
Goal-Oriented Self-management of In-memory Distributed Data Grid Platforms

Liliana Rosa
INESC-ID/IST-UTL
lrosa@gsd.inesc-id.pt

Luís Rodrigues
INESC-ID/IST-UTL
ler@ist.utl.pt

Antónia Lopes
University of Lisbon
mal@di.fc.ul.pt

August 2011
Abstract

This paper addresses the self-management of in-memory distributed data grid platforms. A growing number of applications rely in these platforms to speed up access to large sets of data. However, they are complex to manage due to the diversity of configuration and load profiles. The proposed approach employs an adaptation policy expressed in terms of high-level goals to facilitate the task of the system manager, and address the complexity issues posed by the management of multiple configurations. The approach is validated experimentally using the open-source RedHat’s Infinispan platform.
Goal-Oriented Self-management of In-memory Distributed Data Grid Platforms

Liliana Rosa
INESC-ID, Instituto Superior Técnico, Universidade Técnica de Lisboa
Email: lrosa@gsd.inesc-id.pt

Luís Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade Técnica de Lisboa
Email: ler@ist.utl.pt

Antónia Lopes
LASIGE, Faculdade de Ciências, Universidade de Lisboa
Email: mal@di.fc.ul.pt

August 28, 2011

Abstract

This paper addresses the self-management of in-memory distributed data grid platforms. A growing number of applications rely in these platforms to speed up access to large sets of data. However, they are complex to manage due to the diversity of configuration and load profiles. The proposed approach employs an adaptation policy expressed in terms of high-level goals to facilitate the task of the system manager, and address the complexity issues posed by the management of multiple configurations. The approach is validated experimentally using the open-source RedHat’s Infinispan platform.

1 Introduction

Today, many services such as Twitter, Flickr, Yahoo, Slashdot, Facebook, and Wikipedia, among many others, rely on in-memory distributed data grids to substantially speed up their websites. Distributed data grids supply applications with a scalable storage repository where data can be accessed without bottlenecks and shared across a pool of virtual servers. Platforms such as Memcached [1], Infinispan [2], Coherence [3], Scale Out [4], or Velocity [5], provide...
fast access to data to the applications, decoupling the management of persistence from the critical request processing path. By dramatically improving the deployment of scalable applications, distributed data grids play a key role in cloud-based infrastructures.

These systems, that we will refer to as IMDDGs from now on, are able to adapt their operation in response to changes in the workload. In fact, on-demand elasticity is one of the main features of any middleware platform for cloud computing. These platforms offer several configuration options that can be adapted with significative impact on the system performance. The scaling out process is an adaptation that allows to respond to sudden changes in demand, for instance, flash crowds. This adaptation adds or removes resources as necessary, providing the elasticity so necessary in cloud computing. The elasticity can be obtained using many aspects such as cache size, number of cache cluster nodes, number of copies of each object, underlying communication protocols, eviction algorithms, the locking and deadlock detection schemes, just to name a few.

We propose an approach that allows to autonomously manage the many configurable settings of the platform and other associated middleware. The approach relies in high-level goal policies to guide the self-management of the settings, tied or not to the platform implementation. The policy is defined by a system manager, that is aware of the application needs, while the configurations are provided by the platform developers, experienced with its behavior. We are particularly interested in the aspects related with data replication, as we consider that fault-tolerance is also a fundamental aspect to manage in cloud-computing platforms.

The contribution of this work is an approach for the self-management of IMDDGs which provides a clear separation of the functions of the system manager, and platform and middleware developers. This separation allows to offer the adequate abstraction level for each, while providing an effective manner to tackle the complexity of self-management. This contribution is validated experimentally using a prototype of the approach that relies in Infinispan, a widely used open-source data grid developed by RedHat. The case study also allowed to identify which add-ons Infinispan is lacking, that would allow to take full advantage of the many configuration options and the proposed approach.

The paper continues with the motivation for the approach in Section 2. The approach itself is addressed in Section 3.1. The case study and the evaluation results are presented in Sections 4 and 5. The related work is discussed in Section 6.

2 IMDDGs

IMDDG platforms are rich in configurable settings, resulting in a wide variety of system behaviors, that influence significantly the system performance. Thus, the management of so many behaviors becomes overwhelming when the system is subject to unpredictable and variable load. In this section, we describe and discuss some of the relevant configurable settings of IMDDGs, and the useful
metrics to assess the performance of the resulting behavior.

2.1 Key Performance Indicators

In the following, we list some of the key performance indicators (KPIs) for IMDDGs. We do not aim at providing an exhaustive list of all possible performance aspects of this class of middleware. Such exercise is outside the scope of this paper. Instead, we just list the KPIs that are used in our experimental evaluation.

2.1.1 Service Ratio (SR)

Let $R_I$ be the rate of incoming (read and write) requests from the application to the IMDDG, i.e., the current load on the system. Let $R_S$ the rate at which these requests are served, i.e., the throughput. The service ratio is defined as $SR = R_S / R_I$. Ideally, the service ratio should be 1, otherwise, requests start to be queued and, eventually, the application needs to be blocked or requests dropped.

2.1.2 Service Latency (L)

The latency is defined as the average time needed to process a request. It is obtained as the weighted average of the read and write latencies, the time required to process read and write requests respectively. Typically, writes are slower than reads given that multiple copies of the data may need to be updated and/or invalidated in response to a write request.

2.1.3 Abort Ratio (AR)

IMDDGs may offer different consistency guarantees, from stronger serializability to snapshot isolation, or, weaker consistency guarantees. In any case, a common techniques to achieve consistency is to rely on some form of locking, that may cause one or more requests to abort, in particular if deadlocks occur. The abort ratio is defined as the ratio between the number of write requests that abort and the total number of write requests served. Deadlocks are particularly pernicious for performance, as locked entries remain unavailable until the deadlock is detected.

2.1.4 Resource Consumption (C)

This metric captures how many resources are used to maintain the cache operational. As noted before, it is important that the IMDDG is able to scale elastically with demand to save resources.
2.2 Reconfiguration Mechanisms

Implementations of IMDDGs have several mechanisms and deployment options that may be reconfigured in runtime, providing different behaviors whose careful selection may help improve the performance in face of evolving workloads. In the following we enumerate the configuration options used in our prototype to control the KPIs presented before.

2.2.1 Number of Servers

Most systems can be configured to use a varying number of servers. More servers may support additional load (increasing the service ratio, when it is below 1) but they consume more resources. Therefore, it is of interest to only have the strictly needed servers.

2.2.2 Replication Degree

The replication degree defines how many copies (replicas) of each data item are maintained at the global cache. In full replication mode, there is one replica per server cache. Replication serves a dual purpose: it provides fault-tolerance and it allows for additional concurrency for read requests. Unfortunately, the larger the number of replicas the more expensive are update operations. Therefore, the ideal number of replicas depends also on the read/write ratio.

2.2.3 Replica Update Protocol

To ensure fault tolerance more than one copy of data is maintained, therefore, an update protocol is necessary. Many implementations rely on some form of lock-based two-phase commit protocol to implement the update. Furthermore, a multicast protocol (with different ordering and reliability properties) may be used to communicate with the replicas (for instance, to acquire the locks). In this context, several trade-offs may need to be considered. The use of a totally ordered (TO) guarantee may prevent the occurrence of deadlocks, a phenomena that is known to plague non TO-based systems in face of contention [6]. Unfortunately, the TO guarantee is expensive in terms of communication steps and may increase the service latency. Therefore, in face of low contention workloads, it may be preferable to use cheaper communication primitives and risk a few deadlocks.

3 Self-management of IMDDGs

The aim of the proposed approach is to automate the management of the configurable settings of IMDDGs, where the adaptations are selected and deployed automatically in response to runtime changes. This allows to handle the complex task of manually determining which system configuration is the more appropriated for the current execution conditions. The approach’s planning phase
targets the complex trade-offs identified in the previous section, making a clear distinction between guiding the system management and achieving it. This distinction allows the system manager to focus on the higher-level management of the system and benefits from the expertise of platform developers in terms of platform configuration and its impact on the performance and system behavior.

3.1 Architecture

The approach relies on an external controller that follows a control loop model [7, 8], where the main activities performed by the control layer are i) the collection of relevant data from sensors; ii) the analysis of the collected data; iii) the decision on how to adapt the system to reach a desirable state; and iv) the implementation of the decision via the available effectors. Figure 1 depicts the architecture of the external controller. The main activities around the control loop are carried out by three components: Monitor, Planner, and Executor. The Monitor is independent of the IMDDG platform and collects data captured by several sensors present in each node of the cache cluster. The Monitor also analyzes this data to achieve the KPIs and detects when the system is in an undesirable state. When such a state is identified, the Monitor notifies the Planner, triggering an event carrying the relevant state information. In reaction to such notifications, the Planner determines how to adapt the system and, once a decision is made, it passes this decision to the Executor. The Executor controls the adaptation process, relying in a number of effectors, which exist at every node and implement the reconfiguration mechanisms described in the previous section.

Figure 1: Architecture for the self-management of IMDDGs

There are three distinct sensors. An application sensor that measures the in-
coming requests rate \((RI)\) to which the platform is subject to. A platform sensor that collects the throughput \((RS)\), the service latency \((L)\), and the number of aborted requests. Finally, in each node there is also an operating system sensor that captures different elements of resource consumption \((C)\): CPU, memory, network bandwidth, and energy consumption.

### 3.2 Planning based on Goal Policies

The proposed approach employs high-level goal policies \([9, 10, 11]\) to define the behavior goals for the system. This choice offers several advantages. One is that it allows to express the management guidelines independently of the platform used. Additionally, the goals can be changed without affecting the remainder of the controller. Another advantage is that it allows to explore the many trade-offs of the reconfigurable options in terms of performance. Also, it also allows to express behavior goals not related with performance; this is useful to express the self-* properties of the autonomic system, such as the self-healing property, by maintaining a level of redundancy to survive failures and address failovers. Finally, this approach is flexible enough to change the management guidelines without changing the adaptations or the effectors.

While there are other approaches that also employ goal policies to achieve an autonomic behavior \([9, 10]\), they do not address distributed systems. IMDDGs not only are distributed but their adaptation depends on the system topology, making the distribution an unavoidable aspect. This is also the case \([11]\), which led us to evolve the approach to support IMDDGs. Among other features, the new model introduces the notion of scopes for monitoring and actuation in a distributed setting, including system, component, node and instance-level scopes. The set of adaptations has also been expanded to consider new adaptations and targets. Finally, the adaptation selection process was revised, in order to take into account the new model, adaptations, and other aspects associated with the distribution.

Figure 2 presents an overview of the planning phase based on goal policies. The planning encompasses an offline and an online phase. During the offline phase, a set of adaptation rules is generated from a goal policy and a specification of the available adaptations. Each rule defines a collection of adaptations that might correct a particular deviation in the system behavior. The rule generation process itself is executed in two steps. The first step determines the types of deviations that may affect the system behavior. The second step determines the adaptations that might help to correct each deviation. The online phase is executed at runtime when the system deviates from the desired behavior, signaled by the triggering of an event. In this phase, the rule triggered by that event is evaluated against the current system state and the policy. One or more adaptations are selected as a result. During the rule evaluation, the adaptations that can return the system to a desired state are identified. To achieve that, the set of adaptations that is optimal is selected, with respect to a degradation of goals based on their rank. This means that, when it is not possible to fulfill all goals, the rule evaluation process will enforce that goals with lower rank
are violated first. For instance, in a situation with two goals – one limiting the resource consumption and another setting a minimum for the redundancy level (number of servers), if it is not possible to maintain the number of servers without surpassing the maximum established for the resource consumption, the satisfaction of the goal with highest rank will be enforced.

3.2.1 KPIs

The KPIs allow to characterize the system behavior and assess its state. Their specification of a KPI includes a name, the value type, a function defining how the global value is calculated from local values, and the acceptable error margin ($\text{Error}$) in any evaluation of the KPI (any two values are equivalent if the distance is below the margin). Four types of KPIs can be defined: system, node, component and instance-sensed. The values of system-sensed KPIs are measured for the entire system as a whole. For instance, the number of active servers in the system. The values of node-sensed KPIs are measured by individual node and its specification includes an aggregation function $AF$. This function defines how the value of the KPI for the entire system is obtained by combining the values measured in each node of the system. For instance, the power consumption can be measured per node. The values of component-sensed KPIs are measured by individual component as a whole (even if they are distributed over different nodes) and its specification includes a combination function $CF$. This function defines how the KPI value for the entire system is obtained by combining the values measured for all the components. For instance, the latency of requests may be measured by component. Finally, the values of instance-sensed KPIs are measured per component instance, i.e., by individual component in each node. The specification includes a combination function and an aggregation function that are employed to calculate the value of the global KPI from sets of values.

![Figure 2: Overview of goal based planning](image-url)

7
measured per component instance. The functions can be applied in any order. Some examples of KPIs follow:

- **KPI - System**
  - number_servers: int Error 0
- **KPI - Component** latency: double CF Avg Error 0.1
- **KPI - Instance** throughput: double CF Sum AF Sum Error 0.2

The service ratio \( (SR) \) is an example of a CKPI whose values are calculated dividing the request throughput \( (RS) \) by the rate of incoming requests \( (RI) \).

### 3.2.2 Components and Adaptations

The component specification includes the description of all the components available for use in the system. In particular, this specification defines a type hierarchy organizing components in types according to the functionality they provide. A component type is either concrete or abstract. The first designates a specific type for which an implementation is available, while the second represents the characteristics of a group of components. The component specification also includes information regarding which KPIs are measured for a component. The following example describes an IMDDG platform, namely Infinispan, as one of the system components.

Component **Infinispan**

- **subtype** IMDDG, \{throughput, latency\} Movable
- **GlobalParameters**
  - updateProtocol: \{TO, noTO\}

The adaptation specification describes the adaptations that can be used to manage the system. Adaptations are defined in terms of a fixed set of actions. To address both distributed and non-distributed components, three groups of adaptations were considered, characterized by their scope: component, instance, and node. Component adaptations target a component \( c \), affecting all the instances in the system. Instance adaptations only affect an instance of the component \( c \) in a particular node \( n \). Finally, node adaptations affect only a node \( n \). The type of adaptation and the accepted actions are depicted in Table 1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>c.setParameter(param,value)</td>
</tr>
<tr>
<td></td>
<td>c.replaceBy(c')</td>
</tr>
<tr>
<td>Instance</td>
<td>n.c.setParameter(param,value)</td>
</tr>
<tr>
<td></td>
<td>n.c.replaceBy(c')</td>
</tr>
<tr>
<td>Node</td>
<td>n.addComponent(c)</td>
</tr>
<tr>
<td></td>
<td>n.removeComponent(c)</td>
</tr>
</tbody>
</table>

Table 1: Adaptation actions per type of adaptation

Each adaptation also declares the impact of the declared actions on the system KPIs and CKPIs and the time for stabilization. The impact is an estimate of the values of affected KPIs after the adaptation. The impacts may be declared for the global value of a KPI (even if it is not a system-sensed KPI),
or regarding the contribution of the affected target only, referring to the component, node or instance-sensed KPI. The definition of an impact in a KPI may employ the value of itself, other KPIs, or monitored context information. Examples of adaptations are provided in Section 4.

The adaptation specification may also list dependencies and explicit conflicts between pairs of adaptations, to force or prevent that these adaptations are executed at the same time. Additionally, some conflicts are also imposed by our approach, hereafter referred to as implicit conflicts. An implicit conflict occurs when, from the information of the adaptations in the specification, it is not possible to determine the impact of the combination. These conflicts are described in Section 3.2.4.

### 3.2.3 Goal Policy

The goals of a policy are the high-level directives that will guide the system management. These goals describe what is the acceptable behavior in terms of KPIs values. There are three types of exact goals and another three of approximation goals. The exact goals separate the values of a KPI in two disjoint sets: acceptable and not acceptable. An above goal will only find acceptable the values above the threshold. A below goal will only accept the values below, and a between goal only the values in the specified interval. The approximation goals are best effort goals that specify a total order between the values of a KPI. A maximize goal states that the largest is the best, while a minimize goal aims at the smallest. A close goal tries to keep the value as close as possible to the aim value. The system will try to optimize its behavior with respect to approximation goals periodically (every period of time) but there is a minimum gain for an adaptation be worthwhile. The six types of goals follow:

<table>
<thead>
<tr>
<th>Goal</th>
<th>goalName</th>
<th>kpiName</th>
<th>action</th>
<th>condition</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>goalName</td>
<td>Above</td>
<td>threshold_down</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goal</td>
<td>goalName</td>
<td>Below</td>
<td>threshold_up</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goal</td>
<td>goalName</td>
<td>Between</td>
<td>threshold_down</td>
<td>threshold_up</td>
<td></td>
</tr>
<tr>
<td>Goal</td>
<td>goalName</td>
<td>Minimize</td>
<td>kpiName</td>
<td>MinGain</td>
<td>value</td>
</tr>
<tr>
<td>Goal</td>
<td>goalName</td>
<td>Maximize</td>
<td>kpiName</td>
<td>MinGain</td>
<td>value</td>
</tr>
</tbody>
</table>

### 3.2.4 Rule Generation and Evaluation

The rule generation process consists in two steps that aim to generate the rules that will be evaluated to select adaptations. A rule is composed by an event and a set of combinations of adaptations. Its generation starts by extracting the relevant events from the goals. Next, for each event, the set of possible useful adaptations is selected, i.e., those that help correcting the problem signaled by the event. For instance, assuming that we have an event that demands to lower the KPI (because it is above the limit), all the adaptations that decrease the KPI are selected. The adaptations whose helpfulness is unknown are also selected. Then, all combinations of adaptations that can be performed simultaneously are calculated.
Contrary to the work in [11], the identification of the valid combinations of adaptations needs to consider distribution. The following combinations have implicit conflicts: i) instance adaptations that affect the same component, the same node or both, and have impact over a common KPI; ii) component or instance adaptations that affect the same component and have impact over a common component-sensed or system-sensed KPI; iii) node or instance adaptations that affect the same node and have impact over a common node-sensed or system-sensed KPI; iv) component and instance adaptations that have impact on different components but on the same node-sensed KPI (the first on the global value, while the second on the local node value); or v) component and node adaptations that have impact on the same KPI.

The rule evaluation process analyzes one rule at a time, using the system state as input. It starts by estimating the impact of the adaptations in the KPIs. Afterwards, the selection algorithm evaluates the different combinations of adaptations against the goal with the highest rank. If a combination fulfills the goal, it passes to the next step, otherwise, it is discarded. If none of the combinations fulfills the goal, they all pass to the next step, as the KPIs associated with lower ranked goals may be still corrected or improved. After, the combinations that passed are evaluated against the next goal, and so forth until all the goals are evaluated. Among the combinations that reach the last step, one is selected to be performed.

4 Case Study

The case study developed to validate the proposed approach relies on a website, that serves a majority of dynamic content. The website is served by several web servers, each running on its own node. The website load is balanced by the several web servers. The system relies on an IMDDG to speed up web servers’ access to data, with a local instance serving each of system web servers. The local instance has its own cache that serves as a proxy to query and update remote caches of the platform. The website content is generated from data that is available in the cache of the IMDDG platform. If the data is not available in cache (local or remote), it will be requested to the database and added to cache. Besides the web server and the IMDDG instance, each node also includes a communication toolkit that provides communication and coordination support between IMDDG instances. In our prototype we use the Infinispan [2] as an IMDDG platform. The communication toolkit used by Infinispan is JGroups [12]. Figure 3 illustrates the case study architecture. The management of the database is not considered in this case study.

The case study’s self-management aims at taking advantage of the reconfigurability of the IMDDG platform and associated communication toolkit. The goal is to optimize performance in order to maintain user satisfaction, but at the same time to minimize resource consumption to cut on costs. Furthermore, and as mentioned before, we are also concerned with fault-tolerance and healing properties. The development and evaluation of the case study will focus on
4.1 KPIs and Goal Policy

The following objectives guide the self-management. One is to self-optimize performance, balancing the different trade-offs in terms of service ratio, service latency, and abort ratio. The aim is to have the highest possible service ratio, and the lowest service latency and abort ratio, guaranteeing that user satisfaction is the best possible. Another objective is to be energy-efficient, namely, by reducing the energy consumption required to maintain the system, thus, cutting on costs. Finally, the last objective is fault-tolerance, providing self-healing and self-protection features to the system, so that it can tolerate server failures without loss of data.

To fulfill these objectives, the first step in our approach is to characterize the system behavior, selecting the adequate KPIs. These KPIs will be used by the system manager to describe the goal policy and by the platform developers to specify the adaptations. The case study KPIs are depicted in Table 2. All the KPIs were already described in Section 2.1, with the exception of $\#servers$ that describes how many instances of the IMDDG/servers are active.

Next, it is necessary to translate the informal objectives to goals. The goal policy chosen for the case study is one among several, as it is possible to derive different policies that give distinct priorities to different objectives. The policy consists in five goals. The first is to maintain the redundancy, i.e., a minimum number of servers/replicas. This self-healing property is the most important goal because it will allow the system to recover from fail overs and avoid downgrading the service to a critical level. The next three goals address performance and...
user satisfaction issues. The second goal limits the abort ratio because, when it becomes higher than the 0.008 threshold, the system becomes irresponsive, a consequence of being blocked most of the time (the threshold was determined experimentally, through benchmarks). In the third goal, the system attempts to process as many requests as possible, to maximize the service ratio. The fourth goal is to minimize the latency. Finally, the last goal minimizes the resource consumption, if other goals are not violated, to cut on costs.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Description</th>
<th>MinGain</th>
<th>Every</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preserve Redundancy</td>
<td>#servers Above 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Limit Abort Ratio</td>
<td>AR Below 0.008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max Service Ratio</td>
<td>Maximize SR</td>
<td>0.05</td>
<td>300</td>
</tr>
<tr>
<td>Min Latency</td>
<td>Minimize AL</td>
<td>0.05</td>
<td>400</td>
</tr>
<tr>
<td>Min Cost Resources</td>
<td>Minimize #servers</td>
<td></td>
<td>500</td>
</tr>
</tbody>
</table>

4.2 Adaptations

The platform developers have knowledge of the different settings used to configure the platform and their impacts in the behavior. In the case study, we focus on two settings: the replication degree and the replica update protocol. Regarding the first, while Infinispan can handle changes in the number of local instances running, it does not offer any support to adapt it during runtime. For instance, it does not adapt the #servers to address changes in the workload. As already mentioned, by adding a new server/replica we are able to process more incoming requests (improve the service ratio), but at the same time, write requests will take longer to be performed and cost increases. In terms of replica update protocol, Infinispan uses a fixed configuration defined before runtime. The adaptation of the configuration may yield improvements in performance, namely, through the (de)activation of the total order guarantee for the communication. As already discussed, this allows to reduce the abort ratio at the expense of increasing the request latency. From the four adaptations possible, we only present two due to space constrictions.

Reversible Component Adaptation Activate Total Order

Component:
JGroups
Actions:
JGroups.setParameter(totalOrder, on)
Requires:
JGroups.totalOrder = = off
Impacts:
AL += 1
AR ÷= 3.33
RS *= (11 – writePercentage * log(1.2 * #servers))
Stabilization:
period = 60 secs

Reversable NodeAdaptation AddServer(n)

Actions:
 n . addComponent(ApacheHTTP)
n . addComponent(Infinispan)
n . addComponent(JGroups)
Requires:
! n . hasComponent(ApacheHTTP)

Impacts:
#servers += 1
RS = (writePercentage * (1 – AR) * writeTime)^-1
Stabilization:
period = 120 secs

The first adaptation activates the total order guarantee in all instances of the JGroups component, increasing its contribution to the service latency, decreasing the abort ratio, and with impact on throughput. The second adaptation allows to add a new server/replica, in an inactive node. This adaptation is parametrized by a node, thus, it will originate as many adaptations as the number of servers (ten). This is achieved by adding all the necessary components to the node: the web server, Infinispan, and JGroups. As a result, the #servers KPI increases and has impact on RS. The impact on RS is specified using the average write time, computed by the monitor as described in [13]. While this adaptation may affect the latency, the impact is negligible (compared with the first adaptation), thus, we do not consider it.

The impact functions of the available adaptations in RS are based on results obtained from distinct benchmarks made to the system, where different combinations of the write percentage, the key pool size, and the number of servers were explored. The impact functions used are not exact, but they do provide enough accuracy for the approach.

4.3 Generated Rules

With the information provided by the human operations, the proposed approach is now able to generate the rules that will be used to manage the system. These rules are composed by an event, extracted from the goal, and the suitable combinations of adaptations to address the issue described by the event. The extracted events are described in Table 3.

Before generating the adaptation rules, the specified adaptations need to be unfold into the final set of adaptations. For reversible adaptations this involves finding the inverse adaptation, while for node adaptations it demands to instantiate the adaptation for each node. The impacts of an adaptation are used to assess if it contributes to satisfy the target goal. An example rule follows:

When kpiDecrease(#servers, .500, true)
<table>
<thead>
<tr>
<th>Type</th>
<th>Goal</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>maintain_redundancy</td>
<td>kpiBelow(#servers,3)</td>
</tr>
<tr>
<td>Exact</td>
<td>limit_abort_ratio</td>
<td>kpiAbove(AR,0.008 + 0.0001)</td>
</tr>
<tr>
<td>Approx</td>
<td>maximize_SR</td>
<td>kpiIncrease(SR,300,true)</td>
</tr>
<tr>
<td>Approx</td>
<td>minimize_latency</td>
<td>kpiDecrease(AL,400,true)</td>
</tr>
<tr>
<td>Approx</td>
<td>minimize_cost_resources</td>
<td>kpiDecrease(#servers,500,true)</td>
</tr>
</tbody>
</table>

Table 3: Events extracted from the goals

5 Experimental Evaluation

The prototype relies in Infinispan 5.0.0, whose local instances were deployed using the replication topology for fault-tolerance purposes and the default configuration included in Infinispan. JGroups version 2.11.0 was used, which includes a mechanism to seamlessly activate and deactivate the total order layer. We used the default communication stack with UDP and flow control, as well as the sequencer protocol for total order. The load imposed by the web servers is emulated by the benchmark Radargun 1.1.0 [14]. The benchmark simulates the clients, the virtual server load balancer, and the web servers at each node. The benchmark detects when a new server is added to the cluster, through a monitoring agent present in each node, which notifies the virtual server’s load balancer. Radargun emulates the operation of the web server, using a configurable load profile, that includes the write percentage and the object pool. The benchmark was extended to allow a finer control of the workload and the duration of each experiment. The autonomic controller was implemented in the Java™ language. The testbed consists of eleven machines. One hosts the autonomic controller and the remaining machines can run an instance of the Infinispan/JGroups (up to 10 instances total). Each machine is a Dual Intel Xeon Quad-Core, 2.13 GHz clock speed, and 8 GB of RAM running Linux (kernel 2.6.32-21-server) and are all connected by a 1 Gbps Ethernet.

5.1 Workloads

We rely in several workloads to simulate variable load. All experiments follow the same pattern: we first let the system stabilize in the best configuration for a given workload, then we change the workload characterization and observe how the system reacts. Changes to the workload are made such that different adaptations are more appropriate in each experiment. The workload transitions are based on 4 workloads described in Table 4. The workloads are differentiated by two main characteristics: high (HC) or low contention (LC), and load (RI). The HC workload captures a scenario where concurrent writes to the same item occur often, which creates many opportunities for deadlocks and increases
the abort ratio. This is achieved using a write-dominated sequence of requests accessing a small pool of objects. The LC workloads capture scenarios where potential number of aborts is small. This is achieved using a more balanced read-write ratio and a larger object pool. The difference between the three LC scenarios is the total load, where 3, 5 and 6 state the number of servers that are necessary. We have experimented 4 transitions: i) LC-3 to HC-3; ii) HC-3 to LC-3; iii) LC-5 to LC-6; and iv) LC-6 to LC-5. Each transition is discussed next.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Object Pool</th>
<th>% of Writes</th>
<th>RI</th>
</tr>
</thead>
<tbody>
<tr>
<td>HC-3</td>
<td>100</td>
<td>90</td>
<td>200</td>
</tr>
<tr>
<td>LC-3</td>
<td>5000</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>LC-5</td>
<td>5000</td>
<td>40</td>
<td>2400</td>
</tr>
<tr>
<td>LC-6</td>
<td>5000</td>
<td>40</td>
<td>2750</td>
</tr>
</tbody>
</table>

Table 4: Workloads

**LC-3 to HC-3.** The website is subject to an increase in the contention level. Before the increase, total order was inactive in JGroups to obtain lower latency. With the workload change, the abort ratio increases and worsens the service ratio, with less requests being served with success. Figure 4(a) depicts, on the first part of the plot (until the vertical line that marks the adaptation), this situation. The abort ratio rises above the desired limit, violating the corresponding goal. To avoid reacting to isolated peaks and to allow stabilization of monitoring data, the monitor only notifies the planner after successive violations of the threshold. When the corresponding event reaches the planner, the corresponding rule is evaluated and the resulting adaptation is *activateTotalOrder*. Afterwards, the abort ratio decreases and the service ratio improves, as depicted in Figure 4(a).

**HC-3 to LC-3.** This transition illustrates the inverse situation of the previous transition. The system is initially using total order due to HC. The change in the workload causes the contention to drop and the abort ratio to decrease. At some point, the periodic event corresponding to the abort ratio approximation goal is triggered and the planner determines that it is possible to improve the latency without compromising any of the higher ranked goals. The selected adaptation is *DeactivateTotalOrder* that decreases the latency while maintaining a good service ratio, as depicted in Figure 4(b). This effect is noticeable by observing the average write time.

**LC-5 to LC-6.** The website is subject to an increase in the load. Before the increase, the system is not using total order and requires 5 servers to process all incoming requests. When the workload changes, the 5 servers/replicas become overloaded and the service ratio decreases. As a result, the planner selects the adaptation *AddServer* to add another web server and Infinispan instance. Figure 4(c) depicts the improvement of the service ratio after the adaptation. However, the addition of new server increases the service cost and the resource consumption, as demonstrated by the increase in the power consumption, as
the server is no longer idle. We opted to show the average power consumption because power consumption is not steady over time.

**LC-6 to LC-5.** This transition illustrates the inverse of the previous transition, decreasing the load, and the system is using 6 servers. At some point, the periodic event corresponding to the cost and resource consumption goal is triggered and the planner determines that it is possible to cut on cost and resources, while maintaining the service ratio. The planner selects the RemoveServer adaptation. Figure 4(d) shows that after the adaptation, the service ratio is maintained, power is saved and the service becomes cheaper.

![Graphs](image)

(a) First scenario: activating the total order guarantee  
(b) Second scenario: deactivating the total order guarantee  
(c) Third scenario: adding a new instance  
(d) Fourth scenario: removing one instance

Figure 4: Experimental results some minutes before and after the adaptation
5.2 Discussion

The results show that the approach is able to automatically manage the system during runtime; it successfully satisfies the management objectives in distinct scenarios, taking advantage of the available adaptations to improve the system performance. Many works address the automatic scaling of similar systems by dynamically changing the number of nodes. As shown here, our approach also support that type of adaptation. More importantly, we also show that other, finer grain adaptations, such as replacing the communication protocol used to obtain locks, can also bring significant advantages. In the evaluation, the self-management addresses two settings whose adaptation demanded that Infinispan was extended with features to allows its reconfiguration. These are not the only settings that could benefit from our approach. For instance, when reconfiguring group communication, we were limited to change the protocol for all objects. Actually, it would be preferable to have a finer grain control and to make this adaptation only for the objects most affected by contention. Similarly, it would be interesting to be able to change the cache topology per object. Unfortunately, Infinispan, as well as other IMDDGs do not currently offer the mechanisms necessary to make these changes. Nonetheless, the goal was not to experiment with the entire set of possible adaptations, but to demonstrate that our approach is viable in this context. Furthermore, our previous work in centralized environments [11] shows that the planning can scale for larger sets of adaptations. Larger sets of adaptations dramatically increase the difficulty of manually balancing the trade-offs necessary to specify a low-level policy, not to mention more error-prone.

6 Related Work

IMDDGs employ different technologies that several works have tried to dynamically reconfigure and tune. Many works focus on the adaptation of different aspects of caching middleware. They provide adaptive solutions for the expiration time of objects in the cache [15], cache update algorithms [16, 17], or mapping of requests to groups of cache servers in web caching [18]. None of these approaches relies on high-level policies to autonomously manage the aspect they target. In [15, 16, 17, 18] the planning is hard-coded in the algorithm and only takes in consideration a limited number of metrics, such as the workload, and a specific target behavior for the system. To extend these algorithms to consider other metrics, adaptations, and a distinct desired behavior, the planning would have to change significantly. In contrast, our approach provides enough flexibility to change the system’s adaptive behavior, by changing the goal policy. Therefore, the self-management support does not require re-development. Nevertheless, these approaches could be integrated in our approach as part of the pool of cache policies.

Resource provisioning in variable workloads is another rich research area. Several works propose different approaches to elastic servers [19, 20] to cope
with variable workloads. These works are focused on how web servers handle load, increasing the number of servers when the load increases and decreasing it when traffic becomes less intensive. As the evaluation of the approach shows, this is not always the case. The effectiveness of this guideline depends also on the load profile, cache topology, among others. For instance, in a replicated topology with significant percentage of writes, adding more nodes to the cache cluster will worsen performance.

7 Final Remarks

In this paper, we present an approach for the self-management of systems employing IMDDGs, where the system administrator is required to express the target behavior of the middleware in terms of a high-level policy that establishes goals for a set of KPIs. The platform developers describe the possible adaptations, that are selected and deployed automatically, in response to runtime changes in the workload. The approach was validated using a prototype based on Infinispan. In the future, we would like to extend our work to more configuration settings and other cloud middleware, where self-managed elastic allocation of resources, such as the one illustrated in our example, is of paramount importance.

Acknowledgements

This work was partially supported by the FCT (INESC-ID+LASIGE multi annual funding through the PIDDAC Program and CMU-PT/ELE/0030/2009).

References


