Cloud-TM

Specific Targeted Research Project (STReP)

Contract no. 257784

D3.2: Workload Analyzer

Date of preparation: 10 January 2012
Start date of project: 1 June 2010
Duration: 36 Months

Contributors

Bruno Ciciani, CINI
Diego Didona, INESC-ID
Pierangelo Di Sanzo, CINI
Roberto Palmieri, CINI
Sebastiano Peluso, CINI
Francesco Quaglia, CINI
Paolo Romano, INESC-ID
Heiko W. Rupp, RED HAT

Contents

1 Introduction

This document is a companion of the software release associated with deliverable D3.2 of the Cloud-TM project, namely the prototype implementation of the Workload Analyzer (WA). The document presents design/development activities carried out by the project's partners in the context of WP3 (Task 3.1).

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

- providing an overview of the WA architecture, including a description of (i) its core mechanisms/functionalities and (ii) the technologies used for the implementation of the prototype, and how they were integrated
- describe how to install and configure (README) the release of the WA prototype.

The structure of this document is the following. In Section 2 we overview the set of key functionalities provided by the WA, and how it fits in the architecture of the Cloud-TM platform.

Section 3 describes the architecture of the main components of the Workload Analyzer and how they have been integrated to achieve efficient interoperability with the ecosystem of components composing the Cloud-TM platform.

Next, we describe in detail the key functionalities provided by the WA, and some insights on their implementation. Specifically, in Section 4 we discuss the data aggregation and filtering functionalities. The set of statistics gathered by the WA are reported in Section 5, focusing in particular on the metrics selected/introduced in order to characterize the transactional workload of Cloud-TM applications. Section 6 discusses the load forecasting functionalities provided by the WA, which are based on the ample library of time-series analysis techniques provided by the R-project [1]. Section 7 focuses on the alert definition and QoS monitoring functionalities.

Finally, in Section 8, we provide instructions on how to install and test the WA.

1.1 Relationship with other deliverables

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

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

2 Key functionalities of the Workload Analyzer

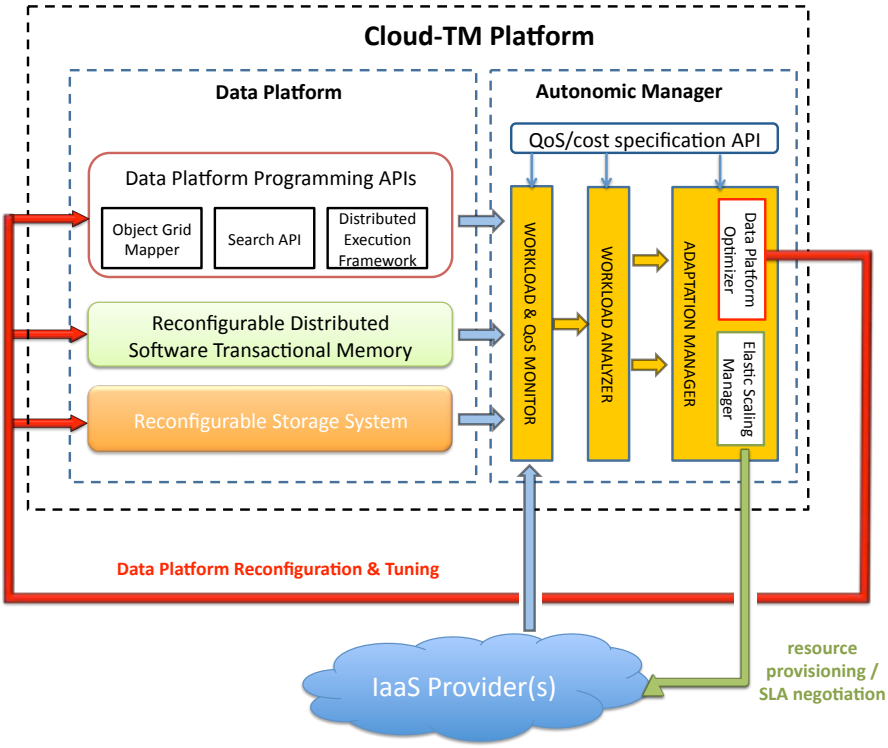


Figure 1: Architectural Overview of the Cloud-TM Platform.

Figure 1 shows where the Workload Analyzer (WA) fits in the global architecture of the Cloud-TM platform. Sitting between the Workload and Performance Monitor (WPM) and the Adaptation Manager (AM), the WA bears the following responsibilities in the Cloud-TM platform (see Figure 3):

- **Data aggregation and filtering**: the streams of monitoring data produced by the distributed nodes of the Cloud-TM platform via the WPM are gathered by the WA, which exposes programmatic APIs and web-based GUIs allowing for aggregating/filtering statistics originated by different software layers and/or groups of nodes.
- **Workload and resource demand characterization**: the WA allows for deriving detailed transactional profiles that include a number of statistical information characterizing the resource usage demand of applications deployed in the Cloud-TM platform both at the physical (e.g. CPU, memory, etc.) and data (e.g. probability of conflicts among transactions, identification of hot spots for lock contention and remote reads) levels.

- **Workload and resource demand prediction**: the WA integrates a set of scripts/interfaces allowing for using the ample library of statistical functionalities implemented by the R [1] free software project. This opens the possibility to run a wide range of time-series analysis methods (such as, moving averages, ARIMAX models, Kalman filters [2]) aimed to forecast future trends of the workload fluctuations.
- **QoS monitoring and alert notification**: the WA allows for graphing raw or aggregated statistics (e.g. on the performance or availability of some set of services), and defining complex alert conditions on the base of the collected data.

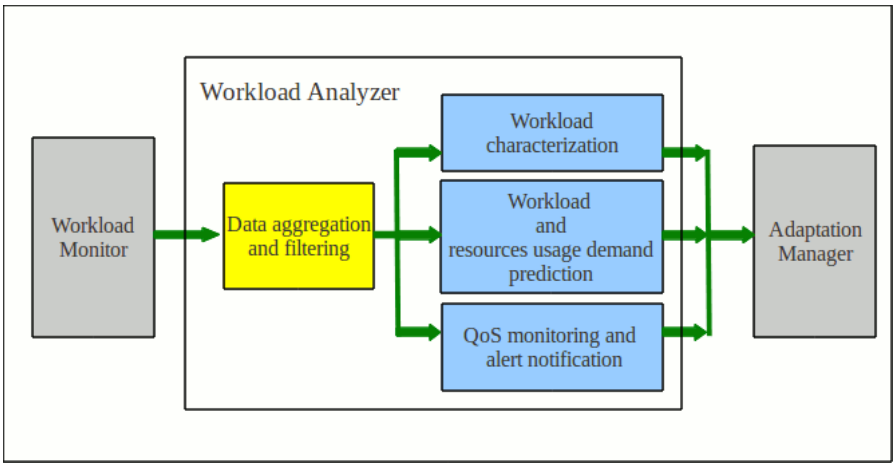


Figure 2: Key functionalities offered by the Workload Analyzer.

3 Architecture of the Workload Analyzer

A key decision taken within the Cloud-TM consortium was to build the WA by capitalizing on several existing open-source software packages, which currently represent leading solutions in their application domain:

- **RHQ**: The RHQ project is a popular systems management suite that provides extensible and integrated systems management for multiple products and platforms. Developed as an open source by Red Hat, RHQ [3] is the upstream to JBoss Operations Network. RHQ is licensed under the GPL, with some pieces individually licensed under a dual GPL/LGPL license.

  The project is designed with layered modules that provide a flexible architecture for deployment. It delivers a core user interface that provides audited and historical management across an entire enterprise. A Server/Agent architecture provides remote management and plug-ins implement all specific support for managed products.

  The RHQ project provides industrial-quality implementations of some of the key functionalities required by the WA (and more in general by Cloud-TM's Autonomic Manager, see Figure 1), including:

  - monitoring and graphing of values
  - alerting on error conditions
  - remote configuration of managed resources
  - remote operation execution
  - provisioning of software onto managed machines

  At the light of the above consideration, the Cloud-TM consortium has taken the key decision of developing the WA around the RHQ framework. This choice, not only has avoided to "reinvent the wheel", but also permitted to take advantage of the know-how of the Red Hat team to assist the researchers from CINI and INESC-ID in the integratinon of RHQ within the Cloud-TM platform. In addition, it enabled the academic researchers to re-use a set of robust, highly-tested, open-source building blocks, and to focus their efforts on developing innovative methodologies and tackle fundamental research challenges (such as defining automated policies for the self-optimization process of the Cloud-TM platform). By electing a popular framework such as RHQ as the starting point/reference for the development of the Cloud-TM WA (and, more in general, as the skeleton of the Cloud-TM Adaptation Manager), the Cloud-TM project has also the possibility to maximize the visibility of its research results, by reaching the mass of users that already use RHQ to manage their distributed applications.

- **Stream-lib**: developed by Clearspring Technologies©, and distributed under the Apache license, stream-lib is an open-source JAVA library that integrates a number of recent algorithms for summarizing data in streams on-the-fly, namely avoiding to store all events in the stream [4].

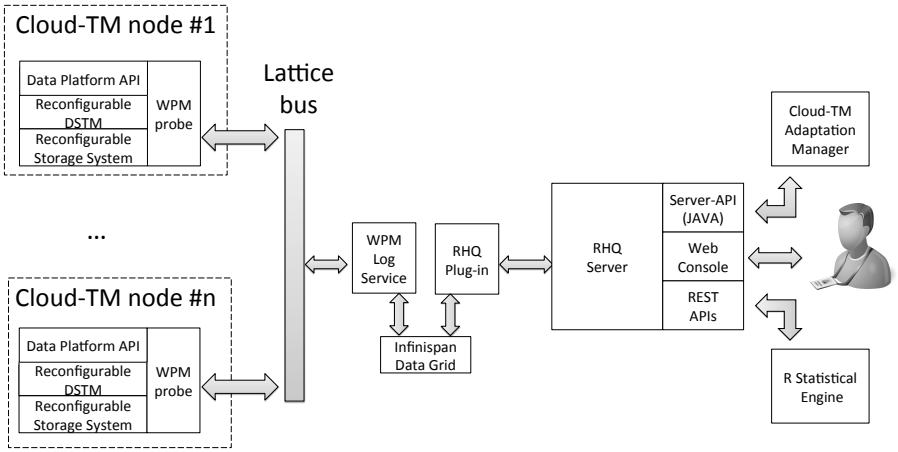


Figure 3: Architectural Scheme of the Workload Analyzer.

As we will detail in Section 5, we rely on Stream-lib to identify, using lightweight probabilistic top-k algorithms, hot spots of different nature in the data access patterns generated by transactions running in the Cloud-TM platform.

- **R-project**: R is a language and environment for statistical computing and graphics. It is a GNU project similar to the S language and environment that was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R provides a wide variety of statistical, including linear and nonlinear modelling, classical statistical tests, time-series analysis, classification and clustering.

  As we will detail in Section 6, the WA exploits the ample library of statistical functions provided by R in order to derive workload forecasts using a range of time series analysis methodologies (ranging from simple moving averages, to more complex ARMA models and Kalman filter-based predictors), allowing for identifying trends and seasonal components in the incoming streams of statistics.

The diagram in Figure 3 depicts the architecture of the WA. Let us analyze it more in detail, discussing how the above mentioned open-source projects have been extended and integrated in the Cloud-TM platform architecture.

A first important step was to extend the set of statistics exported by the components of the Cloud-TM Data Platform (and in particular of Infinispan [5]) in order to generate a detailed profile of the transactional workload in input to the system. More details on this can be found in Section 5.

The next step was to integrate the WPM framework (described in Deliverable D3.1) with the RHQ infrastructure. This was achieved by developing an ad-hoc RHQ plug-in, designed to be fully compatible with the WPM's Log Service component (LS) output. The plug-in externalizes to RHQ the statistics collected by the Cloud-TM nodes that

are being monitored by the WPM. In order to decouple the LS from its RHQ plug-in (to allow them to be deployed on different machine, for instance), the plug-in registers a set of listeners on an Infinispan cache, which is populated by the LS whenever a new sample (or batch of samples) is gathered from the monitored nodes. This way, we take advantage from the fault-tolerance features of Infinispan in order to ensure high availability of the communication bus between LS and its RHQ plug-in.

Note that, with this architectural organization, we are are replacing the native RHQ monitoring infrastructure with the one provided by the Cloud-TM's WPM (that had been already developed in Deliverable D3.1). Experiments are currently on-going to evaluate the efficiency and robustness of the two solutions and take an informed decision on which of the two monitoring platforms to adopt in the final release of the Cloud-TM platform.

Finally, once the monitoring data is convoyed by RHQ, we use its rich set of APIs and interfaces as building blocks to support a breadth of functionalities for workload analysis and forecasting. These include statistics visualization, aggregation and filtering, alert definition and monitoring of QoS levels, as well as exporting data (via REST interfaces) for analysis with the R statistical framework.

4 Data aggregation and filtering

In order to support aggregation and filtering of incoming monitoring data streams, the WA exploits the advanced grouping functionalities provided by RHQ. A detailed description of this RHQ functionality can be found in [6], but, for the sake of self-containment, we provide a short overview also in the following.
In RHQ, groups serve a twofold purposes:

- defining which access permissions are applied to the resources monitored by RHQ;
- providing a way to view aggregate data and perform actions across all group members en mass.

RHQ enables flexible group membership policies with which:

- users can manually add resources to a group "one-by-one" for fine-grained membership control via a web console
- users can create expressions / rules that dynamically maintain groups membership (DynaGroups). DynaGroups allow for defining expressions based grouping conditions on the various attributes made available by the resources registered in RHQ. A screenshot displaying how DynaGroups can be defined in RHQ is shown in Figure 4. Note that membership in DynaGroups is dynamic in the sense that it is periodically tested for the presence/absence of members belonging to the group.

Once groups are defined, it is possible to specify control access polices directly to groups of resources, instead of individual resources. By using DynaGroups, one can effectively create dynamic ACLs (access control lists) to lessen the burden of security maintenance, especially against large inventories.

Compatible groups (those composed entirely of the same type of resource [e.g. all Linux platforms, all JBossAS servers]) have additional features available to them:

- group-wise availability;
- min, max, and average metrics across the group;
- aggregate events viewer;
- operations against all group members, either serialized or concurrent execution policies;
- fine-grained changes to connection properties and resource configuration across one or more members of the group.

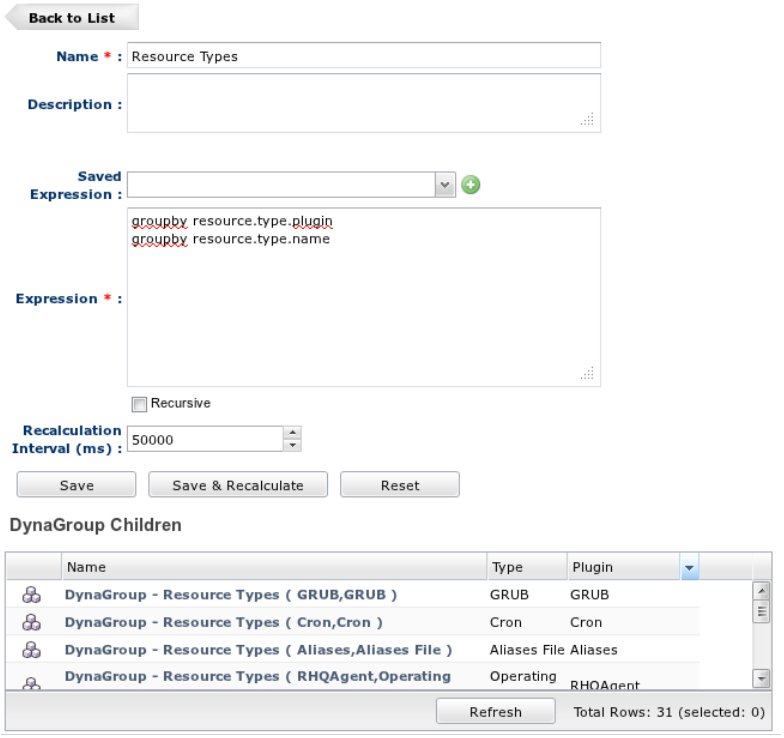


Figure 4: Example of DynaGroups' definition in RHQ.

5 Workload and resource demand characterization

A significant effort has been put in implementing probes collecting a number of statistics aimed at characterizing the transactional profile of the applications deployed over the Cloud-TM platform.

The probes were inserted in the key component that is responsible for the transactional management of data, Infinispan. We recall (see Deliverable D2.1: "Architecture Draft") that Infinispan is a transactional data grid platform developed as an open-source project that was selected by the Cloud-TM consortium as the starting point to develop the Cloud-TM's Reconfigurable Distributed Software Transactional Memory platform. In the context of WP2, in fact, Infinispan is being extended with a library of replication schemes and with mechanisms for allowing the dynamic reconfiguration of the protocols used to ensure data consistency across the nodes of the data grid.

The additional statistics introduced within the Cloud-TM project have been integrated in Infinispan's pre-existing statistic collection mechanism. In fact, Infinispan already collects a significant number of statistics on the performance and status of several of its subcomponents (e.g. Lock Manager, Distribution Manager, RpcManager, see [7] for a complete reference), and externalizes them using JMX interfaces allowing for their monitoring via standard JMX-based consoles (and by the WPM).

The new statistics have been collected using probes scattered across the LockManager, the LockingInterceptor, the DistributionInterceptor, the ReplicationInterceptor, the TxInterceptor, the CacheMgmtInterceptor and the RPCManager. Furthermore, we introduced a new Interceptor, the StreamLibInterceptor, which runs efficient stream analysis algorithms (to be described shortly). The new statistcs are then published via JMX interfaces together with the original Infinispan statistics.

The full set of additional statistics collected from Infinispan is reported in Table 1 and Table 2, classifying them into *high-level* and *low-level* statistics, which are described in the following.

5.1 High Level Statistics

High-level statistics can be, in turn, distinguished in two classes:

1. statistics tracking keys that represent hot spots for two essential subsystems of a data grid: the data placement and concurrency management schemes. More in detail we trace the top-k keys (where k is a parameter that is dynamically configurable via JMX) that have been:

   i) updated (using the put command);

   ii) either remotely or locally read - thus requiring or not a remote interaction with another node during transaction execution;

   iii) locked, causing either i) no contention, ii) contention, or iii) abort, of a transaction.

   Note that this information is extremely valuable for the automatic and human-driven tuning of these performance-critical modules of the system, and we plan to

High Level Statistics

Statistic name	Short description
Application Contention Factor	The Application Contention Factor is a measure of the max degree of concurrency achievable by a transactional application given its data access pattern.
Top-K put	Map containing the k keys for which it has been more frequently requested a put operation (together with the estimated number of times the key has been put).
Top-K local-get	Map containing the k *local* keys for which it has been more frequently requested a get operation (together with the estimated number of times the key has been read locally).
Top-K remote-get	Map containing the k *remote* keys for which it has been more frequently requested a get operation (together with the estimated number of times the key has been read remotely).
Top-K locked	Map containing the k keys for which it has been more frequently requested a lock operation (together with the estimated number of times the key has been locked).
Top-K contended	Map containing the k keys for which it has been more frequently encountered lock contention (together with the estimated number of times there has been lock contention on the key).
Top-K abort	Map containing the k keys that have have more frequently caused the failure of a transaction due to contention (together with the estimated number of times there has been lock contention on the key).

Table 1: High level statistics

make use of this info into the Autonomic Manager component to drive different of self-optimizing strategies.

In order to minimize overheads, we identify these keys using recent results from literature on data stream analysis. In particular we used the *top-k* algorithm presented in [4] (implemented by the stream-lib opensource project [8]): unlike classic solutions that provide exact guarantees at the cost of storing a possibly unbounded amount of information, this algorithm analyzes streams using a limited (constant) memory space, thus optimizing performance and lending itself to the analysis of massive streams of data.

2. Application Contention Factor: Another key high level statistic computed by the Workload Analyzer is an innovative metric, which we named Application Contention Factor (see the technical report [9] for more details on it), that allows for characterizing the maximum degree of data parallelism exhibited by transactional applications.

   In order to explain more rigorously its definition, it is required to introduce some background concepts at the basis of the analytical performance modelling approaches of transactional systems presented so far in literature. Existing works in this area [10, 11] share a common reliance on queuing theoretical arguments to derive the transaction contention probability. Denoting with λ the average arrival rate of locks to a data item, and assuming that locks are held for an average time T_H, one can model a data item as a queue and approximate the probability of encountering lock contention on a data item with the utilization of the corresponding queue (namely, the fraction of time during which the data item is locked), which is computable as [12]:

   $$U = \lambda_{lock} T_H$$

   assuming $\lambda_{lock} T_H < 1$. Then, assuming that accesses are uniformly distributed on one [11] (or more [13]) set of data items of cardinality D, a-priori known, it

is possible to compute the probability of lock contention on any of the data items simply as:

$$P_{lock} = \frac{1}{D}\lambda_{lock}T_H \tag{1}$$

Unfortunately, the availability of information on D, and the assumption on the uniformity of the data access patterns strongly limits the employment of these models with complex applications, especially if these exhibit dynamic shifts in the data access distributions.

The idea underlying the definition of the Application Contention Factor (ACF) is to extract the equivalent value of D for an application in execution on the Cloud-TM platform by exploiting the availability of information on P_{lock}, λ_{lock} and T_H in the current configuration. Given P_{lock}, λ_{lock} and T_H, in fact, we can invert Eq. 1 and obtain the Application Contention Factor (ACF) as:

$$ACF = \frac{P_{lock}}{\lambda_{lock}T_H} \tag{2}$$

By equation 1, it follows that $\frac{1}{ACF}$ can be interpreted as the size D of an "equivalent" set $\mathcal{DB}$ of data items, such that, if the application issues lock requests on disjoint data items selected uniformly from set $\mathcal{DB}$, it would incur in the same contention probability that it experienced in the current configuration.

From an other perspective, the ACF (or better, its inverse) represents the maximum number of transactions that can be concurrently executed in the system assuming that each transaction holds its locks for a single time unit. The ACF allows for characterizing the application data access pattern distribution in a very concise, lightweight and pragmatical manner, abstracting over arbitrarily complex data access patterns (e.g. with strong skew or complex analytical representation) and over the effects of contention on physical resources (abstracted away by normalizing the ACF with respect to T_H) via an easily tractable analytical model.

This result represents the foundation on top of which we are building analytical models of the lock contention dynamics. These models aim to determine the contention probability that would be experienced by that same application in presence of different scenarios of workloads (captured by shifts of λ_{lock}), as well as of levels of contention on physical resources (that would lead to changes of the execution time of the various phases of the transaction life-cycle, capturable by shifts of the T_H).

In Figure 5 and Figure 6 we report the ACFs and, respectively, transaction commit probability obtained when running two well-known benchmarks, TPC-C [14] and Radargun [15], configured to generate very heterogeneous workloads for what concerns both the data access skew and contention probability.

TPC-C is a standard benchmark for OLTP systems (of which we ported an implementation to execute on top of Infinispan), which portrays the activities of a wholesale supplier and generates mixes of read-only and update transactions with strongly skewed access patterns and heterogeneous duration. Radargun,

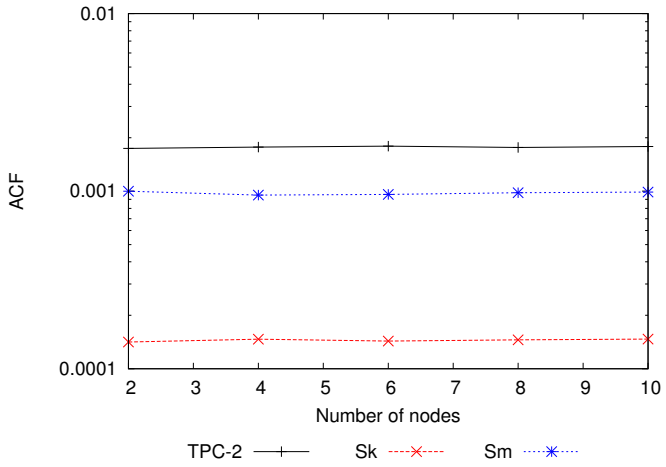


Figure 5: ACF of heterogeneous workloads.

instead, is a benchmarking framework specifically designed to test the performance of distributed, transactional key-value stores. The workloads generated by Radargun are much simpler and less diverse than TPC-C's ones, but have the advantage of being very easily tunable.

For TPC-C we consider a workload (TPC-2) that include around 50% of update transactions and generate high contention. For Radargun we consider two workloads: Sk, which generates transactions that issue 10 writes distributed on a set of 100K keys and selected according to a highly skewed distribution (as defined by the NuRand(100000,8191), used by several TPC benchmarks); Sm, which uses a uniform data access pattern updating in each transaction 10 data items selected over a set of cardinality 1K. All the results reported in this section were collected using a private cloud of 10 servers equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 8 GB of RAM, running *Linux 2.6.32-33-server* and interconnected via a private Gigabit Ethernet.

The plots shows that, once fixed an application workload, and even when considering very skewed workloads, the ACF (see Figure 5), unlike the commit probability (see Figure 6), is invariant as the size of the underlying data grid varies. This confirms the appropriateness of the ACF to characterize application's data access patterns in a way that is independent from the current degree of parallelism in the system (unlike for instance the transaction commit probability) and of the actual data access pattern distribution.

5.2 Low Level Statistics

The set of additional low level statics gathered from each individual Infinispan node, reported in Table 2, is oriented to provide a detailed characterization of the performance

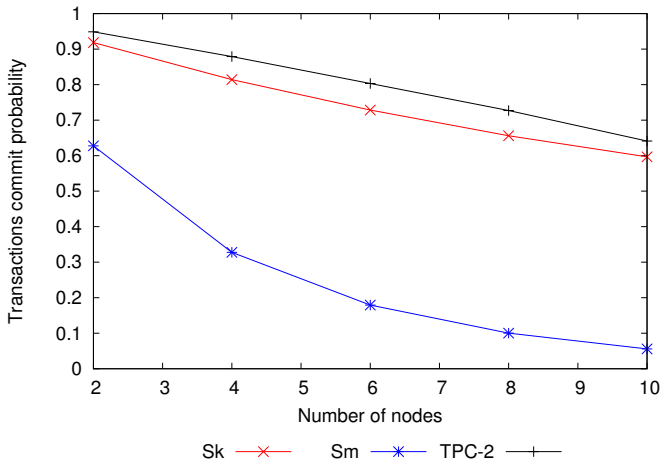


Figure 6: Transaction commit probability of heterogeneous workloads.

and costs of the main subsystems involved in the processing of transactions along its life-cycle. These include both statistics (mean, and percentiles) on metrics typically used in SLAs (for instance, transaction execution time) and statistics useful for modelling purposes, such as the latency experienced by transactions along their various execution stages, the frequency of different types (write vs read) of transactions and of various contention-related events (e.g. successful vs failed lock acquisition).
Among these, two types of statistics are particularly noteworthy:

- the probability distribution of lock inter-arrival time: this information, encoded as an histogram, allows verifying whether one critical assumption holds for the applicability of Equation 1, namely, whether the lock arrival rate can be approximated by an exponential distribution. Equation 1, in fact, is guaranteed to hold only in case the lock requests arrival rate is poissonian, a condition sufficient to ensure the PASTA (Poissonian Arrival See Time Averages) property [16].

  The data reported in Figure 7 shows an example of three lock inter-arrival time distributions that were obtained by configuring Radargun to generate transactions accessing data using different data access patterns (uniform vs skewed) on keysets of different sizes (1K vs 100K). By observing the graph, it is clear that the above parameters have a significant impact on the shape of the empirical lock inter-arrival time distributions, which present, at high skew or contention levels, spikes that are symptomatic of non-poissonian behaviors that can have an impact on the accuracy of the modelling methodology at the basis of the computation of the ACF.

  By comparing, via Good of Fitness tests [17], the empirical lock arrival rate with (best-fitting) exponential distributions (or with other distributions for which the PASTA property holds, such as uniform distributions), one can therefore obtain a measure of the expected accuracy of the ACF in predicting the maximum degree

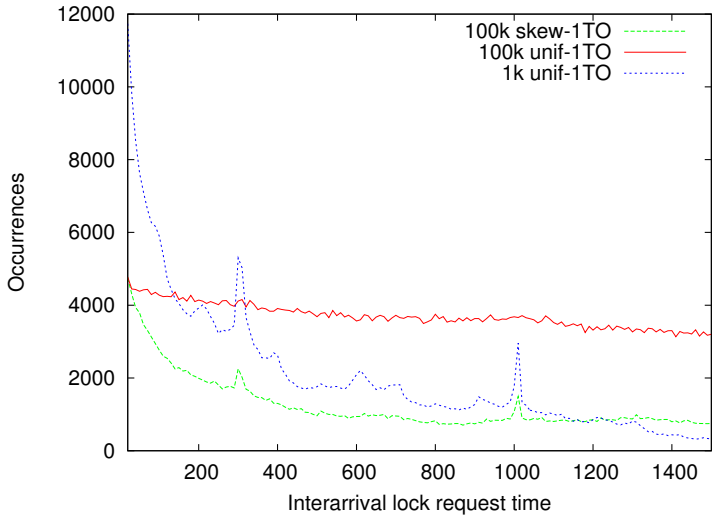


Figure 7: Distribution of lock inter-arrival time using three different Radargun workloads.

of concurrency for a transactional application.

- percentiles on transaction execution times: percentiles are often preferred to simple averages in SLA negotiations as they provide more meaningful guarantees on the actual QoS delivered to the population of end users of a system. On the other hand, computing exact percentiles requires storing all the samples across the considered time window, or solving the problem of determining (statically or dynamically) an appropriate binning size [18].

  In order to avoid the above complexity, we compute percentiles using Vitters reservoir sampling algorithm [21], which over time gives us an appropriate model for the distribution of the transaction execution lengths. Vitters algorithm (shown in Figure 8) fills an initially empty reservoir (array) of size n with the first n samples. Then, each k-th element is inserted in a random spot of the reservoir with a probability of n/k. This ensures an uniform sampling over the stream of data. The requested percentile is obtained by sorting the reservoir and picking the percentile of interest. For instance, to obtain the 95% of the transaction execution time we can simply read the value stored at index $j = n * 0.95$ of the sorted array.

5.3 Thread Level Statistics

The native statistics collection mechanism of Infinispan relies on a set of counters maintained by each node of the data grid. These counters are implemented by means of shared atomic variables that are updated (possibly concurrently) by threads upon the

Low Level Statistics

Statistic name	Short description
Probability distribution of lock inter-arrival time	Histogram containing the distribution of lock requests' inter-arrival time.
K-th percentile of update transactions duration	K-th percentile of update transactions duration
K-th percentile of read-only transactions duration	K-th percentile of read-only transactions duration
Number of nodes involved in a prepare	Average number of nodes involved in a prepare phase
Deadlocks during prepare phases	Number of transactions aborted during prepare phase due to a deadlock
Timeouts during prepare phases	Number of transactions aborted during prepare phase due to a timeout on lock acquisition
Remote get operation execution time	Time needed to perform a get on a remote node (without considering the round trip time)
Size of a PrepareCommand	Average size of a PrepareCommand in bytes
Size of a ClusteredGetCommand	Average size of a ClusteredGetCommand in bytes
Size of a CommitCommand	Average size of a CommitCommand in bytes
Read-only transaction execution time	Average execution time for a read-only transaction that commits
Update transaction execution time	Average execution time for an update transaction that commits
Update transaction local execution time	Average execution time of the local part of an update transaction, i.e. up to the prepare phase
Replication time for an update transaction	Average time needed by the cohorts to replicate modifications contained in a PrepareCommand
Round Trip Time	Time needed to send a PrepareCommand and get the responses, without considering the replication time on the cohorts' side
Local contention probability	Probability that a lock requested by a local transaction is already taken by another one, whether local or remote
Lock waiting time	Average time spent by a transaction before acquiring a lock it is waiting for
Update transaction local execution time in isolation	Average execution time of the local part of an update transaction without considering the time spent to acquire the locks
Lock hold time	Average time that lasts between the acquisition of a lock and its release
RollbackCommand cost	Time spent to process a RollbackCommand
CommitCommand cost	Time spent by a local transaction to process a CommitCommand
Acquired locks	Average number of locks acquired by local transactions that manage to get to the prepare phase
Transactions arrival rate	Average number of transactions that arrive to the system per second
Trhoughput	Number of completed transactions per second
Transactions write percentage	Percentage of transactions that perform at least one put operation, whether they commit or not
Successful transactions write percentage	Percentage of transactions that perform at least one put operation among the committed ones

Table 2: Low level statistics

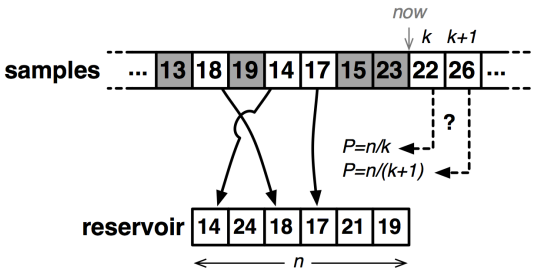


Figure 8: Reservoir sampling algorithm [19] (Figure from [20]).

occurrence of relevant events. E.g., the total number of committed transactions by data grid node is stored by means of an AtomicLong type variable (provided by the package java.util.concurrent.atomic). This variable is shared by all threads of the node and is (atomically) incremented by a thread whenever a transaction is committed.
This approach to gather statistics has two main drawbacks:

- In many transactional applications, different threads have distinct transaction profiles (e.g. read vs write dominated workloads). By aggregated statistics at the data grid node level, it is impossible to capture statistical information that would allow for performing a detailed workload profiling on basis of activity of the different threads.

- On multi-core machines, the presence of these atomic variables tends to increase the cache coherency traffic and imposes the use of low-level atomic constructs (e.g. Compare and Swap), which, typically, rely on costly hardware operations, requiring, e.g., the generation of cache invalidation traffic or locking of system buses. The impact on system performance due to these factors may became relevant with some workload profiles and/or with high concurrency level, and may limit the system scalability. Further, with the introduction of additional statistics in the version of Infinispan tailored for the Cloud-TM Data Platform, the update frequency of the counters is notably increased with respect to the original version.

On basis of the above motivations, the statistical data collection mechanism used in Infinispan has been extended, introducing, as a configurable alternative to the native centralized scheme, also a per-thread data gathering scheme. In the novel mechanism, each thread maintains a set of private copies of counters, one copy for each monitored metric. Upon the occurrence of an event which requires the update of a counter, the thread updates its own copy of the counter avoiding any kind of synchronization. This allows, on one hand, to gather differentiated statistics for each thread. Of course, when statistics at node level are needed, they can be still computed, by collecting the values of the counters of the locally executing threads for the desired metric and by calculating the aggregated value (e.g. the average, the maximum, the minimum). In this implementation, the calculus of an aggregated metric is performed when the metric is queried via its JMX interface. Figure 9 and Figure 10 provide a comparison be-

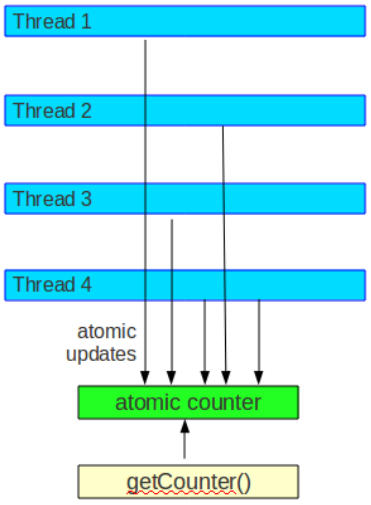


Figure 9: Schema of the centralized statistics collection mechanism natively implemented in Infinispan.

tween the architectures of the collection mechanism used in Infinispan and the novel mechanism.

Implementation considerations. Figure 11 depicts the class diagram of the novel data collection mechanism. The new mechanism is based on a per-thread private set of counting variables, named parameters. This set is defined by the class ThreadStatistics, which implements the interface ISPNStats. The getter and the setter methods read the value and assign a value for a specific parameter, respectively. The input variable index identifies the accessed parameter. The method addParameter(int index, double delta) is used by a thread whenever a parameter has to be updated. This method adds the value delta to the current value of the parameter identified by index. Finally, the method reset() sets to zero all parameters. The privates sets of parameters for each thread are implemented by means of the ThreadLocal class, ensuring that, upon initialization (i.e. upon thread creation), a new ThreadStatistics object is associated with the new thread and a reference to this object is added, atomically, to the StatisticsListManager object. The latter contains a list of ThreadStatistics objects.

The StatisticsListManager allows to access to the statistical data of each thread and provides the methods that calculate the aggregated metrics. When a thread is terminated, the ThreadStatistics object of the thread remains in the list, thus allowing to access statistics also after the thread termination. Statistics belonging to the terminated thread are removed by the list by calling the method clearList().

Note that the set of entries of the list of ThreadStatistics objects managed by StatisticsListManager changes whenever a new thread is created (because a new reference to the ThreadStatistics object of the new thread is added to the list). Concurrently, the

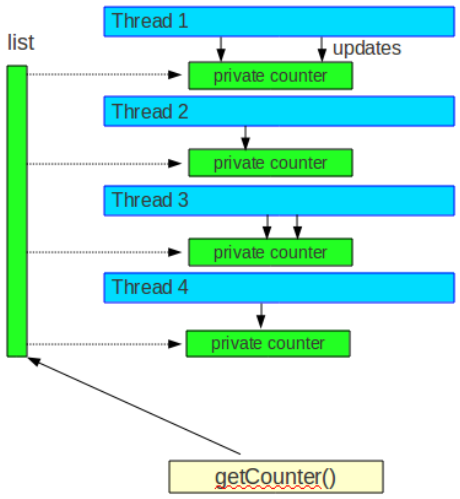


Figure 10: Schema of the new per-thread statistics collection mechanism implemented in Infinispan.

list may be traversed by other threads executing a method which calculate an aggregated metric. Such a method uses the method getParameters(int[] indexes). This latter method receives a list of parameters and, for each parameter, returns the sum of values of the private copies of the parameter of all threads. This is done by means of a list iterator and by traversing the whole list.

Due to the concurrent accesses, the aforesaid list has been implemented as an instance of the class CopyOnWriteArrayList. This is a thread-safe implementation of the List interface, which, in particular, does not block operations that perform list traversals. This improves the responsiveness of the operations calculating aggregated metrics. On the other hand, list insertions pay an extra cost. Anyway, this implementation is optimized for scenarios with a low rate of list updates with respect to the rate of list traversal operations. As the rate of list updates depends on the creation rate of new threads, profitable scenarios are likely to happen in multi-tier architectures tailored for web-based applications, where threads are not created and destroyed for each operation invoked by the users. Instead each operation is executed by a thread belonging to a pre-existing thread-pool.

5.4 Evaluating the Overhead of Statistics Collection

We conclude this section by presenting in Figure 12 the results of an experimental study aimed to assess the impact on Infinispan's performance with the introduction of the new set of statistics described in Table 1 and Table 2. We used a Radargun workload generating transactions with a very reduced conflict probability (performing 1 write access out of 10 operations distributed uniformly across 100K keys), and measured the

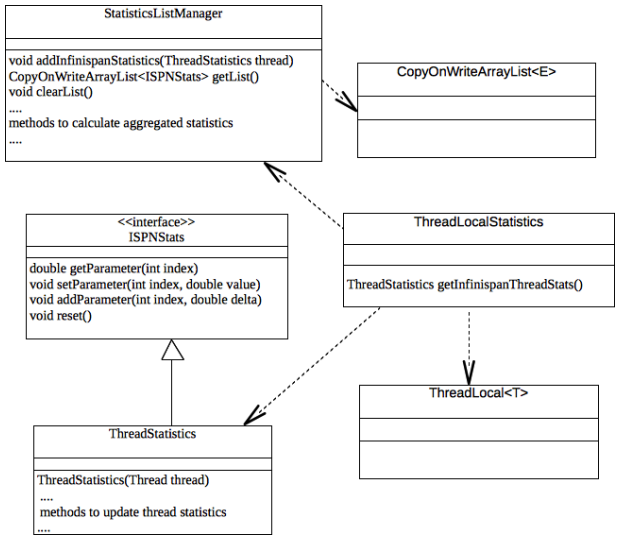


Figure 11: Class diagram of the per-thread data collection mechanism.

throughput (committed transactions per second) achieved when running Infinispan in a single node and on 8 nodes (replication mode). The plots the show that the throughput achieved by Infinispan when gathering the whole new set of statistics (implemented using the per thread collection scheme) is around 2% lower than when totally disabling the statistics collection system.

This confirms the efficiency and feasibility of the proposed workload monitoring and analysis methodology.

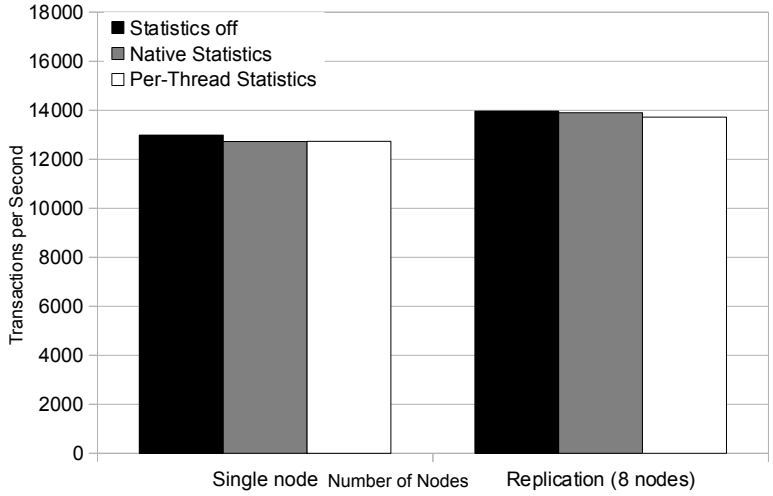


Figure 12: Evaluating the overhead of the statistics collection mechanisms

6 Workload and resource demand prediction

As already mentioned, the WA relies on the powerful R statistical engine in order to perform various time series analysis. This is made possible by exploiting the recently introduced REST APIs of RHQ, which allows exporting the statistical data gathered from the monitored platform as time-series encoded in JSON [22] format.

An example of the potentialities of this approach and the simplicity to access from R to the time series is provided by the Listing 1 in which the $RCurl$ and $rjson$ packages (provided by R) are used to acquire (via REST) and to import into R the vectors of the time series of the last eight hours of a metric. In the reported example, the requested values, uniquely identified by $scheduleId = 10013$, are acquired from the RHQ server listening on port 7080 and running on the same machine on which the R engine is deployed.

The data is then plotted along with its 5% and 95% quantils as well as 20-items simple moving average. Figure 13 shows a plot obtained running this script on example data spanning a 3 days time frame. The metrics are plotted in black, the average in blue, the 5% and 95% quantils in orange and green and with the help of the TTR library, the 50 samples moving average is plotted in red.

Listing 1: Example R listing to produce the graph shown in Figure 13

```
library("RCurl")
library("rjson")

## get raw data for user rhqadmin and schedule 10013 for the last 86400
    sec (=24h)
json_file <- getURL("http://localhost:7080/rest/1/metric/data/10013/raw
    ?duration=86400", httpheader=c(Accept = "application/json"),userpwd
    ="rhqadmin:rhqadmin")
## convert json to list of vectors
json_data <- fromJSON(paste(json_file, collapse=""))

options(digits=16)

## convert into a data frame
df <- data.frame(do.call(rbind,json_data))

## convert timestamps to date expressions in the whole list for the y
    axis
times <- lapply(df$timeStamp, function(x) {format(as.POSIXlt(round(x/
    1000),origin="1970-01-01"),"%H:%M")})

## plot the data
plot(df$timeStamp,df$value,xlab="time",ylab="Free memory (bytes)",xaxt=
    'n',type='b')
## and the labels on the x-axis
axis(1,df$timeStamp,times)

## translate values into a numeric vector to run some analysis on
g<-as.vector(mode="numeric",df$value)

## remove NaN values
h<-g[!is.na(g)]
```

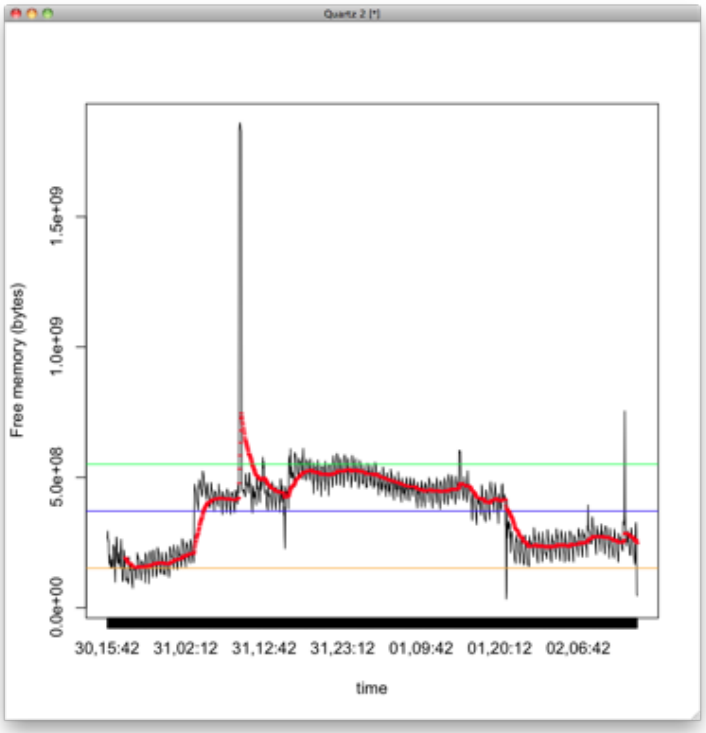


Figure 13: Example plot of time series analysis obtained on data extracted via REST interfaces from RHQ.

```
## Plot line for the avg value
abline(h=mean(h), col="gray")

## Plot markers for 20% and 80% quantiles
abline(h=quantile(h,.20), col="lightblue")
abline(h=quantile(h,.80), col="lightgreen")

## compute and plot 20 items moving average requires library 'TTR'
libFound <- library("TTR", logical.return=TRUE)
if (libFound) {
  points(df$timeStamp,EMA(h,n=20),col="red",pch="-")
}
```

As a final remark, note that by exposing data via REST interfaces, the data gathered by RHQ can be straightforwardly provided as input to a plethora of machine learning tools, and not only to R. In fact, work is currently ongoing, in the context of Task T3.2 "Performance Forecasting Models", in order to automatize data extraction from several popular machine learning tools, such as:

- Rulequest's Cubist© [23]: Cubist© is a decision tree based regression commercial tool developed by Quinlan, the author of C4.5 [24] and ID3, two popular decision tree based classifiers. Analogously to these algorithms, Cubist© builds decision trees choosing the branching attribute such that the resulting split maximizes the normalized information gain (namely the difference in entropy). Unlike C4.5 and ID3, which contain an element in a finite discrete domain (i.e. the predicted class) as leafs of the decision tree, Cubist© places a multivariate linear model at each leaf.

- Weka [25]: Weka is an open-source framework providing a common interface to a large number of machine learning algorithms, including Neural Networks [26], Support Vector Machines [27], decision trees [24] and various data clustering algorithms [28].

7 QoS monitoring and alert notification

As already mentioned, the WA leverages on RHQ's advanced QoS monitoring and alert notification engine. This choice has allowed the researchers of the Cloud-TM consortium to focus on the automatic determination of the policies to be defined to trigger alerts (e.g. associated to elastic-scaling or to the self-tuning of the Cloud-TM platform), rather than on the implementation of yet another alert notification engine.

RHQ's QoS monitoring and alert notification engine is designed to provide proactive notifications about events happening throughout the monitored platform. These events can be resources failing or being disconnected, specific values for metrics being collected, resource configuration being changed, operations being executed, or even specific conditions found by parsing log events.

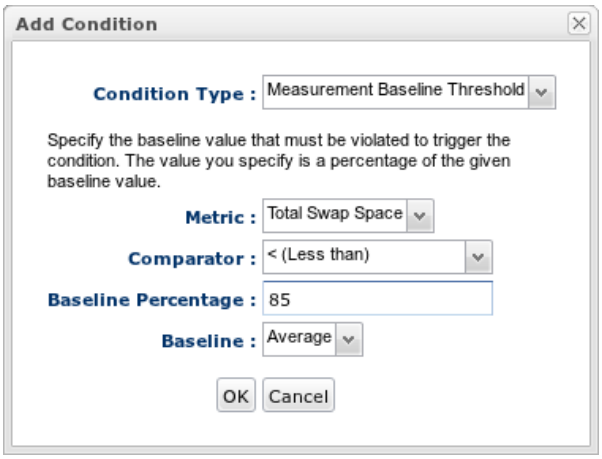
As information flows into the RHQ system, it passes through the alerts processing engine. Here, the data can be transformed, filtered, or even correlated with data from other parts of the system. Users have full control over what they want to be notified about, and RHQ keeps a full audit trail of any conditions that have triggered alerts to fire.

The alerts subsystem provides a wealth of different options for being notified proactively about potential issues in the system. As a result, it supports a breadth of different configuration options that allow for deriving very specific and customized semantics.

A detailed description of these functionalities is reported in [29], but we present in the following a brief summary of the key features that are more relevant to the usage within the context of the Cloud-TM project:

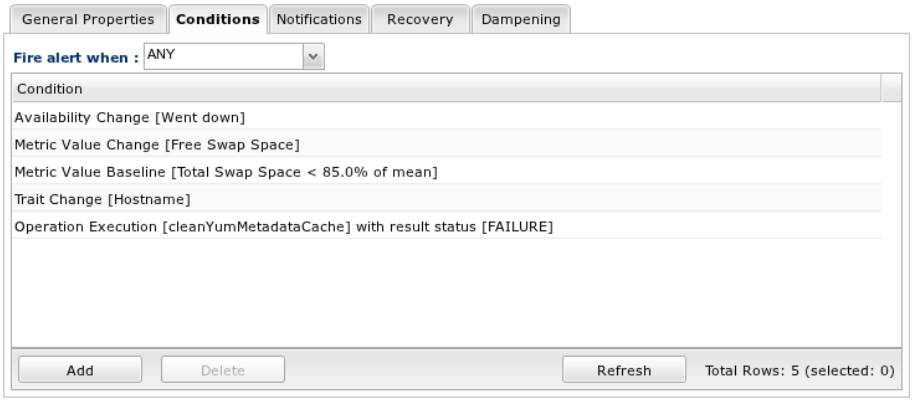
7.1 Alert Definitions & Alert Conditions

Each resource monitored by a RHQ server may have zero or more alert definitions. At the heart of the alert definition is the condition set.



There's no limit to the number of conditions that can be created for a single alert definition, and either all or just one of them needs to be met simultaneous in order for

this definition to trigger an alert.



When an alert definition's condition set is met, an alert is created which serves as the primary piece of audit data in the system. However, several types of external notifications can also be sent, such as:

- an email to a list of explicit email addresses
- an email to a list of RHQ users
- an email to all of the users in a list of RHQ roles
- an SNMP trap
- server-side scripting
- JAVA-based alert plugins

Note that the last option lends itself naturally to trigger, in an efficient way, the logics of Cloud-TM's Adaptation Manager.

7.2 Action Filters & Recovery

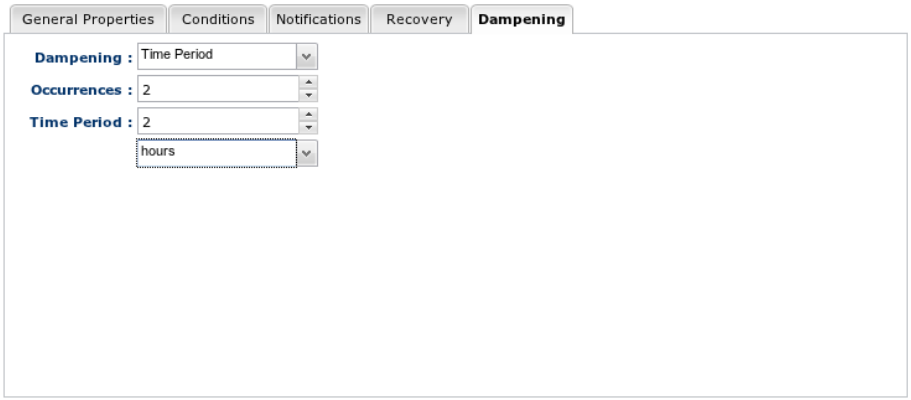
These are hooks that allow the RHQ system to have enhanced control over alerting. In tandem, they help to semi-automate the process of responding to alerts by giving pseudo-intelligence to RHQ itself.

When an alert is triggered, action filters can be used to prevent duplicate alerts while the problem that caused it to fire is being fixed (either by developers or system admins). Recovery can be used to automatically re-enable an alert definition once the problem condition in the system is resolved.

7.3 Dampening

RHQ supports the possibility of "dampening" rules. By default, each time the condition set is met an alert will fire. Dampening rules is a flexible way of changing this semantic

to suppress some of these firings, specifying, for instance, to fire an alert only if its condition is set at least x times within a given time frame.



This provides a nice way to ignore false positives caused by, say, momentary spikes in metrics. In this case, the problem metric would have to remain problematic for an extended period of time before the administrators are notified of the issue.

8 Setting up the WA prototype

In this section we describe the content of the package and the necessary steps to compile and run all the modules belonging to the WA prototype.

8.1 Structure and Content of the Package

The content of the package is structured as follows

Listing 2: Example R listing to produce the graph shown in Figure 13

```
infinispan_test
  |_test_infinispan.jar
wpm
  |_config
  |_lib
  |  log
  |_src
    |_eu.reservoir.monitoring
    |_eu.cloudtm
       |_resources
       |_rmi.statistics
       |_wpm
          |_consumer
          |_hw_probe
          |_logService
          |_main
          |_parser
          |_producer
          |_sw_probe
wpm-rhq-plugin
  |_src
    |_main
      |_java
         |_eu.cloudtm.wpm.rhq.platform
         |_eu.cloudtm.wpm.rhq.cpu
         |_eu.cloudtm.wpm.rhq.fs
         |_eu.cloudtm.wpm.rhq.net
         |_eu.cloudtm.wpm.rhq.infinispan
         |_eu.cloudtm.wpm.rhq.manager
         |_eu.cloudtm.wpm.rhq.registry
      |_resources
doc
  |_D3.2-CompanionDocument.pdf
```

- The *infinispan_test* folder contains a simple test application that uses an Infinispan cache. The used version on Infinispan exposes some relevant Key Performance Indicators (see Tables 1, 2) via the MBean sever, which are acquired by WA via WPM.
- The *wpm* folder contains the WPM system's source code, scripts and configuration files; the WPM system is composed by three modules
  - Log Service: this module logs the collected statistics within an Infinispan cache that is used for distributing data to RHQ server via an RHQ agent

plugin. The Log Service configuration file is *config/log_service.config*, and it also contains the name of the Infinispan configuration file.

  - Consumer: its configuration file is *config/resource_consumer.config*.
  - Producer: its configuration file is *config/resource_controller.config*.

- The *wpm-rhq-plugin* folder contains the source code of the RHQ plugin and includes the software components and the file descriptor used to integrate the WPM into the RHQ platfom. In particular, the plugin components, contained in the *java* subfolder, are connectors to the resources monitored by WPM and there is a component for each specific monitored resource type, i.e. platform, cpu, filesystem, network interface, infinispan cache. On the other hand the file descriptor, contained in the *resources* subfolder, specifies the type of resources supported by the plugin, the relationship between resource types and a definition of what metrics can be collected for each resource type. In addition, the plugin defines:

  - a *manager* that provides and manages the connection to the WPM for the other plugin components;
  - a *registry* module that provides an interface to the Cloud-TM global registry in order to discover all the monitored resources.

- The *doc* folder contains a textual document concerning the content of the package.

8.2 Compile the WPM prototype

1. The compile process requires the ANT program installed on the system. Decompress the zip file and, using the command line, locate the control within the folder *wpm*
2. In this folder run the command to clean the (possibly) previous project builds: *ant clean*
3. In order to compile the application, run the command: *ant compile*. The results of the execution should be the creation of the build folder that contains all the .class of the application.
4. The run scripts require the generation of an executable jar file. To do that run the command *ant jar*. If success, in the *wpm* folder a new jar file, called *wpm.jar*, should be appeared.
5. Now the application is compiled and ready to execute.

8.3 Setting up the WPM prototype

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

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

3. The command to run the application is:

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

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

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

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

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

5. The command to run the Consumer module is:

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

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

6. The command to run the Producer module is:

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

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

8.4 Compile the WPM-RHQ Plugin

1. The compile process requires the Apache Maven software. Download and install Maven as described in the Maven official web site [30].

2. Since the plugin depends on a set of JBoss software modules configure Maven in order to download JBoss artifacts in your builds as described in the ”Maven Getting Started - Users” page [31].

3. Using the command line, locate the control within the *wpm-rhq-plugin* folder and type the command *mvn install* in order to compile the plugin. If the compile process succeeds, it generates a file named *wpm-rhq-plugin-4.3.0-SNAPSHOT.jar* in the *target* folder.

8.5 Setting up the WPM-RHQ Plugin

1. WPM-RHQ Plugin is a component that runs as part of the RHQ platform. At this stage the first step includes the download and the installation of the RHQ platform as follows:

   - Download and install the PostgreSQL DBMS [32].
   - Setup a *rhq* database as described at this RHQ documentation page [33].
   - Download, install and run the RHQ Server as described at this RHQ documentation page [34].
   - Download, install and run a RHQ Agent as described at this RHQ documentation page [35].

2. Deploy the plugin as described at this RHQ documentation page [36]. The *.jar* file referenced in the docuemntation is the result of the compile process defined in Section 8.4.

References

[1] R-project, "The R-project." http://www.r-project.org.

[2] G. Box, G. Jenkins, and G. Reinsel, *Time series analysis: forecasting and control*. Wiley series in probability and statistics, John Wiley, 2008.

[3] Red Hat - JBoss, "RHQ project." http://www.rhq-project.org.

[4] A. Metwally, D. Agrawal, and A. E. Abbadi, "An integrated efficient solution for computing frequent and top-*k* elements in data streams," *ACM Trans. Database Syst.*, vol. 31, no. 3, pp. 1095–1133, 2006.

[5] Red Hat / JBoss, "JBoss Infinispan." http://www.jboss.org/infinispan, 2011.

[6] RHQ project - Red Hat, "Grouping - RHQ Documentation." http://www.rhq-project.org/display/JOPR2/Groups.

[7] Red Hat / JBoss, "Infinispan JMX statistics." http://docs.jboss.org/infinispan/5.1/apidocs/jmxComponents.html.

[8] Clearspring© Technologies, "The stream-lib library." https://github.com/clearspring/stream-lib.

[9] D. Didona, P. Romano, S. Peluso, and F. Quaglia, "Transactional auto scaler: Elastic scaling of nosql transactional data grids," Tech. Rep. 50, INESC-ID, December 2011.

[10] P. S. Yu, D. M. Dias, and S. S. Lavenberg, "On the analytical modeling of database concurrency control," *J. ACM*, vol. 40, 1993.

[11] P. D. Sanzo, B. Ciciani, F. Quaglia, and P. Romano, "Analytical modelling of commit-time-locking algorithms for software transactional memories," in *Proc. 35th International Computer Measurement Group Conference (CMG)*, 2010.

[12] L. Kleinrock, *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.

[13] Y. C. Tay, N. Goodman, and R. Suri, "Locking performance in centralized databases," *ACM Trans. Database Syst.*, vol. 10, 1985.

[14] TPC Council, "TPC-C Benchmark." http://www.tpc.org/tpcc, 2011.

[15] Red Hat / JBoss, "Radargun." http://sourceforge.net/apps/trac/radargun/wiki/WikiStart, 2011.

[16] D. Konig, V. Schmidt, and E. A. Van Doorn, "On the pasta property and a further relationship between customer and time averages in stationary queueing systems," *Communications in Statistics. Stochastic Models*, vol. 5, no. 2, pp. 261–272, 1989.

[17] . W. D. Schunn, C. D., "Evaluating goodness-of-fit in comparison of models to data," *W. Tack (Ed.), Psychologie der Kognition: Reden and Vortrge anlsslich der Emeritierung von Werner Tack*.

[18] H. Shimazaki and S. Shinomoto, "A method for selecting the bin size of a time histogram," *Neural Computation*, vol. 19, no. 6, pp. 1503–1527, 2007.

[19] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, pp. 37–57, March 1985.

[20] W. Maldonado, P. Marlier, P. Felber, J. L. Lawall, G. Muller, and E. Riviere, "Deadline-aware scheduling for software transactional memory," in *DSN*, pp. 257–268, 2011.

[21] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985.

[22] D. Crockford, "Request for Comments 4627: The application/json Media Type for JavaScript Object Notation (JSON)." `http://www.ietf.org/rfc/rfc4627.txt?number=4627`.

[23] J. R. Quinlan, "Cubist." `http://www.rulequest.com/cubist-info.html`.

[24] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[25] E. Frank, M. A. Hall, G. Holmes, R. Kirkby, B. Pfahringer, and I. H. Witten, *Weka: A machine learning workbench for data mining.*, pp. 1305–1314. Berlin: Springer, 2005.

[26] S. Haykin, *Neural Networks: A Comprehensive Foundation*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1994.

[27] S. K. Shevade, S. S. Keerthi, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to the smo algorithm for SVM regression," *IEEE-NN*, vol. 11, September 2000.

[28] R. Xu and Ii, "Survey of clustering algorithms," vol. 16, pp. 645–678, May 2005.

[29] RHQ project - Red Hat, "Alerts - RHQ Documentation." `http://www.rhq-project.org/display/JOPR2/Alerts`.

[30] Apache Software Foundation, "Apache Maven Project, howpublished = `http://maven.apache.org/`."

[31] Red Hat / JBoss, "Maven Getting Started - Users, howpublished = `http://community.jboss.org/docs/15169`."

[32] PostgreSQL Global Develpment Group, "PostgreSQL, howpublished = `http://www.postgresql.org/`."

[33] Red Hat / JBoss, "PostgreSQL - RHQ User Documentation, howpublished = http://rhq-project.org/display/jopr2/postgresql."

[34] Red Hat / JBoss, "RHQ Server Installation - RHQ User Documentation, howpublished = http://rhq-project.org/display/jopr2/rhq+server+installation."

[35] Red Hat / JBoss, "RHQ Agent Installation - RHQ User Documentation, howpublished = http://rhq-project.org/display/jopr2/rhq+agent+installation."

[36] Red Hat / JBoss, "Adding and Updating Agent Plugins - RHQ User Documentation, howpublished = http://rhq-project.org/display/jopr2/adding+and+updating+agent+plugins."