Lucene Directory type. Traditional directory types are File-System directories and In-memory directories. For Cloud-TM we will make use of an Infinispan directory: the index information will be stored in a distributed fashion in the data grid. The data store will then contain both the data and the meta-data associated to it. See InfinispanDirectory for more information.

Hibernate Search uses the Lucene index to search an entity and return a list of managed entities saving you the tedious object to Lucene document mapping. The same persistence context is shared between Hibernate Core and Hibernate Search. As a matter of fact, the FullTextEntityManager is built on top of the JPA EntityManager so that the application code can use the unified javax.persistence.Query API exactly the same way a JP-QL query would do.

To be more efficient Hibernate Search batches the write interactions with the Lucene index. There are currently two types of batching. Outside a transaction, the index update operation is executed right after the actual database operation. This is really a no batching setup. In the case of an ongoing transaction, the index update operation is scheduled for the transaction commit phase and discarded in case of transaction rollback. The batching scope is the transaction. There are two immediate benefits:

- Performance: Lucene indexing works better when operation are executed in batch.
- ACIDity: The work executed has the same scoping as the one executed by the database transaction and is executed if and only if the transaction is committed. This is not ACID in the strict sense of it, but ACID behaviour is rarely useful for full text search indexes since they can be rebuilt from the source at any time.

You can think of those two batch modes (no scope vs transactional) as the equivalent of the (infamous) autocommit vs transactional behaviour in JDBC. From a performance perspective, the in transaction mode is recommended. The scoping choice is made transparently. Hibernate Search detects the presence of a transaction and adjust the scoping.

It is recommended - for both your database and Hibernate Search - to execute your operations in a transaction, be it JDBC or JTA.

### Infinispan Lucene Directory

#### *The Directory API in Lucene*

The Lucene Directory represents the interface between Lucene's index writer and it's search engine to interact with a storage system where the index is kept. In the Lucene distributions there are two main implementations: an FSDirectory and a RAMDirectory, which store the index elements on the filesystem or in a concurrent hashmap.

An FSDirectory instance is created by invoking a static helper, and in practice a different implementation might be created - all options obviously extend FSDirectory - according to the operating system being used: you might get for example a NIOFSDirectory or a MMapDirectory, among other implementations. The exact implementation returned varies among Lucene releases: the factory attempts to provide the optimal implementation but might discard some options because of known problems at the time in which the Lucene version was released.

The Directory API is not a complete abstraction of the index structure, only of it's storage. So for example it leaks the file concept, file timestamps, directory listing and the expected contents of these elements are arrays of bytes; IOExceptions are expected, and all of Lucene's design requires the "filesystem" to implement correctly a "delete on last close" semantic: Lucene might delete files still being used by some search process, as most filesystem implementations will make sure the file is not actually deleted until the last file handle is closed. This guarantee is often not provided by many network-shared filesystems, or poorly implemented / hard to setup properly, and because of this it's problematic to share a Lucene index across multiple servers.

A Directory needs to store and load index segments. When an index segment is completely written and it's outputstream is closed, then it will never change: index updates including deletions are managed by adding additional information to new segments, and periodically by configurable optimization strategies old segments are deleted. A single segment can be written as a single file (compound format), or have it's separate logical areas written in separate files; in such a case the filename will represent the segment generation and a three characters file extension will mark the kind of the file. In addition to these segment files which contain the actual index data, two additional files contain general meta-data information about the index; these don't have the same immutable guarantees as the segment files.

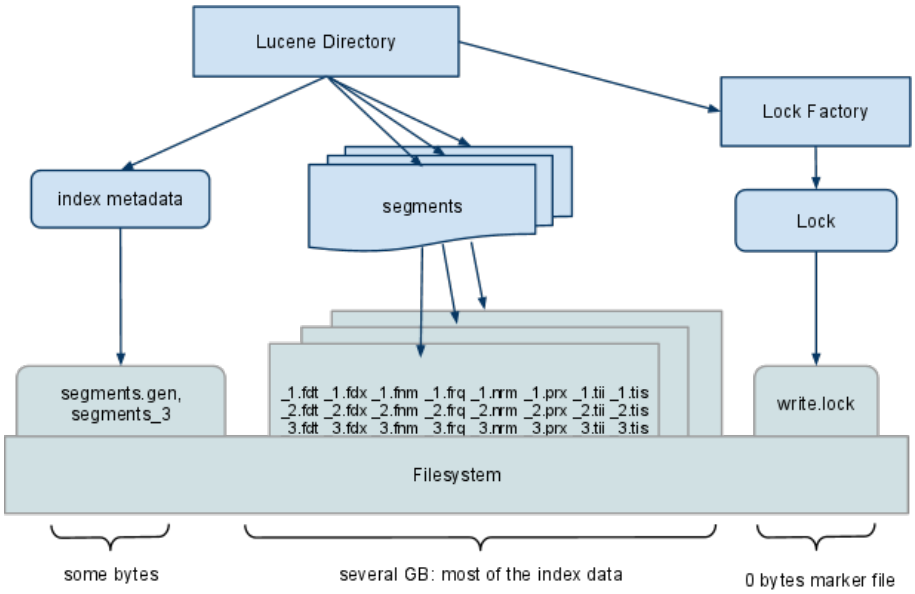



Figure 11: Lucene Directory

The directory structure of Lucene assumes that a single IndexWriter instance will apply changes to it at any time; so to protect against multiple writers a locking strategy must be selected as well. When using a filesystem based Directory, tipically a filesystem based LockFactory such as NativeFSLockFactory will be used, but in practice any combination of Directory and LockFactory implementations are expected to work together. Such a NativeFSLockFactory would use marker files stored in the index directory, again relying on proper filesystem semantics.

*Infinispan Lucene Directory*

The Infinispan Directory closely mimicks the filesystem behaviour, extends Lucene's org.apache.lucene.store.Directory and implements all methods closely remapping the file-based API to Infinispan stored objects, it doesn't change any of the assumptions Lucene has about it's storage and so the resulting implementation does not prevent to use any advanced Lucene feature; also because the Directory API proved historically quite stable the current version 5.0.0.CR2 was tested compatible with all Lucene versions 2.4.x, 2.9.x, 3.0.x, 3.1.x.

Comparing the Infinispan Directory to it's filesystem based counterpart, there are three notable differences:

1. Files are split in two main objects:

    (a) file contents (a byte[]);

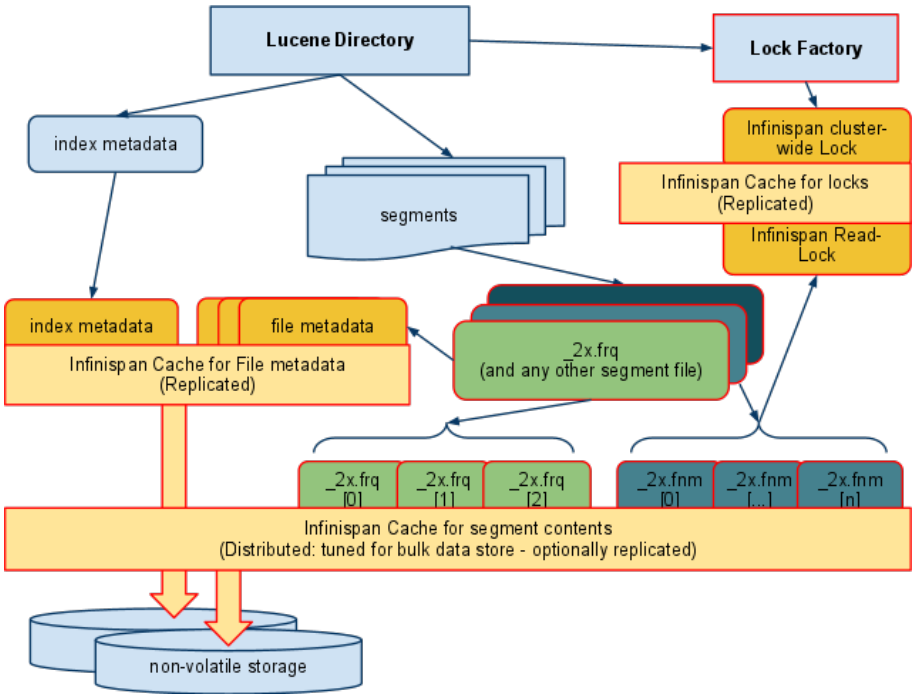



Figure 12: Infinispan Lucene Directory

(b) file meta-data (timestamp, length, . . . ).

2. Additional Infinispan specialized Lock Factories are added.

A single file contents might be fragmented in smaller parts

Technically any other LockFactory could be used, as long as it's able to guarantee that no two IndexWriters will be opened at the same time.

*Performance considerations*

Because the file meta-data is read very frequently and because it is significantly smaller than the file contents, it's recommended to use a replicated Cache to store this information. For the segments contents instead, a distributed cache is recommended so that it won't be required to keep a full copy of the index in the JVM memory of each node; still using replication is an option in case of small indexes or plenty of available memory.

The locks cache uses a third cache, because it's likely that the first two caches will be configured to use a CacheLoader, to keep a copy of the index as a backup in some

permanent storage, while the locks meta-data is frequently changing information for which we don't need to bother the slower storage engines; especially because if we where to shut down all Infinispan nodes, the locking information can be lost, so that in the remote case of all nodes crashed the service could be restarted without any leftover lock.

While the locking information and the file meta-data are quite small, the index segments might get very big. In Infinispan it's better to have many smaller values than a couple of very big elements; smaller values are essentially simpler to transfer around in the network and make sure that some random JVM won't have to keep the whole segment value in memory; so the InfinispanDirectory takes an additional parameter called chunkSize and it will automatically split values larger than this amount of bytes in several chunks.

Splitting segments in chunks has however two drawbacks:

- Searching will be slower. Lucene is not aware of this buffer split and we might introduce some network latency while fetching the next chunk from a remote node; this doesn't happen very frequently, and Infinispan can enable a L1 cache too, but it's on a hot path in Lucene's search process.
- Deleting of a segment is no longer an atomic operation. If the Infinispan Directory detects that a searcher is opening a segment which was split in more than a single part, it will acquire a cluster-wide "ReadLock".

Both operations are performed transparently and nor Lucene nor the user has to deal with it; but it's worth spending some effort to tune the system in such a way that a good tradeoff is found between number of values around the cluster and fragmentation.

If Lucene's IndexWriter was configured to use the LogByteSizeMergePolicy, this can be configured to generate segments which don't exceed a specified size; the strategy uses some approximate estimates tough, so it's still needed to do some experiments to find a good value; for all segments which where produced of a size below the specified chunksize the readlock won't be acquired. Also, the MergePolicy only applies to segments which are merged together; size of newly created segments is affected by the ramBufferSizeMB and the maxBufferedDocs parameters of the IndexWriter.

The Infinispan Directory will use the Infinispan based LockFactory by default, but it's possible to provide a different implementation. Also while it expects three caches to be used for the above mentioned purposes, when providing a custom LockFactory only two caches are needed.

*Permanent storage*

Infinispan has a notion of CacheLoaders; this is a quite simple interface which provides some simple Map operations; in contrary to the name suggestion, it's not only able to load information from the implementation but it's also able to store it. What is of particular interest of the Lucene Directory is that it's able to use Infinispan as a distributed write-behind implementation: the write operations to the cluster

can be synchronous, but still have the write operations to the CacheLoader handled asynchronously, and when it will write to the store it will only write the latest current state for any key. This means we can "shield" slower storage services from too many changes, which improves performance and in the case of some cloud services lowers the IO costs. It's worth pointing out that while the storage is asynchronous, the information is safely stored in the grid, so if the writer happens to crash before the values are stored, another Infinispan node will still have the data and will be able to load and store it.

Because some locking meta-data is really not needed to be stored, and because it changes frequently, it's adviseable to enable the CacheLoader only on the two main caches.

Infinispan provides many reference implementations for such a CacheLoader: load-/store to Amazon S3 or any other JClouds supported cloud storage system, to a database using a JDBC datasource, to a filesystem, to Apache Cassandra; of course custom implementations can be added to the configuration as well.

*Usage*

To instantiate a org.infinispan.lucene.InfinispanDirectory the main constructor requires the three cache instances. Technically it is possible to pass it a single cache instance three times, just you will loose the ability to tune and configure different cache properties so this is not advisable unless for unit testing. There are other constructor alternatives to override the default chunkSize and to define a custom LockFactory; in this case only two caches are needed.

Multiple InfinispanDirectory(s) can use the same caches: to differentiate indexes an indexName parameter is provided, so it's possible to store multiple indexes in the same caches as long as they are using different names; this can be considered an analogy of the path as used by the filesystem implementation.

When an InfinispanDirectory is created, it extends org.apache.lucene.store.Directory and all normal Lucene APIs can be used to interact with it.

So several different JVM can share the same in memory Lucene Index and providing very quick replication of index changes to the other nodes.

The limitation in this design is that while it scales pretty well the search aspect, it doesn't scale the writer: the IndexWriter will be faster than on disk as it's writing now in memory, but you're still limited to a single IndexWriter at a time; Hibernate Search deals with this limitation by providing a queue of changes, a master node is selected and all changes are sent to this node to be applied on behalf of the other nodes. Other applications not using Hibernate Search will have to implement something similar, or design the application in such a way that it will always write from a single node.

**Support of JP-QL** For Hibernate OGM the primary query API is JP-QL. JP-QL support will be implemented in two steps:

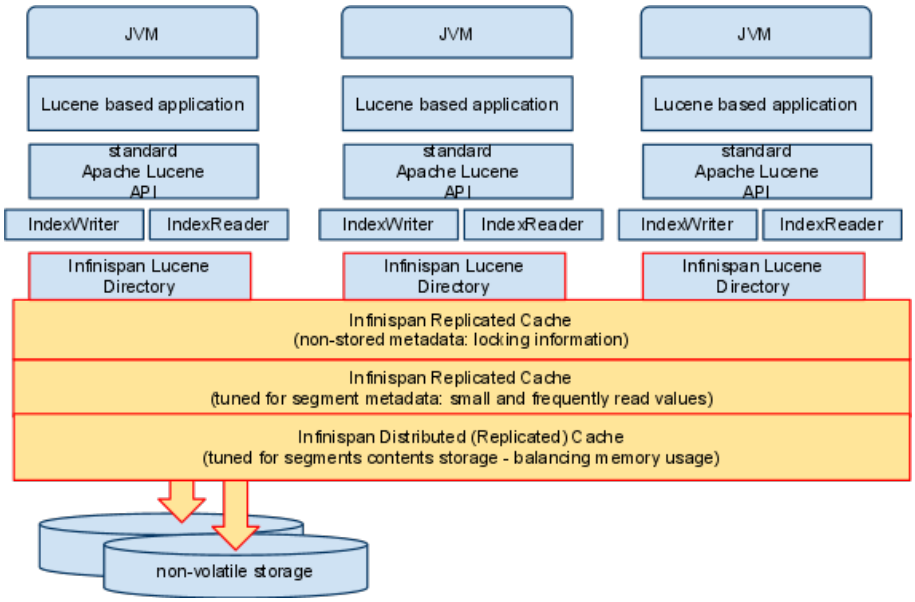



Figure 13: Infinispan Lucene Directory instantiation

- convert JP-QL queries into Lucene queries and execute them via Hibernate Search;
- convert JP-QL queries into Teiid queries where Teiid sources will be the various Lucene indexes exposed via Hibernate Search.

*JP-QL to Lucene query*

The first step is to write a JP-QL parser that converts JP-QL queries into one or more Lucene queries (Hibernate Search queries really to benefit from the right level of abstraction). While this approach is limited in what JP-QL queries are supported, we can already cover the following functionalities:

- restrictions on the data set ( where user.age > 30, where order.price > 10 and user.city = 'Paris' )
- some simple *-to-one joins

The second step is to use Teiid as the query engine.

*JP-QL to Teiid query*

Teiid is a data virtualization system that allows applications to use data from multiple, heterogeneous data stores.

The heart of Teiid is a high-performance query engine that processes relational, XML, XQuery and procedural queries from federated datasources. Features include

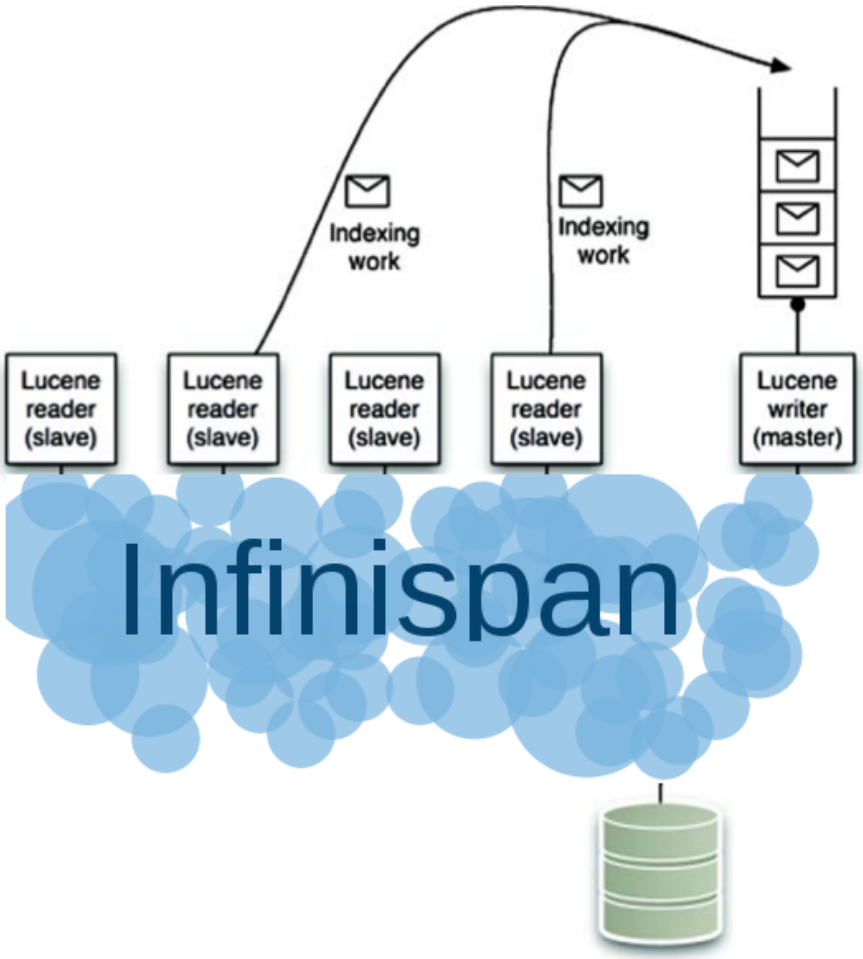



Figure 14: Queueing mechanism for changes

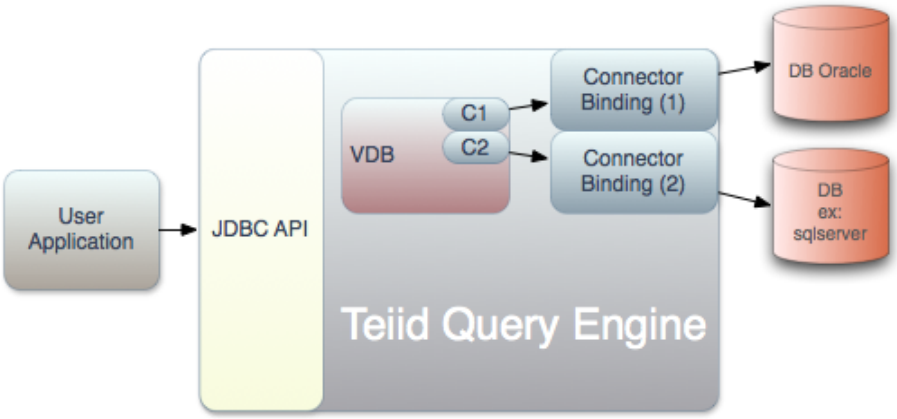



Figure 15: Architectural Overview of Teiid Query Engine

support for homogeneous schemas, heterogeneous schemas, transactions, and user defined functions.

Using Teiid, the query execution would:

- convert the JP-QL query into a Teiid query tree;
- let Teiid convert the query tree into one or more Hibernate Search / Lucene queries;
- execute Lucene queries on each dedicated index;
- aggregate and or do the cartesian join if necessary;
- return the results to Hibernate OGM.

This second step is a bit further in time but definitely exciting as we would be able to do more complex join operations including for *-to-Many associations.

### 3.2.3 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 balancing, thread synchronization and scheduling, fault-tolerance, asymmetric processing speeds in different nodes.

The Distributed Execution Framework will consist of two main parts.

1. An extension of the java.util.concurrency framework, designed to transparently support, on top of the in-memory transactional data grid:
   - execution of Java's Callable tasks, which may consume data from the underlying data grid;
   - synchronization of tasks/threads via, e.g., queues, semaphores, countdown latches and other classic concurrency tools;
   - concurrent, transactional data collections, such as sets, hashmaps, skiplists, arrays;
   - processing of data streams, via the definition of pipelines of data stream processing tasks.
2. An adapted version of the popular Google's MapReduce framework. MapReduce is a programming model and a framework for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate k. MapReduce framework enables users to transparently parallelize their tasks and execute them on a large cluster of machines.

   The framework is defined as "adapted" because the input data for map reduce tasks is taken from the underlying in-memory data grids rather than using input files as it was defined by Google's original proposal.

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 load balanced (using a Consistent Hashing scheme [5]). Since input data is already balanced execution tasks will be automatically balanced 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.

  Both node failover and task failover policy will be pluggable. An initial implementation will define interfaces to implement various node failover policies but we will provide only a simple policy that throws an exception if a node fails. In terms of task failure the default initial implementation will simply re-spawn the failed task until it reaches some failure threshold. Future implementations might migrate such a task to another node, etc.

- As the in-memory transactional data grid will provide extensive support for data replication, more than one node can contain the same entry. Running the same task on the same node would lead to redundant computations, useful possibly for fault-tolerance reasons, but otherwise superfluous. In order to avoid this problem, it is necessary to schedule the processing of disjoint input data entries in nodes, keeping into account the fact that they possibly maintain replicas of the same data. Thus the input entries for the task have to be split by the framework behind the scene. Once split, approximately the same number of non-overlapped input entries will be fed to the task on each replica.

  Ideally, the split input entries don't need to be merged or re-split. However, in reality, there are two cases that need to be considered.

  The task might run faster on a certain replica while slower on some others even if the input entries were split evenly depending on how the task was implemented by user. In such a case, the idle nodes could process the input entries which were not processed yet by the busy nodes (i.e. work stealing), only if the cost of work-stealing does not cancel the gain.

  Also, during the task execution, a replica can go offline for some reason. In such a case, other replicas have to take over the input entries that the offline replica was assigned to. This is basically implemented in the same way with work stealing because we can consider the offline node as "the busiest one".

  Note that, once the input entries are fed to a node, they can split again to fully utilize all CPU cores. Proper scheduling needs to be done so that only the same number of threads with the number of available cores run at the same time to avoid excessive context switching. Work stealing also needs to be implemented to address the problem mentioned above in a local level.

**3.2.3.1 Distributed Execution Model** The main interfaces for distributed task execution are DistributedCallable and DistributedExecutorService. DistributedCallable (see Listing 3) 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 form of key/value pairs (see Section 3.3). 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.

**Listing 3: DistributedCallable interface**

```
public interface DistributedCallable<K, V, T> extends Callable<T> {

 /**
 * Invoked by execution environment after DistributedCallable
 * has been migrated for execution to a specific Data Platform's
 * node.
 *
 * @param cache
 *       Data Platform's store whose keys are used as input data for
 *       this DistributedCallable task
 * @param inputKeys
 *       keys used as input for this DistributedCallable task
 */

 public void setEnvironment(Cache<K, V> cache, Set<K> inputKeys);

}
```

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 node, 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 Infinispan would execute DistributedCallable on all keys for a specified data store.

**3.2.3.2 Map Reduce Execution Model** The MapReduce model supported by the Cloud-TM platform is an adaptation of Google's original MapReduce. There are four main components in each map reduce task: Mapper, Reducer, Collator and MapReduceTask.

Each implementation of a Mapper interface (see Listing 4) is a component of a MapReduceTask that is invoked once for each input entry <K,V>, namely a key/value pair of the underlying in-memory data grid. Given a cache entry <K,V> input pair, every Mapper instance migrated to a node of the data platform, transforms that input pair into intermediate key/value pair emitted into a provided Collator. Intermediate results are further reduced using a Reducer.

**Listing 4: Mapper interface**

```
public interface Mapper<KIn, VIn, KOut, VOut> extends Serializable {
 /**
 * Invoked once for each input cache entry KIn,VOut .
 */
 void map(KIn key, VIn value, Collector<KOut, VOut> collector);
}
```

Reducer (see Reducer interface in Listing 5), as its name implies, reduces a list of intermediate results from map phase of MapReduceTask. The Distributed Execution Framework's environment creates one instance of Reducer per execution node.

**Listing 5: Reducer interface**

```
public interface Reducer<KOut, VOut> extends Serializable {

/**
* Combines/reduces all intermediate values for a particular
* intermediate key to a single value.
*
*/
VOut reduce(KOut reducedKey, Iterator<VOut> iter);

}
```

Collator (see Collator interface in Listing 6) aggregates results from Reducers executed on the Cloud-TM platform and assembles a final result to return to an invoker of MapReduceTask. Collator is applied to the final Map<KOut,VOut> result of MapReduceTask.

**Listing 6: Collator interface**

```
public interface Collator<KOut, VOut, R> {
/**
* Collates all reduced results and returns R to invoker
* of distributed task.
*
* @return final result of distributed task computation
*/
R collate(Map<KOut, VOut> reducedResults);

}
```

Finally, MapReduceTask is a distributed task uniting Mapper, Reducer and Collator into a cohesive large scale computation to be transparently parallelized across the Cloud-TM Data Platform nodes. Users of MapReduceTask need to provide a cache whose data is used as input for this task. The Distributed Execution Framework's execution environment will instantiate and migrate instances of provided mappers and reducers seamlessly across Infinispan nodes.

## 3.3 Reconfigurable Software Transactional Memory

At the light of the evaluation of the project's enabling technologies reported in deliverable D1.1, the Consortium has reached the agreement on basing the implementation of the in-memory transactional data grid on Infinispan [6], a recent open source project led by JBoss/Red Hat.

The choice has been motivated by the following key considerations.

1. Despite being a quite recent project, Infinispan could benefit from the contributions from a wide and active community which enriched with a wide range of appealing features that would have to be implemented anyway if we opted to develop from scratch this essential component of the Cloud-TM platform. Further, representing the reference clustering solution for JBoss Application Server, which is probably the most popular open-source J2EE AS at current date, Infinispan is widely known in the middleware software industry. These features bring two main advantages: on one side, the Cloud-TM project will be able to rely on a robust and well documented piece of software, which will be enriched to meet the requirements of the Cloud-TM project allowing the partners to focus on integrating innovative research results (such as new replication/distribution algorithms, or self-tuning mechanisms); on the other it will ensure the high visibility of the project itself in the industrial arena, maximizing the chances of successful exploitation of the project's research results.

2. Infinispan exposes a JSR-107 (JCACHE) compatible Cache interface [7] and provides JTA compatibility [8], thus leading to an easy integration in most of the existing architectures in the industry world, which is expected to ultimate facilitate the adoption of the Cloud-TM platform as well.

3. Infinispan is based on a modular and scalable, object-oriented architecture, which entails the possibility to extend the existing modules with minimum intrusiveness. This allows members of the Consortium to focus much more on challenging research problems (such as self-tuning and design of new replica/distribution algorithms) than pure software engineering issues.

In the remainder of this section we will first provide an high level overview of the current architecture of Infinispan (v 5.0.0 CR2), and then focus on the plans concerning how it will be enriched to meet the Cloud-TM's requirements of dynamic reconfiguration.

### 3.3.1 Infinispan's main operational modes

Infinispan architecture is extremely flexible and supports a range of operational modes.

**Standalone (non-clustered) mode**: in this mode, Infinispan acts as a Software Transactional Memory. Unlike classical STMs, however, Infinispan does not ensure serializability (or opacity) [9], but, in order to maximize performance, it guarantees more relaxed consistency models. Specifically, it ensures the ISO/ANSI SQL isolation levels Read-committed and Repeatable-read isolation levels [9], in addition to what

is called Write skew anomaly avoidance. In the following, we briefly overview these isolation criteria, and how they are ensured.

- In the read committed isolation level, whenever the application wants to read an entry from Infinispan for the first time, the application is provided directly with the current value in the underlying data container that only contains committed values. When and if the application writes an entry, the modifications are not immediately applied; conversely, the updated data value is stored in a private transaction's workspace. Any later read by the same transaction returns the value that had previously written.
- In the repeatable read isolation level when the application issues for the first time a read request for an entry, it is provided with the value from the committed values container; this value is stored in a private transaction's workspace, and is returned whenever a new read operation is issued on that same data item.
- If the write-skew check is enabled, Infinispan checks if, between a read and a write operation on a data item X issued from a transaction A, there has been some transaction B that has updated (and committed) X. In this case the transaction A is aborted.

Independently from the isolation level, read operations are always lock-free; on the other side, every write operation requires the acquisition of a lock in order to avoid concurrent modifications on the same entry. Infinispan provides the per-entry lock mode, meaning that each <key,value> pair is protected by its own lock, and the striped lock mode, meaning that each lock is shared by a set of <key,value> pairs. This second option reduces memory consumption for lock management, but has the drawback of incurring in false lock contentions. Lock contention is solved through a simple and configurable timeout scheme; further, Infinispan provides a simple and lightweight deadlock detection mechanism to detect cyclic dependencies between pairs of transactions.

**Full replication mode**: in this mode, Infinispan is deployed over a set of interconnected nodes, each one holding a replica of the same key set. This mode is mainly intended for small scale deployments, given that the need to propagate updates to all the nodes in the system hampers the global scalability of this operational mode. The replication mechanism employed by Infinispan is based on the usage of the classical two phase commit protocol, which ensures that, in order for a transaction to be committed, it must acquire locks successfully across all the nodes in the platform (and thus ensuring replica-wide agreement on the transaction serialization order).

More in detail, upon the issuing of a commit request from a transaction, the prepare phase is initiated: the set of the keys updated by the transaction together with the relevant modification that it wants to perform is broadcast to all nodes. The transaction is then re-run remotely on every node following the aforementioned locking scheme; if a node can successfully acquire all the locks needed, it sends an acknowledgement to the issuer of the prepare phase, otherwise it communicates its impossibility to proceed. If all nodes send a positive reply, then the node of the original transaction sends a final

commit message and all nodes can safely propagate the modifications on the affected keys.

**Partial replication mode**: in this mode every Infinispan replica stores a subset of the data globally managed by the system; every <key, value> pair is replicated (using a Consistent Hashing scheme [5] to distribute keys among nodes, described in the following) over a subset of the total nodes, thus leading to the possibility to meet availability and fault tolerance requirements without the drawbacks that are endemic to the full replicated mode. Upon the commit of a transaction, only the nodes which store keys affected by it are involved in the two-phase commit.

### 3.3.2 High level architecture

In this section we will provide some details about the inner structure of Infinispan, describing the main building blocks that are at its core.

At the highest level of abstraction, the main components are the following: the Commands, which are objects that encapsulate the logic of the methods invoked by the user on the Cache; the Visitors, which are intended to "visit" the commands in order to execute them; special Visitors are the Interceptors, which act as a chain and interact with the subsystems which are affected by a command; the Managers, that are responsible for managing the subsystems.

The main building blocks of the Infinispan architecture are shown in Figure 16, and a brief description for each of them is provided in the following. We start by describing the modules that are common to any operational mode of Infinispan (which we call Core modules). Next we describe, in an incremental fashion, the additional modules that come into play in the full and partial replication modes.

**Core Modules**

1. DataContainer: it embeds a ConcurrentHashMap which ultimately stores the <key,value> pairs handled by users. Infinispan gives to the user the possibility to define a lifespan for the entries, after which they are removed from the data container. Moreover it supports several eviction algorithms (FIFO, LRU, LIRS [10]) in order to avoid extreme usage of the memory heap. Evicted entries are completely removed, coherently with their lifespan, or stored through the registered CacheLoader, if any. Passivation of entries is also supported.

2. CommandFactory: it is responsible for mapping methods invoked by the application into commands which are visited by the Interceptors.

3. EntryFactory: it is responsible for wrapping DataContainer's entries when accessed. This component interacts with TxInterceptor, which is responsible for implementing the concurrency control algorithms that enforce the isolation level specified by the user.

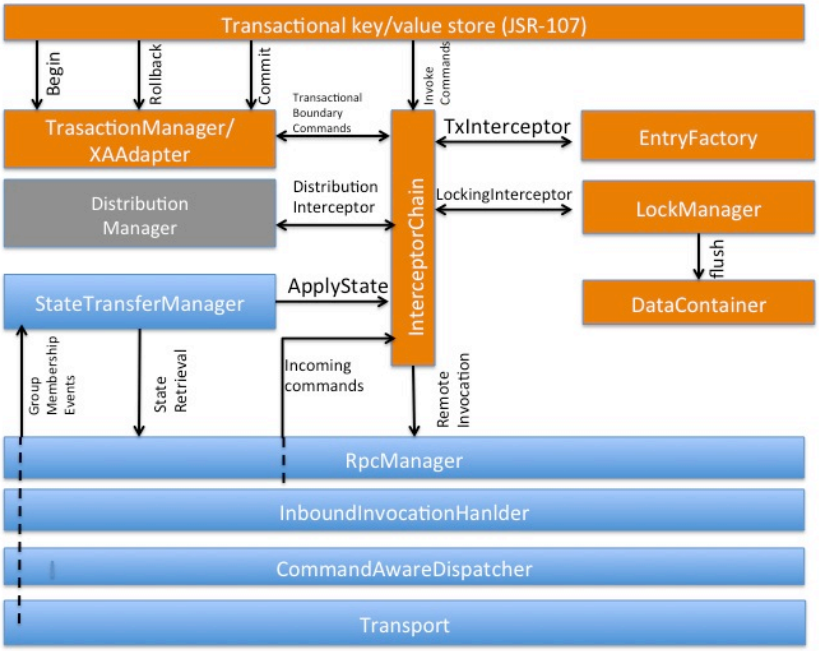



Figure 16: Architectural Overview of Infinispan.

4. LockManager: it deals with all aspects of acquiring and releasing locks for cache entries. It can support both lazy locking mode, meaning that upon a write the lock is taken only locally, postponing the system-wide lock acquisition until the prepare phase, and eager locking, in which a lock is taken immediately on all Infinispan's replica when a request is issued. As already mentioned, lock contentions are solved through timeout and in the case of multiple transactions waiting for the same lock fairness is not guaranteed. The Interceptor responsible for interacting with this component is the LockingInterceptor.

5. Transaction Manager: it allows to define transactions' boundaries. Infinispan is shipped with a simple DummyTransactionManager, but any JTA compliant transaction manager can be used. Infinispan provides also a XAAdapter class which implements the XAResource interface in order to support distributed transactions among heterogeneous resources.

6. InterceptorChain: it is the entry point for any command to be performed. It triggers the visit of the command from all the registered interceptors.

7. ComponentRegistry: it acts as a lean dependency injection framework, allowing components and managers to reference and initialize one another.

**Additional modules involved in full replication mode**

1. Transport: it provides a communication link with remote caches and allows remote caches to invoke commands on a local cache instance. This layer ultimately relies on JGroups, to implement lower level's Group Communication Service's abstractions (such as view synchrony [11], failure detection, reliable broadcast, etc).

2. CommandAwareDispatcher: it's a JGroups RPC dispatcher which knows how to deal with replicated commands.

3. InboundInvocationManager: it is a globally-scoped component which is able to reach named caches and trigger the execution of commands from remote ones by setting up the interceptor chain for them.

4. RpcManager: provides a mechanism for communicating with other caches in the cluster, by formatting and passing requests down to the registered Transport. This component dialogues directly with the ReplicationInterceptor, which triggers the re-execution of transactions via remote procedures calls.

5. StateTransferManager: it handles generation and application of state on the cache.

**Additional modules involved in partial replication mode**

1. DistributionManager: it is responsible both for handling the correspondence between any data entry and its owner Infinispan node and the caching policy of entries which are retrieved from remote instances when locally accessed. It communicates with the DistributionInterceptor which is mutually exclusive with respect to the Replication one.

   To determine which instances store which entries, the DistributionManager exploits a Consistent Hashing algorithm [5]: both keys and caches are hashed onto a logical circle and each cache contains all keys between itself and the next clockwise one on the circle. This mechanism allows to detect the owner of an entry without additional communication steps and is extremely efficient since upon the leave/join of a node, only a reduced set of entries has to be redistributed, involving in this operation only the nodes which are logically adjacent to the one that has just left/joined the system.

   This component is also responsible for managing a level one cache aimed at temporary storing entries which are not a burden, coherently with the consistent hashing, of a specific instance. If a transaction requests an entry that is not local to the node, a remote procedure call is issued aimed at retrieving and storing it in the level one cache. DistributionManager is responsible for defining eviction policies for those entries.

   Moreover, the DistributionManager module is in charge of removing from the DataContainer entries which a node is no longer responsible for (e.g. upon the join of a new instance) or of moving them to the level one cache.

### 3.3.3 Extensions to support complex dynamic reconfigurations

In order to deal with runtime reconfigurations, Cloud-TM enriches Infinispan's architecture with a Reconfiguration Manager: it is designed to communicate with the Autonomic Manager via JMX and, when a reconfiguration request is triggered, it orchestrates the switch operations of the various components via the native Infinispan's visiting mechanism. In particular, a ReconfigurationCommand is defined, which contains a set of ConfigInfo objects, one per component, that contain information about the operations to be performed on every subsystem.

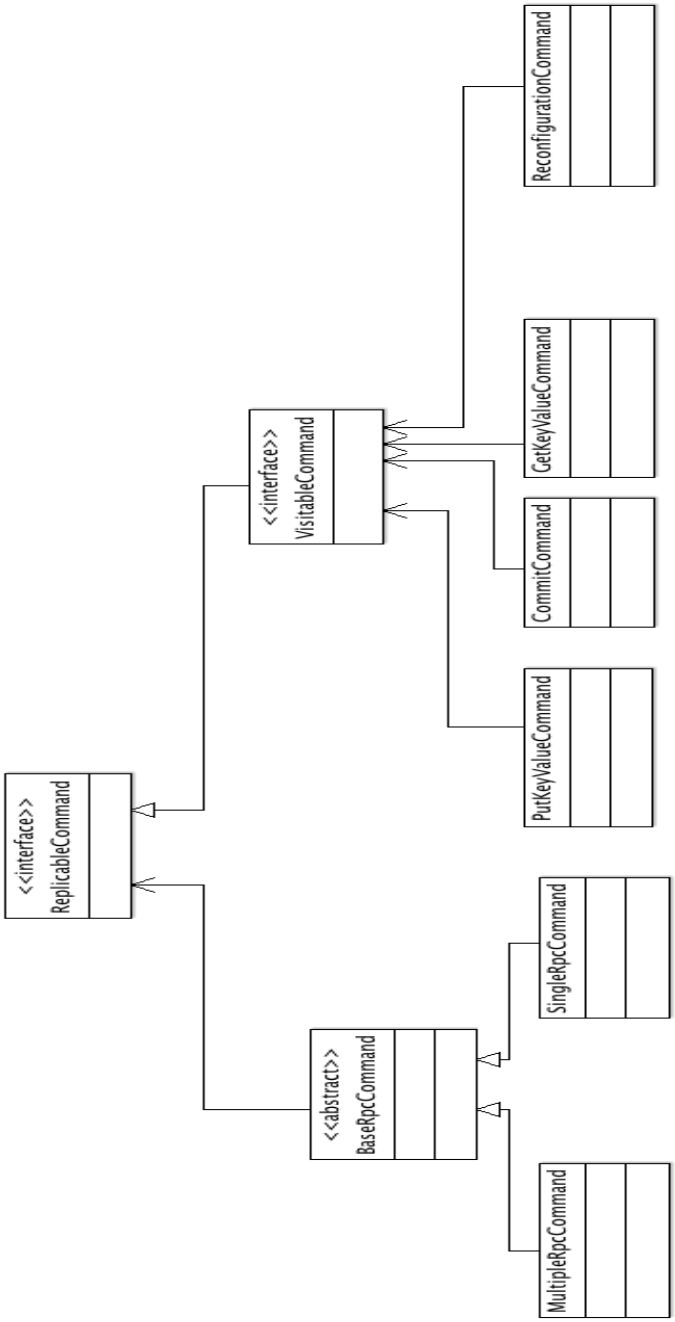



Figure 17: Infinispan's Expanded Command Hierarchy.

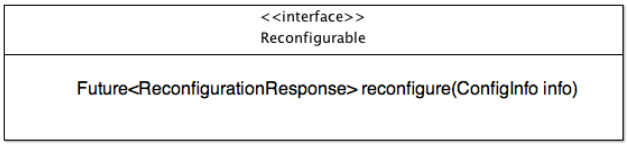



Figure 18: Reconfigurable Interface.

A subsystem that wants its internals to be dynamically tuned, must therefore implement a Reconfigurable interface (see Figure 18) exposing the method Reconfigure(ConfigInfo info), so that this can be properly invoked by the visiting interceptor.

The return type of that method is a ReconfigurationResponse object, which contains every information needed by the ReconfigurationManager to determine the outcome of the switch operations.

By exploiting the polymorphism on the ConfigInfo class, this architecture allows to reconfigure several subsystems with the use of a single command. Besides this, by exploiting the capability of the java.util.concurrent.Future interface, we allow the invocation to the methods which encapsulate the reconfiguration logics to return asynchronously. This allows reconfigurations of modules that are not subject to specific causality constraints to be executed in parallel, reducing the total reconfiguration latency.

The final commit of the switch from a configuration to another one is issued by the Reconfiguration Manager, which can also act as a system-wide synchronization element to guarantee that, if the configuration switch has a global scope, then all Infinispan replicas have reached agreement on it before declaring the reconfiguration phase concluded.

More in detail, upon the request for a reconfiguration, every subsystem involved in the reconfiguration has to perform modifications on its internal state and on its relevant fields on the Infinispan's Configuration object, which stores all state information concerning the configuration of a given instance of Infinispan.

At its core, a ConfigInfo object is a set of String and Objects: the Strings are the names of the methods exposed by the Configuration and are invoked via Reflection; the Objects can represent the parameters for such methods or any other element needed by a subsystem to perform the reconfiguration. The semantic of an Object as well as its specific class are determined at runtime basing both on the method currently invoked via Reflection and the specific logic implemented by the method Reconfigure.

```
Visitor
visitClearCommand(InvocationContext, ClearCommand)                          Object
visitCommitCommand(TxInvocationContext, CommitCommand)                      Object
visitEntrySetCommand(InvocationContext, EntrySetCommand)                    Object
visitEvictCommand(InvocationContext, EvictCommand)                          Object
visitGetKeyValueCommand(InvocationContext, GetKeyValueCommand)              Object
visitInvalidateCommand(InvocationContext, InvalidateCommand)                Object
visitKeySetCommand(InvocationContext, KeySetCommand)                        Object
visitLockControlCommand(TxInvocationContext, LockControlCommand)            Object
visitPrepareCommand(TxInvocationContext, PrepareCommand)                    Object
visitPutKeyValueCommand(InvocationContext, PutKeyValueCommand)              Object
visitPutMapCommand(InvocationContext, PutMapCommand)                        Object
visitRemoveCommand(InvocationContext, RemoveCommand)                        Object
visitReplaceCommand(InvocationContext, ReplaceCommand)                      Object
visitRollbackCommand(TxInvocationContext, RollbackCommand)                  Object
visitSizeCommand(InvocationContext, SizeCommand)                            Object
visitReconfigurationCommand(ConfigInfo)                                     Object
```

Figure 19: Extended Visitor Interface.

The above described architecture, thanks to the Reconfiguration Manager, encompasses also the possibility to orchestrate complex reconfiguration operations which would be difficult to split into several, component-specific minor reconfigurations. For instance, this design pattern allows to modify at runtime the interceptor chain. The latter can therefore be seen as both the means through which ReconfigurationCommand are spread in the system, and as a reconfigurable object itself. A practical example for this scenario is the switch from Replicated to Distributed mode, which entails the substitution of the ReplicationInterceptor with the DistributionInterceptor.

## 3.4 Reconfigurable storage system

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.

As mentioned while illustrating the high level architecture of the Cloud-TM platform in Section 2, the component in charge of ensuring data durability is the "Reconfigurable Storage System" layer. The key requirements identified for this component (R20-R22 of the deliverable D1.1 "User Requirements Report") consisted in achieving maximal portability towards storage systems of different nature, ranging from local/distributed file systems to Cloud-oriented storage services (e.g. Amazon's S3, Google's Big Table, Hadoop's Hbase, Cassandra).

This requirement is perfectly met by the software architecture currently employed by Infinispan's, so called, CacheLoader/CacheStore modules. Infinispan's approach to dealing with external persistent storages, in fact, is based on 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. Indeed, the high flexibility of Infinispan's architecture for interfacing with persistent storage, as well as the availability of plug-ins for a plethora of alternative persistent storage technologies, have been some of the main reasons that have led to choose Infinispan as the backbone of the infrastructure of Cloud-TM.

In fact, this choice will allow the consortium to exploit the existing Infinispan's codebase and focus on the research issues associated with the dynamic reconfiguration and self-tuning of the persistent storage layers. These will include the design of autonomic mechanisms aimed both at tuning the individual parameters of connected data stores, as well as the run-time switch/add/remove of persistent storages on the basis of the current workload and of QoS/cost constraints.

The architecture of the Reconfigurable Storage System is illustrated by the diagram in Figure 20. It will consist of two main logical blocks: a set of plugins allowing interfacing Infinispan with heterogeneous data stores, and a Persistent Store Manager, which will expose a Generic Reconfigurable Component Interface (see Section 3.3) to allow monitoring and tuning by the Autonomic Manager.

These two modules are described in the following.

### 3.4.1 Interfaces towards Persistent Storages

As already discussed, in order to achieve interoperability with heterogeneous persistent stores, the Cloud-TM platform will exploit the modular plug-in based adopted by Infinispan.

The key abstraction used in Infinispan to support interactions with external data stores are the CacheLoader and CacheStore interfaces, which expose, respectively, a set of methods aimed at retrieving and storing data from the persistent store, both in transactional and non transactional context. As shown in the class diagram in Figure

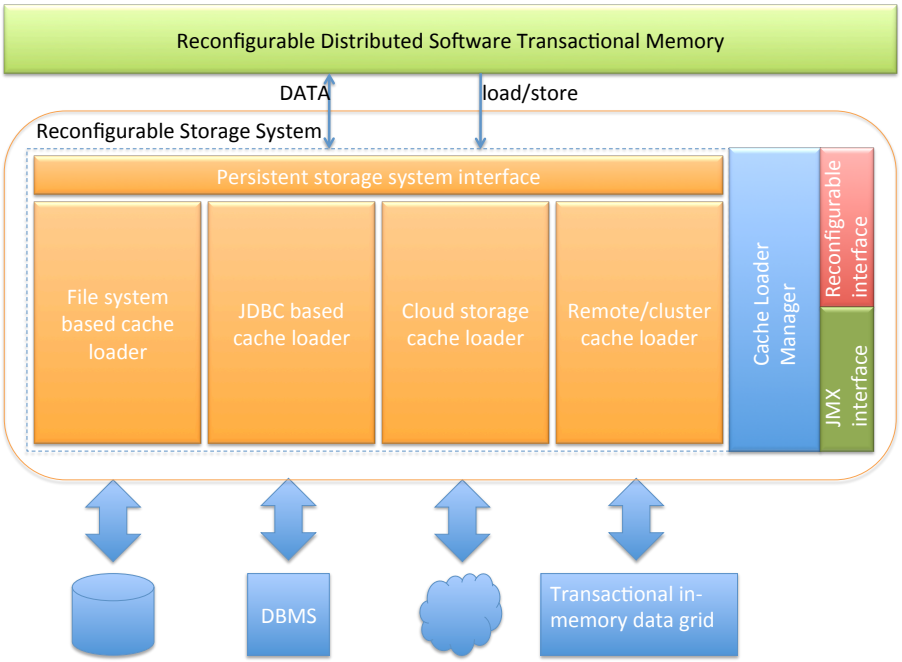



Figure 20: Architecture of the Reconfigurable Storage System.

21 (which is using UML notation), these interfaces are sufficiently generic to allow encapsulating the specific implementation details associated with external data stores of different nature, including (local/distributed) file-systems, relational DBMSs and Cloud-based storage systems.

Referring to Figure 20, the Persistent Store Manager is responsible for handling the relations between a specific Infinispan's instance and the underlying CacheLoaders/CacheStores, supporting, whether needed, the simultaneous interfacing towards multiple data stores. The interaction between these modules and Infinispan's core is achieved via the following interceptors, which are injected in the interceptors chain at the start-up or during the reconfiguration phase (see Section 3.3.3):

1. CacheLoaderInterceptor: this interceptor is responsible for fetching data from the connected data stores. Whenever a read/update operation is performed on the data grid and the requested entries are not found in cache, every active cache loader configured for that cache tries to fetch data from the corresponding connected storage until a valid not-null element of data is found.

2. CacheStoreInterceptor: this interceptor is responsible for writing modifications back to the store; they are stored back through the CacheStore, either after each operation call, if it is not performed in transactional context, or upon the commit of the relevant transaction.

Cache loaders can be grouped in four distinct categories, depending on the store type they handle.

1. File system based cache loaders: they are aimed at providing an abstraction layer to connect to regular local or distributed filesystems (e.g. Cassandra).

2. JDBC based cache loaders: their purpose is to provide connectivity towards JDBC-accessible databases.

3. Cloud storage cache loaders: they are intended to allow the exploitation of data storage facilities over the cloud (e.g. Amazon's S3).

4. Remote cache loaders: they provide the possibility to exploit another cache as a lower layer used to load data.

#### Eviction and passivation

When configured to interact with an external persistent data store, the Reconfigurable Distributed Software Transactional Memory can be seen as a volatile cache having a smaller capacity with respect to the persistent storage. In this case the issue of which entries should be kept in memory, and which stored on the persistent data store, naturally arises.

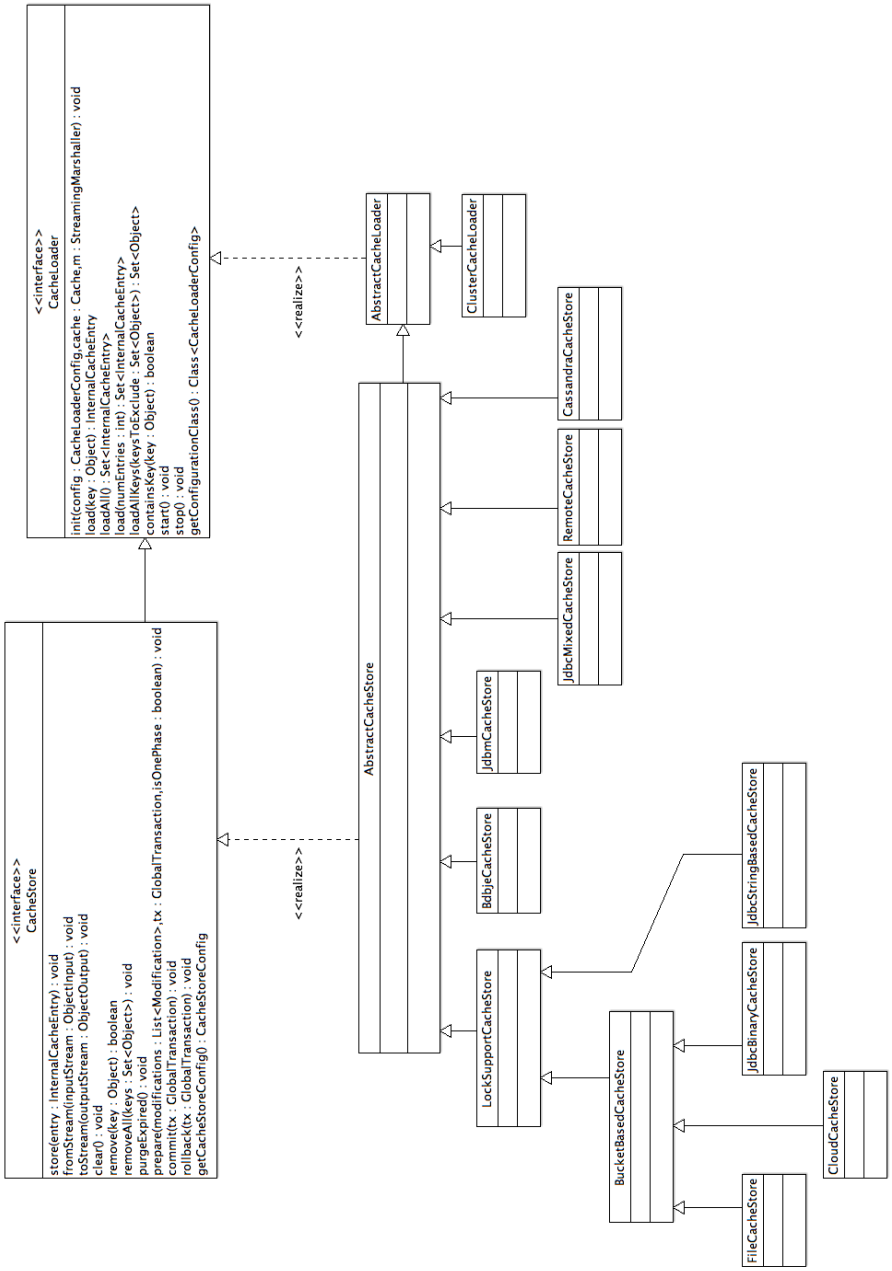



Figure 21: Key Interfaces Used to Achieve Interoperability between a Number of Heterogeneous Persistent Data Stores.

Infinispan already provides advanced supports for this kind of use cases, implementing a set of alternative policies for what concerns cache eviction, write-back/write-through, synchronous/ asynchronous propagation of updates.

Cache loaders/stores can also be used to enforce entry passivation and activation. Cache passivation is the process of removing an object from in-memory cache and writing it to a secondary data store on eviction; cache activation is the dual process.

Even though there is no direct relationship between eviction and cache loading, when passivation is enabled, such a relationship arises: writes to the persistent store via the cache loader only occur as a part of the eviction process. Data is deleted from the persistent store when the application reads it back into memory: in this case, what's in memory and what's in the persistent store are two subsets of the total information set, with no intersection between the subsets.

### 3.4.2 Run-time reconfiguration support

In order to meet Cloud-TM reconfiguration requirements, some of the described components are enhanced in such a way to maintain compatibility with the described architecture while exposing new methods in order to be reconfigurable in compliance with the visiting mechanism on the ReconfigurationCommand triggered by the Reconfiguration Manager that will be integrated into Infinispan (see Section 3.3.3).

In particular the Persistent Store Manager will be requested to implement the Reconfigurable interface, and the CacheLoaderInterceptor and CacheStoreInterceptors will be extended to implement the Extended Visitor Interface, both defined in Section 3.3.3.

The latter will allow Infinispan's Reconfiguration Manager to coordinate the dynamic reconfiguration of the mechanisms employed to interact with external data stores using the same, general approach described in Section 3.3.3 to deal with reconfigurations of its other internal components. Specifically, the Reconfiguration Manager will be able to provide reconfiguration instructions, encoded as a ConfigInfo object within the Reconfiguration Command, to the CacheLoader and CacheStore interceptors. These, in their turn, will trigger reconfigurations of the persistent storage policies by invoking the reconfigure() method exposed by the Persistent Storage Manager via its Reconfigurable interface.

For what concerns the monitoring and tuning of this layer by the Autonomic Manager, given that the Persistent Storage Manager is implemented in Java, it will expose standard JMX interfaces, and rely on the XML-based advertisement mechanism described in Section 3.1 in order to expose meta-data concerning which of its parameters should be monitored/tuned and on the basis of what utility functions.

# 4 Autonomic Manager

In this Section we discuss architectural aspects concerning the design of the Autonomic Manager, namely the component in charge of monitoring and piloting the automatic tuning and scaling of the Data Platform of Cloud-TM.

We start by describing, in Section 4.1, the mechanisms that will be employed to allow applications to express their QoS/cost constrains to the Cloud-TM platform.

In Section 4.2, we discuss the architectural choices underlying the development of the Workload and Performance Monitor. Note that this module has been already developed, and the corresponding prototype has been made publicly available as the deliverable D3.1.

Section 4.3 describes the architecture that has been planned for the Workload Analyzer. This component is in charge of processing the raw data streams generated by the Workload and Performance Monitor, providing differentiated views based on differentiated aggregation rules in order to generate workload characterizations (and forecasts) for individual platform components/layers, as well as to generate alert signals in case of violations of the specified QoS/cost constraints.

Finally, in Section 4.4, we present the envisioned architecture of the Adaptation Manager, namely the component that will host the optimization logic employed to guide the self-tuning of the various components of the Data Platform, as well the automatic provisioning of resources from the IaaS provider(s).

## 4.1 QoS API

The QoS API will provide a means for supporting a structured interaction between the Autonomic Manager and customer applications. The objective of the interaction is the specification of a agreement expressed in terms of:

- a set of parameters, and the associated values, globally defining the customer expectation, in terms of QoS to be matched by the Cloud-TM platform at run-time, and
- the customer agreed upon obligations/costs for obtaining the level of service as specified via the QoS selected parameters/values.

Clearly, such an API must be flexible, in order to support interactions aimed at defining differentiated QoS/cost bindings. Anyway, constraints must be considered in order to prevent scenarios where meaningless agreements could not eventually be committed by the involved parties. In our architectural approach, we intend to define the QoS API as a parametrizable XML-based template, whose structure will define a meta-model according to which any meaningful agreement can be actually instantiated.

We recall that one of the main objectives of Cloud-TM is the exemplification of the applications' development process, which must be also reflected in a simplified negotiation supported via the selected QoS API. This should be reflected in that the template should be able to express the obligations of the customer (hence requirements to be matched by customer applications) in a very synthetic and simplified manner. One immediate advantage would be in avoiding the need for application developers to carry out very extensive and complex profiling activities in order to be able to determine whether those obligations can be actually met when running on top of the Cloud-TM platform.

On the basis of the above reasoning, we can roughly identify the below listed three differentiated operating modes characterizing any application potentially running on top of the Cloud-TM platform:

1. the application does not currently accesses the data layer;
2. the application is currently accessing the data layer via an individual data access operation (either read or write);
3. the application is currently accessing the data layer within a transactional context possibly entailing multiple data access operation (either read or write).

Any KPI related to point 1 is independent of the data layer performance. Instead it would essentially express performance aspects in relation to the computational power and/or the deploy characterizing the infrastructure hosting the application. On the other hand, KPIs associated with the operating modes 2 and 3 would express performance dependencies on the data layer behaviour.

Performance guarantees when running in mode 1 would therefore be essentially related to the scale up/down of the computational power associated with the underlying infrastructure. On the other hand, performance guarantees when running in modes 2

and 3 would require providing statistical bounds on the timing of operations at the level of the data layer. Point 1) will be reflected in our template by expressing guarantees on per-thread computational power and bandwidth, once fixed the maximum number of concurrent threads. On the other hand, for modes in points 2 and 3, the template would express guarantees on:

- the average latency for the access in read or write mode to a single object hosted by the data-platform (i.e. out of a transaction);
- the x percentile of the response time for the access in read or write mode to a object hosted by the data-platform (i.e. out of a transaction);
- the average latency for the execution of a transaction;
- the x percentile of the response time for the execution of a transaction.

The above guarantees should be coupled with obligations to be respected by the application code expressed in terms of (A) the maximum CPU time requested while running within a transactional context (this is needed for point 3), and (B) the maximum level of conflict exhibited by the application, that we easily express via our template as the likelihood that the read/write sets associated with two concurrent transactions (or transactional accesses to single data) have null intersection.

We note that, compared to classical scenarios where QoS parameters are expressed by essentially relying on the volume of requests by the customer, here we have introduced a bit of additional information, related to the customer application profile. This is related to that the Cloud-TM platform does not offer and end-delivered service, but a data access "middleware service" on top of which the customer application runs. Hence, violations of the performance guarantees can be caused by factors that are out of the control of the Cloud-TM Autonomic Manager, such as application run-time that does not conform with an expected profile, which may have strong negative effects just due to the particular, transactional nature of the services offered by the Cloud-TM middleware.

As for the technological support to the template definition, and so to the instantiation of the QoS API, our choice to rely on XML is motivated by the fact that it has been extensively adopted in the context of Web Service Agreement Specification [12].

## 4.2 Workload and Performance Monitor

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

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

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

As pointed out within the deliverable D1.2 "Enabling Technologies Report", one reference framework for the building of monitoring systems in the context of Cloud based platforms is Lattice [2], which has also been used as an architectural building-block within the RESERVOIR project [13]. At a certain level of abstraction, Lattice can be seen as a technology providing the skeleton of a distributed, scalable data-gathering system, hence being suited for instantiating a distributed, scalable monitoring platform.

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

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

The data plane is used to transfer data-messages, whose payload is a set of measures, each kept within a proper message-field. The structure of the message (in terms of amount of fields, and meaning of each filed) is predetermined. Hence fields do not need to be explicitly tagged so that only data-values are really transmitted, together with a concise header tagging the message with very basic information, mostly related to source identification and its timestamp. Such a structure can be anyway dynamically reconfigured via interactions supported by the info plane.

This is a very relevant feature of Lattice since it allows minimal message footprint for (frequently) exchanged data-messages, while still enabling maximal flexibility, in terms of on-the-fly (infrequent) reconfiguration of the monitoring-information structure exchanged across the distributed components within the monitoring architecture.

Finally, the control plane can be used for triggering reconfiguration of the producer component, e.g., by inducing a change of the rate at which measurements need to be taken.

Notably, the actual transport mechanism supporting the planes is decoupled from the internal architecture of producer/consumer components. Specifically, data are disseminated across these components through configurable distribution mechanisms ranging from IP multicast, to publish/subscribe systems, which can be selected on the basis of the actual deploy and which can even be changed over time without affecting other components, in term of their internal configuration. The framework is designed to support multiple producers and multiple consumers, providing the chance to dynamically manage data source configuration, probes activation and deactivation, data sending rate, redundancy and so on. Overall, such a model well fits highly dynamic distributed environments.

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

By relying on Java, portability issues are mostly limited to the implementation of the ad-hoc components. As an example, a probe-thread based on direct access to the "proc" file system for gathering CPU/memory usage information is portable only across (virtualized) operating systems supporting that type of file system (e.g. LINUX). However, widening portability across general platforms would only entail reprogramming the internal logic of this probe, which in some cases can even be done by exploiting, e.g., pre-existing Java packages providing platform-transparent access to physical resource usage. As for this aspect, we plan to incorporate within the WPM architecture cross-platform libraries such as SIGAR, JConsole, JnetPCap, and JPcap, which offer methods to extract relevant data from the devices of interest, across different operating systems and architectures. Among these four libraries, the most complete is surely SIGAR which has the capability to access information exposed by the operating system (e.g. CPU utilization, memory consumption, network activities) in-

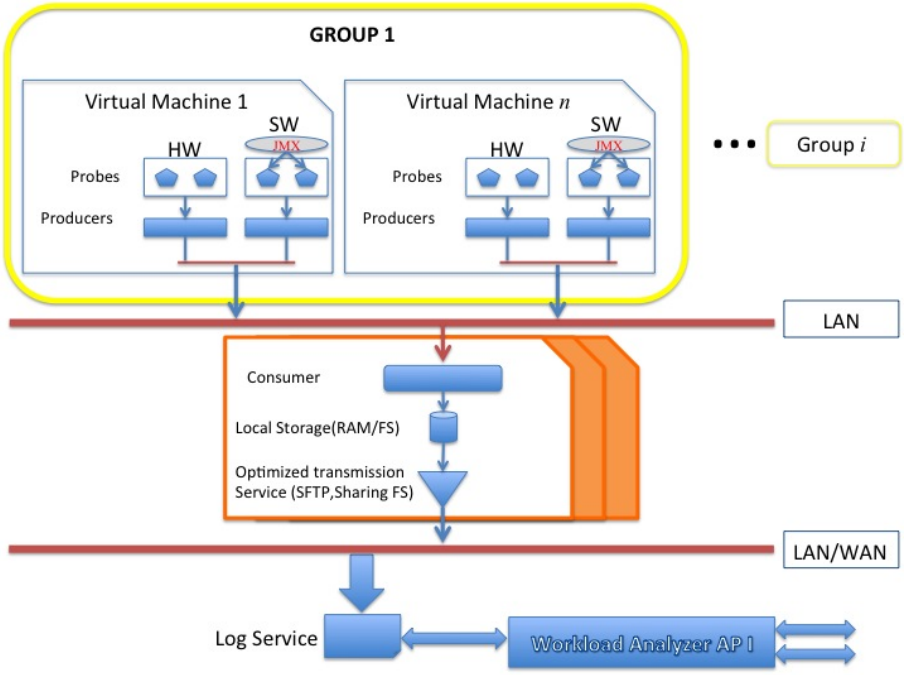



Figure 22: Architectural Scheme for the Workload and Performance Monitor.

dependently of the underlying actual platform. In our architectural plan, it will be therefore used for instantiating wrappers fitting with differentiated platforms, such as Linux, Windows and Mac OS X.

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

Another relevant aspect for Lattice is that the above mentioned activities (i.e. threads) can be run within dedicated demon processes, or within proper processes acting inside (i.e. as a part of) the monitored platform. In the latter case, the probes even have access to the state of the objects belonging to the monitored process, either via information shared across threads, or by exploiting internal audit-API provided by the running software. This facilitates the integration of the monitoring system in cases where the object activities are of interest to the monitoring process, which we expect to be the case for the Cloud-TM platform. As for the latter aspect, User Requirements in the range R.41-R.43 just deal with collection of statistics related to the data-layer (including inner distribution/replication protocols). For Java based components, Lattice probes can be run as demons interacting with the components' native audit system (or extensions of this system tailored to Cloud-TM), via the JMX framework, which will result in a quite natural integration process.

Figure 22 shows the general architecture of the WPM belonging to Cloud-TM. It has been defined according to the need for supporting the following two main function-

alities:

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

The SDG functionality architecturally maps onto an instantiation of the Lattice framework, with Cloud-TM specific probes and collectors.

The computational nodes of the network (e.g. Virtual Machine(VM)s) could be logically grouped. Each group will entail per-machine probes targeting two types of resources: hardware/virtualized and logical. Statistics for the first kind of resources are directly collected over the Operating System (or by standard OS decoupled libraries) as hinted before, while statistics related to logical resources (e.g. data-platform) are collected directly at application level, e.g. by using the JMX Framework for Java components. The data collected by the probes are sent to the Lattice Producer component via facilities directly offered by the framework. Each Producer is coupled with one or many probes and it is responsible of managing them.

The Consumer is the Lattice component that receives the data from the Producers, via differentiated messaging implementations, which could be selected on the basis of the specific system deploy. We envisage a LAN based clustering scheme such that the Consumer is in charge of handling one or multiple groups of machines belonging to the same LAN. Anyway, the number of Consumers in not meant to be fixed, instead it can be scaled up/down depending on the amount of instantiated probes/Producers. Overall, the Consumer could be a centralized or a distributed process, thus supporting specific scalability/availability levels.

Beyond collecting data from the Producers, the Consumer is also in charge of performing a local elaboration aimed at producing a suited stream representation to be provided as the input to the log layer, supporting the SDL functionality. This will result in providing a flexible (in terms of structure) and concise representation of monitored parameters (e.g. provide the histogram of CPU utilization instead of sending individual samples).

We plan to use a storage locally available to the Consumer, which can either be volatile, or stable (e.g. the file system, or a local DBMS) . Periodically the samples are processed and, depending on the nature of the analyzed parameters, their suited, concise representation is built within the stream destined for SDL. The functional block which is responsible of the interaction between SDG and SDL is the so called optimized transmission service. In our plan, it could rely on top of differentiated solutions also depending on whether the instance of SDL is co-located with the Consumer or resided on a remote network. Generally speaking we could employ, e.g., SFTP or File System sharing. Also, stream compression schemes can be actuated to optimize both latency and storage occupancy.

The Log Service is the logical component responsible of storing and managing all the gathered data. Log Service can support queries from the Workload Analyzer so to expose the statistical data for subsequent processing/analysis. The Log Service could be implemented in several manners, in terms of both the underlying data storage technology and the selected deploy (centralized vs distributed).

As for the first aspect, different solutions could be envisaged in order to optimize access operations depending on, e.g. suited tradeoffs between performance and access flexibility. This is also related with the data model ultimately supported by the Log Service components, which might be a traditional relation model or, alternatively, a <key,value> model. Further, the Log Service could maintain the data onto a stable storage support or in a volatile memory, for performance vs reliability tradeoffs.

The second aspect mentioned above is strictly coupled with the functionality/architecture of the Workload Analyzer. Specifically, the Log Service could be geographically distributed so to better fit the deploy of WPM (hence taking advantage from data partitioning and distributed processing).

## 4.3 Workload Analyzer

The Workload Analyzer (WA) within the Cloud-TM platform is in charge of performing the following tasks:

- building differentiated views of the raw statistical data provided by the WPM component, based on differentiated aggregation rules;
- generating workload characterizations for individual platform components/layers;
- providing mechanisms to predict future workload profiles and resource demands;
- providing a mechanism to assess the actual value of QoS related parameters and to notify the occurrence of potentially adverse gradients.

The Workload Analyzer is logically located in between the WPM and the Adaptation Manager. A functional architecture draft is proposed in Figure 23.

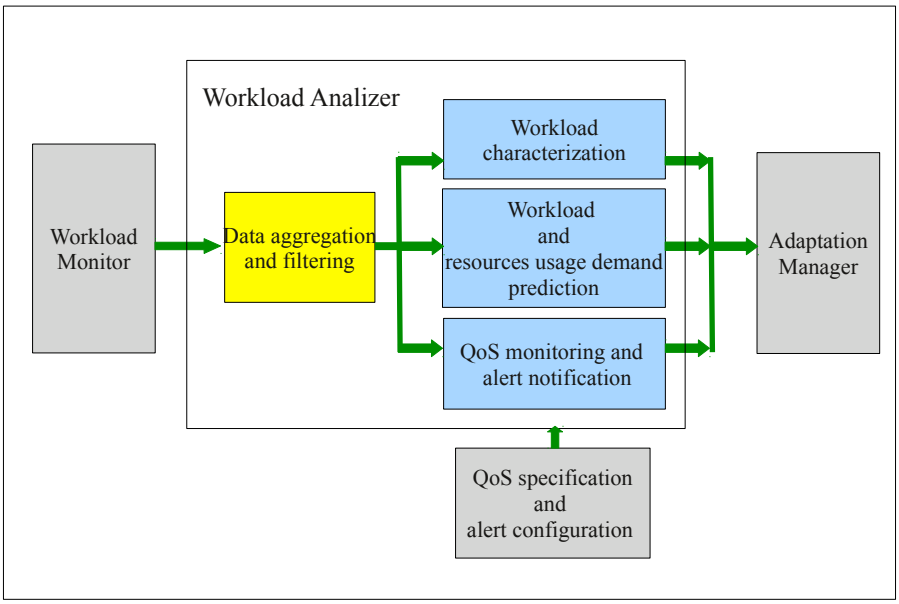



Figure 23: Architectural Scheme for the Workload Analyzer.

Raw data collected by the WPM act as input to the data aggregation and filtering component. This component provides aggregated views of the raw data, built by using differentiated aggregation rules. Within this process, raw data are managed by means of a data structure based on the OLAP cube model [14], entailing flexible and fast analysis facilities, in terms of aggregation/filtering options and operations.

In the OLAP cube model, measurements correspond to the monitored attributes of the system components (see Section 3.1). Also, tailoring the OLAP cube model to Cloud-TM leads to the identification of two different dimensions for a monitored attribute, depending on whether the attribute is associated with (A) an infrastructural component or (B) a data platform component. A third dimension can be further considered, which expresses how the monitorable attribute value changes over time. Overall, we have three different hierarchies of dimensions:

- infrastructural hierarchy;
- data platform hierarchy;
- time hierarchy.

These are schematized in Figure 24.

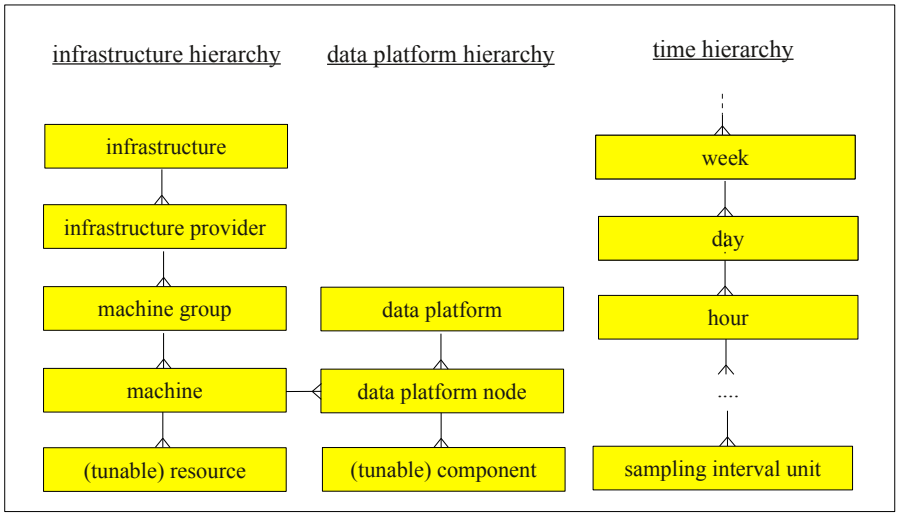



Figure 24: Schematization of the Hierarchies.

A monitorable attribute associated with an infrastructural resource has a dimension related to the infrastructure and a dimension related to time. Considering the case of the CPU resource, this means that is it possible to build a snapshot of CPU usage at different aggregation levels of the hierarchy, e.g. by focusing on the average CPU usage for each single machine or the level of a group of machines.

A monitorable attribute associated with a data platform component, such as the transaction response time, has a dimension related to the data platform and a dimension related to time. Hence, it is possible to build a snapshot expressing, e.g., the x-percentile of the transaction response time at the level of a single node of the data platform, or across the entire data platform.

Anyway, the model is highly flexible, thus allowing its extension to accommodate for additional dimensions, if these are required in relation to the specific monitorable attributes of interest. For instance, if one would need the x-percentile of the transaction response time for each transactional class, we could add a transactional class dimension, which would allows to obtain per-class separate statistics.

As for the time dimension, it can be used to support the identification of statistical snapshots related to time windows with different granularity levels, ranging from minutes, to days or weeks and so on.

By means of the aforementioned data model, the aggregation and filtering component within the Workload Analyzer can execute queries in relation to a monitorable attribute by:

- aggregating raw data at different granularity level, along different dimensions;
- filtering data according the selected dimensions and granularity levels;
- producing statistical measurements by means of differentiated operators (e.g. average, minimum, maximum, variance, x percentile)

The workload characterization component uses the above snapshots to actuate characterizations for the whole Cloud-TM platform workload, such as the workload sustained by the transaction manager, the one sustained by the replica consistency manager and by the group communication manager.

The Workload Analyzer also entails a functional block devoted to actuating mechanisms for predicting workload profiles and resource demands. Snapshots are analyzed by means of time-series exploration techniques with the aim at discovering trends, cyclic behaviour, seasonality, etc.

Finally, the Workload Analyzer embeds a QoS monitoring block, which is in charge to raise alert notifications to the Adaptation Manager whenever the (aggregated) snapshots related to statistics involving parameters that belong to QoS specification indicate a (potential) degradation of expected QoS levels. The alert notification mechanism should allow to configure alarm signals to be notified to other components when specific conditions are satisfied.

Beyond the definition of the functional scheme, another relevant aspect is related to the deploy of the Workload Analyzer components, given that they are required to interact with external components, such as the WPM and the Adaptation Manager. As for the interaction with the WPM, this could be based on the access to a distributed statistical-data log service, exposed by WPM. To optimize such an interaction the Workload Analyzed can be architecturally organized in a way to minimize network traffic related to the retrieve of raw data collected and exposed by WPM via the log-service access. Specifically, since the size of aggregated snapshots is usually smaller than the size of the raw data set forming the base for the snapshot construction the aggregation and filtering component of the Workload Analyzer can be decomposed into sub-components, each of which can be deployed close to an individual instance of the log service, so that snapshots are built in a distributed fashion, and then forwarded by the sub-components toward a collector. This solution is depicted in Figure 25.

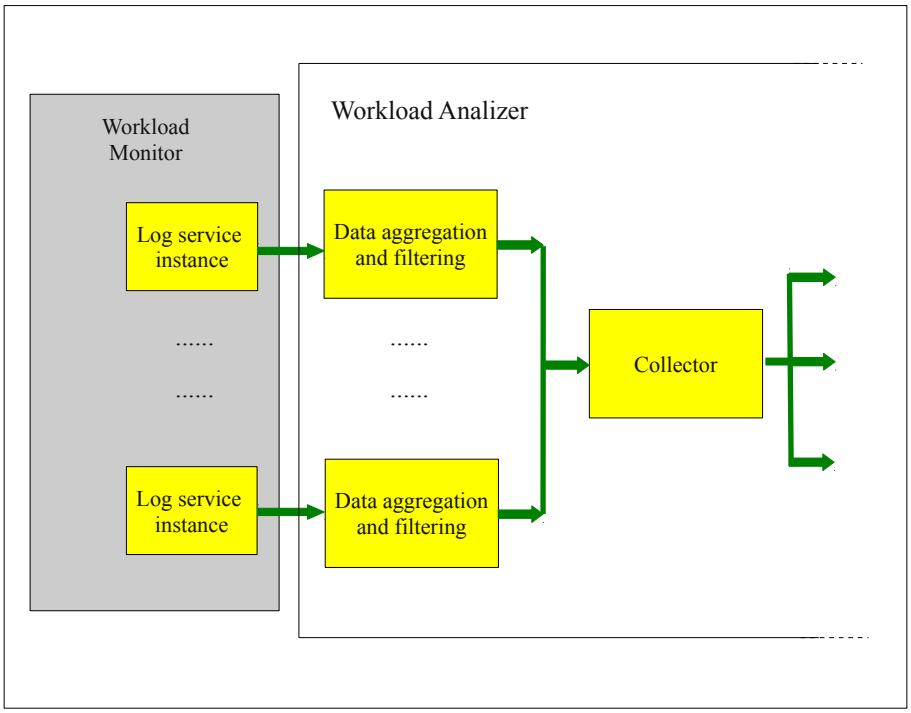



Figure 25: Distributed Data Aggregation and Filtering.

An alternative approach would be to fully distribute the Workload Analyzer, e.g., by replicating all its components. In this solution the different data aggregation and filtering components interact in order to share raw data stored within different log service instances. In this way each replica can access to the whole set of raw data distributed over the log service. This solution is schematized in Figure 26.

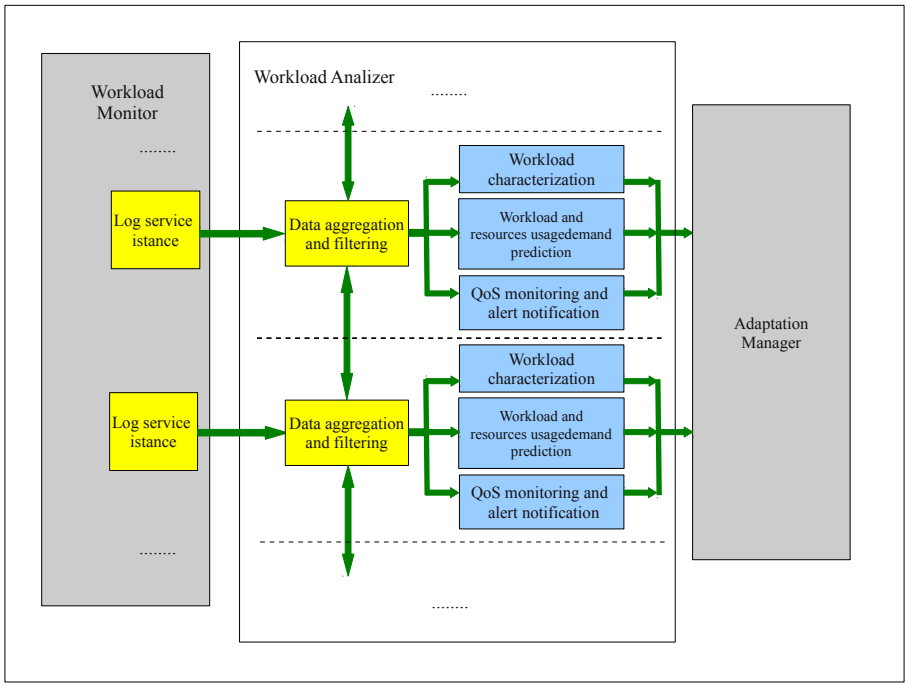



Figure 26: Fully Distributed Architecture for the Workload Analyzer.

## 4.4 Adaptation Manager

The diagram in Figure 27 provides an architectural overview of the Adaptation Manager. This module is in charge of defining the global self-optimization strategy of the Cloud-TM platform and of orchestrating the platform's reconfiguration process by coordinating the tuning of the ecosystem of components forming the Data Platform, and the provisioning resources from IaaS Cloud providers.

The optimization of the Cloud-TM platform is managed by the "Optimization Manager" component, which takes as input the following data flows:

- the QoS/Cost constraints specified by the users via the QoS API (see Section 4.1);

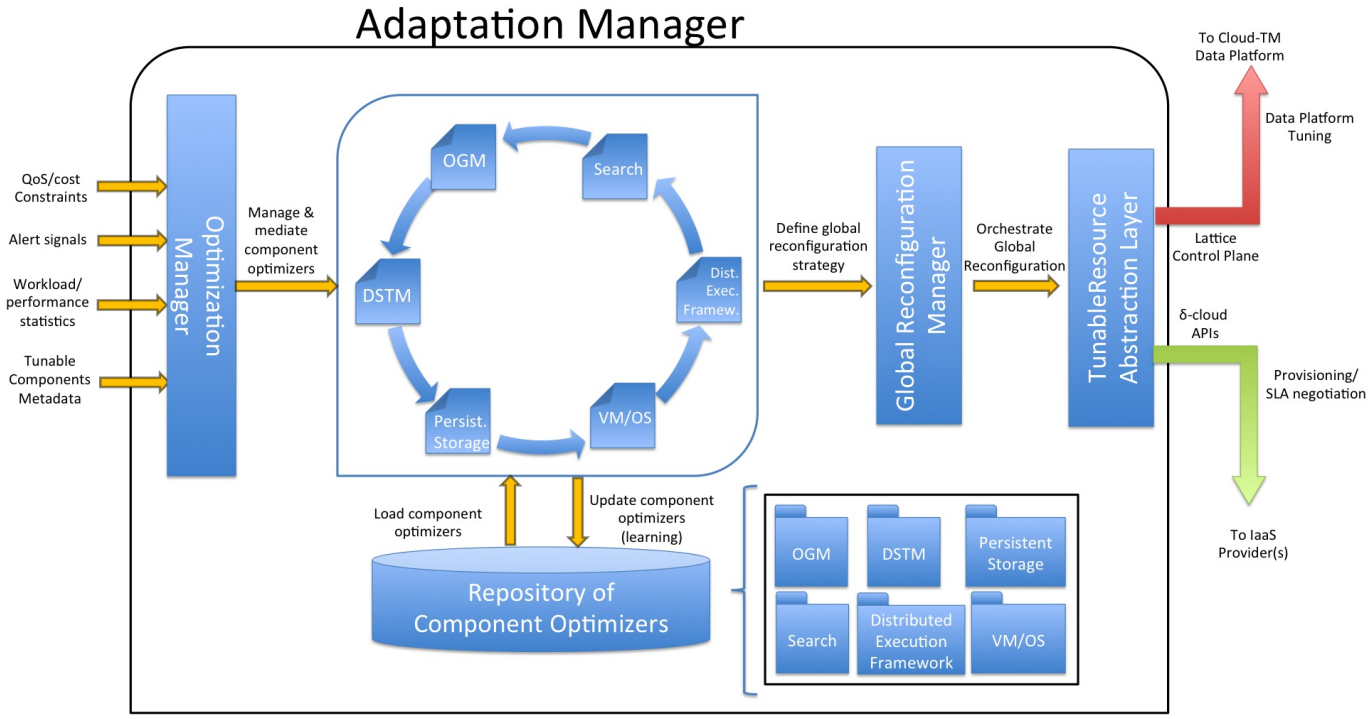



Figure 27: Architectural overview of the Adaptation Manager.

- alert signals, performance/cost statistics and workload characterization information generated by the workload analyzer (see Section 4.3);
- the meta-data of the various tunable components of the Cloud-TM platform (see Section 3.1).

The key role of the Optimization Manager is to manage the life cycle (instantiate, interface and trigger the activation) and mediate the outputs of the set of component optimizers that will be developed throughout the project.

The Cloud-TM Data Platform is in fact constituted by a complex ecosystem of components, each one associated with specific key performance indicators, utility functions and monitorable/tunable parameters. Further, due to the layered architecture of the Data Platform, and to non-trivial interdependencies existing among its constituting components, it is natural to expect that the configuration strategy of a given component, say *Comp*, may strongly affect not only the efficiency of that same component, but also of the efficiency components that interact (directly or indirectly) with Comp.

Classic monolithic optimization approaches, which try to optimize the system as-a-whole by modeling the dynamics of all of its components using a single, unified

approach, are extremely complex to use in a large and heterogeneous system such as the Cloud-TM platform. In order to master complexity, our plan is to use a *divide-et-impera* approach, which subdivides the global optimization task into a set of simpler local optimization subproblems, focused on determining the optimal configuration policy for small subsystems of the Data Platform. The optimization process proceeds then in an iterative fashion, propagating the effects of the tuning of each component (such as shifts of the workload characteristics for other components, or alteration of the demands of shared system resources) along a chain that captures the mutual interdependencies existing among the Data Platform's components.

The Optimization Manager is the coordinator of this concerted optimization process, triggering its activation periodically, or upon reception of alert signals generated by the Workload Analyzer's module, and mediating the outputs (i.e. reconfiguration policies) of the various local optimizers in order to identify a globally optimal reconfiguration strategy (or at least as close as possible to the global optimum).

An important research line that we intend to pursue is to investigate how to combine optimization techniques of different nature, including analytical and simulative models, as well as approaches based on machine learning and control theory. Each of these methodologies comes in fact with its pros and cons, and results particularly adequate to control different classes of systems. Model based performance forecast approaches (e.g. relying on analytical or simulative techniques) typically demand very short training time, but can only be employed if the model developer has a detailed knowledge of the system's internals and dynamics. On the other hand, model-free control methods, which treat the system to control as black-boxes and learn their dynamics using statistical techniques, are more generic and robust than model-based techniques (whose accuracy is affected by the accuracy of the employed system's model), but they are better suited to deal with low-dimensional control problems as their learning time typically grows exponentially with the number of variables to be monitored/controlled.

In order to allow coexistence and interoperability among optimizers of heterogeneous nature, we defined a GenericOptimizer interface, shown in Figure 28, which shall have to be implemented by every component's optimizer. This interface is indeed very simple, as it basically exposes getters and setters methods for the input and output parameters for the optimizers, or more precisely:

1. the parameters that the optimizer should monitor in order to optimize the component;

2. the optimal configuration values for the component's tunable parameters;

3. any side effect that the optimal setting of the tunable parameters is expected to have on different system's parameters (e.g. higher/lower consumption of shared resources, such as CPU or memory). Note that this is what allows to capture the existence of mutual interdependencies among the different components of the platform. This information can in fact be propagated by the OptimizationManager along the chain of components' optimizers, whose individual outputs can then be mediated in order to drive the system towards a globally optimal reconfiguration. How to achieve this goal is clearly an open research issue, which will be object of investigation throughout the project.

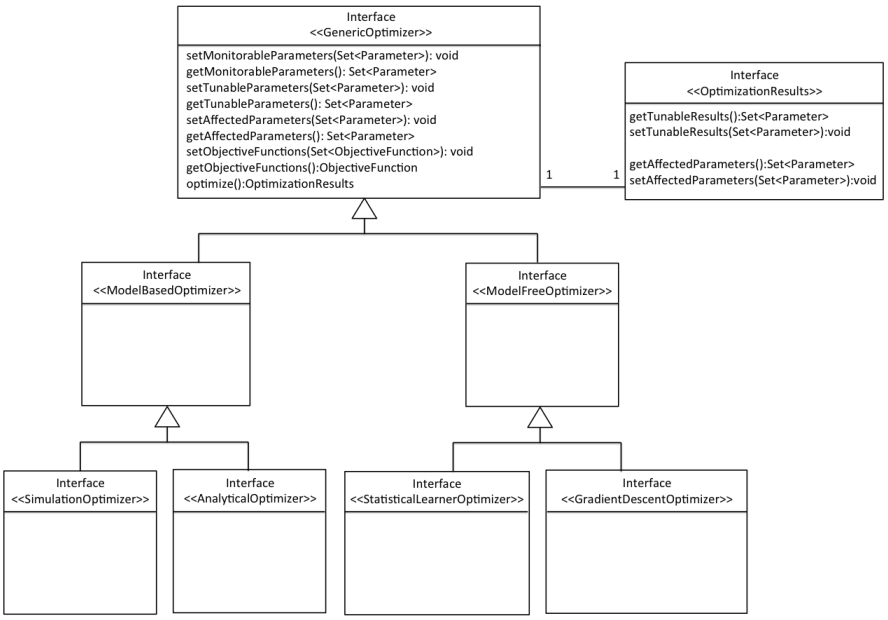



Figure 28: UML Diagram of the Main Interfaces for Encapsulating the Behaviours of Individual Component Optimizers.

Returning to the diagram in Figure 27, the Adaptation Manager will store the Data Platform component's optimizers within an apposite repository that will allow both the retrieval and update of optimizers logic and data (e.g. training data-sets or other statistical information gathered by observing the results of previous self-tuning cycles).

The Adaptation Manager will include two additional building blocks, the Global Reconfiguration Manager and the TunableResource Abstraction Layer.

The former will be in charge of coordinating global reconfiguration actions that entail orchestrating different layers of the Data Platform, such as provisioning new nodes from the IaaS provider to the Data Platform, and having them join the Transactional In-memory Data Grid. Note that our architecture for reconfiguration is inherently hierarchical. Acting from a privileged, centralized perspective, the Global Reconfiguration Manager will in fact be able to trigger reconfiguration of complex systems (such as the In-memory transactional data grid) which may in their turn implement non-trivial distributed protocols to complete their own reconfiguration.

The TunableResource Abstraction Layer will serve the purpose of hiding from the Global Reconfiguration Manager the heterogeneity of interacting with the various platform's components, which, as discussed in Section 3.1, will encompass modules implemented using different technologies and remotely accessible via diverse interfaces. To this end, the TunableResource Abstraction Layer will expose a series of homogeneous stubs implementing the simple Reconfigurable interface defined in Figure 18. Under the scenes, this module will encapsulate the logic to interact with the IaaS provider,

in order to automate the resource provisioning process, and with the Data Platform's components, via the Control Plane of the Lattice framework.

Note that, as shown in Figure 27, in order to achieve transparent interoperability with multiple IaaS Cloud providers, we plan to exploit the δ-cloud APIs [15], a recent open-source project that has recently been accepted into the Apache Software Foundation Incubator, which provides, in its turn, an abstraction layer towards multiple IaaS Cloud providers (including Amazon EC2/S3, GoGrid, OpenNebula, Rackspace Cloud Servers/Cloud Files).

# References


[1] SUN Microsystems, "Java Management Extensions (JMX) v1.2 Specification." http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html, October 2002.

[2] S. Clayman, A. Galis, and L. Mamatas, "Monitoring virtual networks with lattice," in *Network Operations and Management Symposium Workshops*, pp. 239 – 246, 2010.

[3] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *J. Mach. Learn. Res.*, vol. 3, pp. 1157–1182, March 2003.

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, Mar. 1998.

[5] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, (New York, NY, USA), pp. 654–663, ACM, 1997.

[6] JBoss/Red Hat, "INFINISPAN - Open Source Data Grids." http://www.jboss.org/infinispan, 2011.

[7] "Java Caching API." http://jcp.org/en/jsr/summary?id=107, 2001.

[8] SUN Microsystems, "Java Transaction API (JTA)." http://www.oracle.com/technetwork/java/javaee/jta/index.html, 1999.

[9] "ANSI SQL-92 Specification, Document Number: ANSI X3.135-1992." http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt, 1992.

[10] "Infinispan eviction, batching updates and LIRS." http://infinispan.blogspot.com/2010/03/infinispan-eviction-batching-updates.html, 2010.

[11] K. P. Birman, "The process group approach to reliable distributed computing," *Commun. ACM*, vol. 36, pp. 37–53, December 1993.

[12] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Berlin: Springer, 2004.

[13] "Resources and Services Virtualization without Barriers." http://www.reservoir-fp7.eu/.

[14] E. F. Codd, S. B. Codd, and C. T. Salley, "Providing OLAP (On-Line Analytical Processing) to User-Analysis: An IT Mandate," 1993.

[15] "δ-cloud." http://incubator.apache.org/deltacloud/.

# A XML Schema for file descriptors of Generic Tunable Components

**Listing 7: XML Schema for file descriptors of Generic Tunable Components**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema elementFormDefault="qualified"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="tunableComponent">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="tunableOptions"/>
            <xsd:element ref="monitorableAttributes"/>
            <xsd:element ref="KPIs"/>
            <xsd:element ref="utilityFunctions"/>
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:string" use="required"/>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="tunableOptions">
      <xsd:complexType>
         <xsd:choice maxOccurs="unbounded" minOccurs="1">
            <xsd:element ref="tunableOption"/>
         </xsd:choice>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="tunableOption">
      <xsd:complexType>
         <xsd:choice maxOccurs="1" minOccurs="1">
            <xsd:element ref="algorithm"/>
            <xsd:element ref="parameter"/>
         </xsd:choice>
            <xsd:attribute name="id" type="xsd:string" use="required"/>
            <xsd:attribute name="componentName" type="xsd:string"/>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="algorithm">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="implementations"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="implementations">
      <xsd:complexType>
         <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element ref="implementation"/>
         </xsd:choice>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="implementation">
      <xsd:complexType>
         <xsd:attribute name="name" type="xsd:string" use="required"/>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="parameter">
```

```
    <xsd:complexType mixed="true">
       <xsd:sequence>
          <xsd:element ref="Type"/>
          <xsd:element ref="minValue"/>
          <xsd:element ref="maxValue"/>
       </xsd:sequence>
    </xsd:complexType>
       </xsd:element>
          <xsd:element name="Type" type="xsd:string"/>
          <xsd:element name="minValue" type="xsd:decimal"/>
          <xsd:element name="maxValue" type="xsd:decimal"/>
          <xsd:element name="monitorableAttributes">
    <xsd:complexType>
       <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element ref="monitorableAttribute"/>
       </xsd:choice>
    </xsd:complexType>
 </xsd:element>
 <xsd:element name="monitorableAttribute">
    <xsd:complexType mixed="true">
       <xsd:sequence>
          <xsd:element ref="name"/>
          <xsd:element ref="type"/>
          <xsd:element ref="minValue"/>
          <xsd:element ref="maxValue"/>
      </xsd:sequence>
    </xsd:complexType>
 </xsd:element>
 <xsd:element name="name" type="xsd:string"/>
 <xsd:element name="type" type="xsd:string"/>
 <xsd:element name="KPIs">
    <xsd:complexType>
       <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element ref="KPI"/>
       </xsd:choice>
    </xsd:complexType>
 </xsd:element>
 <xsd:element name="KPI">
    <xsd:complexType>
       <xsd:sequence>
          <xsd:element ref="relevantMonitorableAttributes"/>
          <xsd:element ref="relevantTunableOptions"/>
       </xsd:sequence>
       <xsd:attribute name="name" type="xsd:string" use="required"/>
    </xsd:complexType>
 </xsd:element>
 <xsd:element name="relevantMonitorableAttributes">
    <xsd:complexType>
       <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element ref="relevantMonitorableAttribute"/>
       </xsd:choice>
    </xsd:complexType>
 </xsd:element>
 <xsd:element name="relevantMonitorableAttribute">
    <xsd:complexType>
       <xsd:attribute name="id" type="xsd:string" use="required"/>
       <xsd:attribute name="componentName"
```

```
                type="xsd:string" use="required"/>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="relevantTunableOptions">
      <xsd:complexType>
         <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element ref="tunableOption"/>
         </xsd:choice>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="utilityFunctions">
      <xsd:complexType>
         <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element ref="utilityFunction"/>
         </xsd:choice>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="utilityFunction">
      <xsd:complexType>
         <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element ref="argument"/>
         </xsd:choice>
         <xsd:attribute name="classpath" type="xsd:string"
                        use="required"/>
         <xsd:attribute name="priority" type="xsd:string"
                        use="required"/>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="argument">
      <xsd:complexType mixed="true">
         <xsd:sequence>
            <xsd:element ref="type"/>
            <xsd:element ref="value"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="value" type="xsd:string"/>
</xsd:schema>
```