Selfware Deliverable SP2

Lot 2, June 26th, 2008

Selfware Self-Optimization: algorithms, architecture and design principles

Authors

Fabienne Boyer (INRIA/Sardes)

Christophe Taton (INRIA/Sardes)

Jeremy Philippe (INRIA/Sardes)

Bruno Dillenseger (France Télécom R&D)

Version 1.0

Contents

1 Introduction

Autonomic computing, which aims at the construction of self-managing and self-adapting computer systems, has emerged as an important goal for many actors in the domain of large scale distributed environments and applications. This approach more precisely aims at providing systems and applications with self-management capabilities, including self-configuration (automatic configuration according to a specified policy), self-optimization (continuous performance monitoring), self-healing (detecting defects and failures, and taking corrective actions), and self-protection (taking preventive measures and defending against malicious attacks).

Following this approach, the Selfware project aims at providing an infrastructure for developing autonomic management software. An important aspect of this infrastructure is the adoption of an *architecture-based* control approach as described in the SP1-L1 document, meaning that the control loops that regulate the system have the ability to introspect the current software architecture of the managed system, as well as they have the ability to modify (i.e. reconfigure) this architecture.

The objective of this document is to specify the way self-optimization features will be actually provided in the context of this architecture, on the basis of the Selfware framework and tools described in document SP1-L2. By giving specific design principles, algorithms and strategies, this document gives a common basis that practically supports the implementation of Selfware's self management scenarios (SP3, SP4). Self-optimization features are first presented in the general context of Internet Services, with details about the key issues that must be taken into account (load variation profiles, stability). Then, another self-optimization architecture is described, combined with a self-regulated load injection system, for the purpose of self-benchmarking.

2 Self-Optimization of Internet Services

This section presents the way Internet services are self-optimized in the Selfware platform. We first describe the architecture and design principles of the ISS. Then we detail how the ISS manager tackles different types of load variation, and how it prevents system oscillations.

2.1 Architecture and design principles

In the Selfware platform, the Internet Service Self-optimization (ISS) Manager is responsible of applying a given self-optimization policy on an Internet service. The self-optimization policy described here is based on dynamic resource provisioning, i.e., on-line addition and removal of resources to and from an Internet service. It observes the behavior of a service and triggers resource provisioning or un-provisioning according to its observations.

This self-optimization management assumes that Internet services are deployed in the form of replicated entities, providing both scalability and high availability properties. A self-optimization manager is more precisely associated with each set of replicated entities of an Internet service. For instance in the case of an e-mail Internet service composed of a number of replicated e-mail servers, a self-optimization manager will be associated with the set of replicated e-mail servers. In a partitioned video-on-demand Internet service, a self-optimization manager will be associated with each partition of the VoD service. As for multi-tier e-commerce Internet services, a self-optimization manager will be associated with each tier of the multi-tier e-commerce web application.

The ISS manager applies a resource usage threshold-based policy to a set of Managed Elements (as defined in the SP1-L1 document [4]) that correspond to the replicated entities. When the resource usage of the underlying set of replicated entities reaches a maximum threshold, we consider that the system is over-loaded and thus the ISS manager provisions the set of managed entities with additional resources. Symmetrically, when the resource usage of the set of managed entities goes below a minimum threshold, we consider that the system is under-loaded. In this case, the ISS manager removes resources from the set of managed entities.

The ISS manager is organized as follows. It observes the behavior of a set of replicated entities and, based on a particular policy, it triggers resource provisioning or un-provisioning according to its observations. It is more precisely organized in three main parts that are described in the following sub-sections: (i) system observation, (ii) self-optimization policy, and (iii) system reorganization.

2.1.1 System observation

The ISS observation part is responsible of observing the behavior of the underlying managed system in terms of resource consumption. Resource consumption refers to hardware resources such as cpu, memory, disk or network. System observation may have the form of an on-line resource monitoring system that performs real-time monitoring of the system through probes, or it may have the form of predictions of future resource usage of the system. The former is used to implement reactive self-optimization, while the latter applies in this case some form of proactive self-optimization. On-line resource monitoring consists in resource usage indicators (i.e. sensors or probes), that can be provided by the CLIF framework [1] [5].

Self-optimization is triggered when a sensor reports a value that violates some minimum or maximum thresholds. High-level sensors may aggregate and filter many lower-level sensors to provide meaningful resource usage indications. Aggregation allows to consolidate grouped resource usage information (e.g. partition-wide resource usage as shown in figure 1).

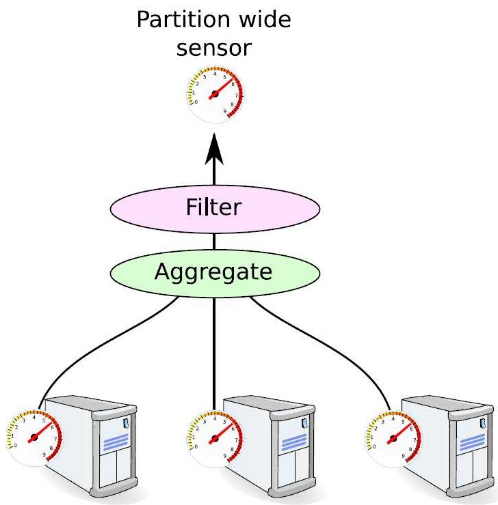

figure 1. Aggregation and filtering of sensors

Aggregation is usually achieved through mathematical computations such as summing, averaging, minimum finding, etc; this depends on the nature of the information to measure and to report. Filtering generally targets the removal of meaningless artifacts for stability purpose through smoothing over time (e.g. raw average or EWMA), flip-flop filters, etc. Filtering effects are illustrated in figure 2.

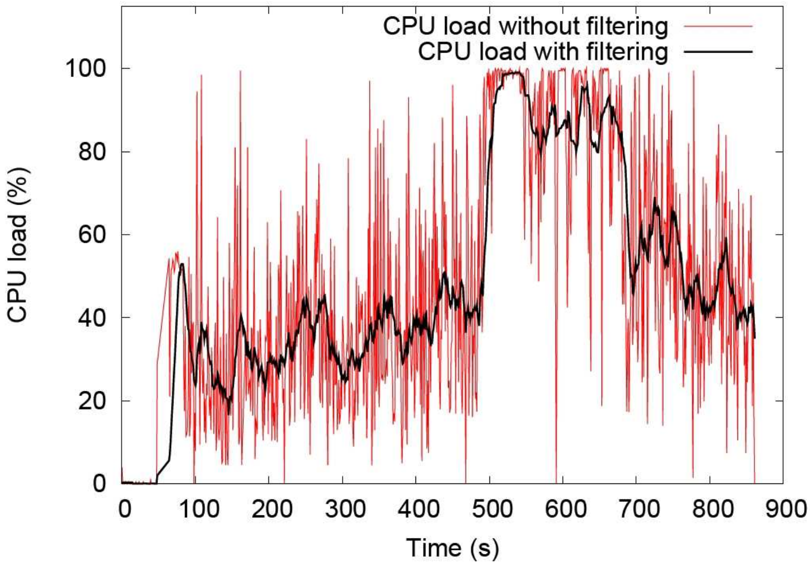

figure 2.Filtering of monitoring data

2.1.2 Self-optimization policy

The central part of the ISS manager is its policy. Its general architecture is briefly described in figure 3 and its in algorithm in figure 4. The ISS policy is a control-loop that reacts to events received from the system observation part. Each time an event is notified, it is analyzed to check if the underlying managed system is over-loaded or under-utilized.

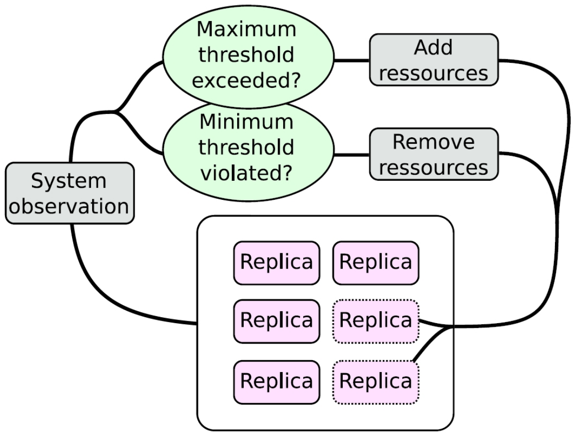

figure 3.Architecture of the ISS manager

If one of the observed resources exceeds its maximum threshold, that means that the managed system faces a bottleneck and is over-loaded. In this case the system is provisioned with additional nodes. Symmetrically, if all observed resources use less than their minimum threshold, that means that the overall system is under-utilized. Thus nodes are removed from the managed system. Addition and removal of nodes is done through architectural reconfiguration operations, the third part of the self-optimization manager.

```
on Receive(event: MonitoringEvent):
if (event.consumption_1 > max threshold_1)
        or (event.consumption_2 > max threshold_2)
        or . . .
        or (event.consumption_r > max threshold_r) then

    // System is overloaded and need more resources
    AddResourcesToSystem();

else if (event.consumption_1 < min threshold_1)
        and (event.consumption_2 < min threshold_2)
        and . . .
        and (event.consumption_r < min threshold_r) then

    // System is underloaded and wastes resources
    RemoveResourcesFromSystem();

end if
```

figure 4.Algorithm of the ISS manager

2.1.3 Architectural reconfigurations

This latter part of the ISS manager provides operations that actually perform the dynamic provisioning or un-provisioning of nodes to the managed system. Such operations consist in assigning new free nodes to the managed system, releasing nodes from the managed system, installing on a new node the software needed by the managed system when necessary, configuring the software and starting the software.

More generally, figure 5 depicts a general example of how self-optimization applies in an Internet service. An Internet service may be organized as partitioned and pipelined sub-systems, where the partitioned and pipelined entities may be sets of replicated entities (S_1 to S_7 in figure 5). In this context, an ISS manager is associated with each set of replicated entities. It is responsible of dynamically provisioning resources to that set of replicated entities. Moreover, the self-optimization managers of the Internet services co-operate in order to provide a global consistent behaviour (e.g. preventing system oscillations as discussed in the following section).

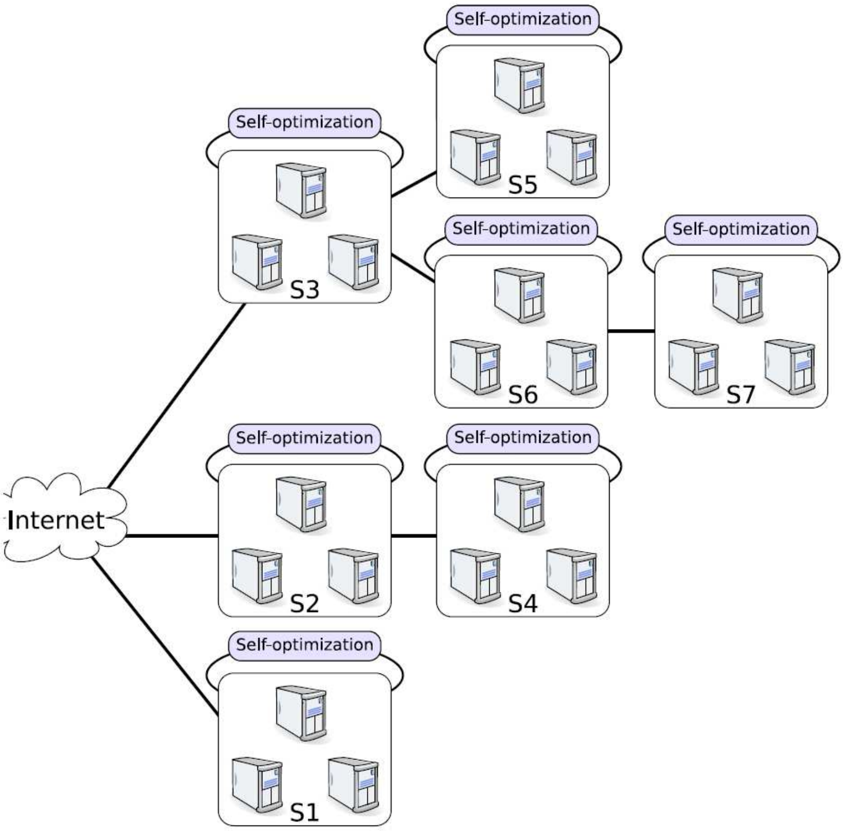

figure 5.Architecture of self-optimized Internet services

2.2 Managing load variations

Load variations may happen following different schemes. A common scenario consists in a gradual change of the load which will progressively induce an under-load or an overload in the system. Another common scenario often happens at the occasion of big events and consists in sudden load variations also commonly referred to as load spikes or flash crowds.

Whether a load variation is considered as gradual or sudden is related to the relative difference between the speed of load variation and the speed of architectural reconfigurations. Gradual load variation corresponds to load variation that happens slower than the architectural reconfigurations speed (see figure 6).

In this case, simple architectural reconfigurations such as single resource addition or removal (i.e. at the granularity of one resource at a time) are sufficient to absorb the load variation. On the contrary, sudden load variation happens when the load variation is faster than the architectural reconfigurations speed (see figure 7). In this case, fine-grain heuristics are required to determine the optimal capacity planning of the system, in order to accelerate the process of architectural reconfigurations towards an optimum state. In the case of sudden load variation, it is necessary to determine the amount of resources to (un-)provision, before actually performing the (un-)provisioning in a single step. In the following, we present two mechanisms to address both types of load variations.

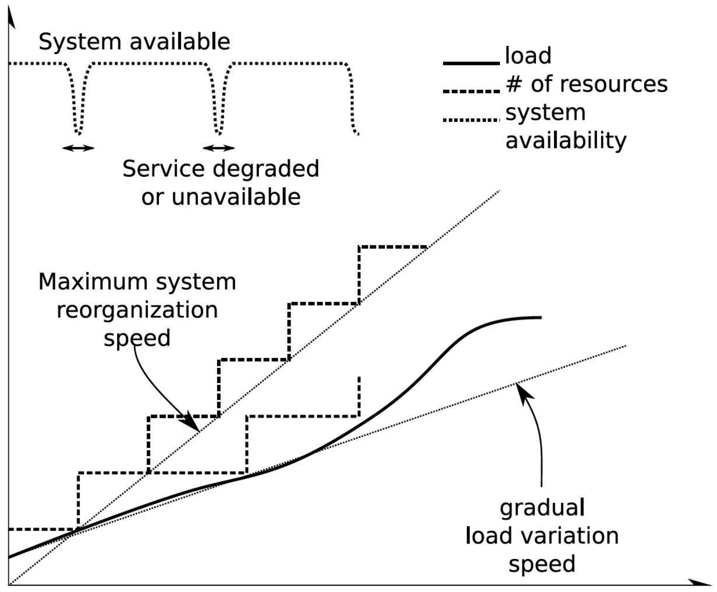

figure 6.Gradual load variation

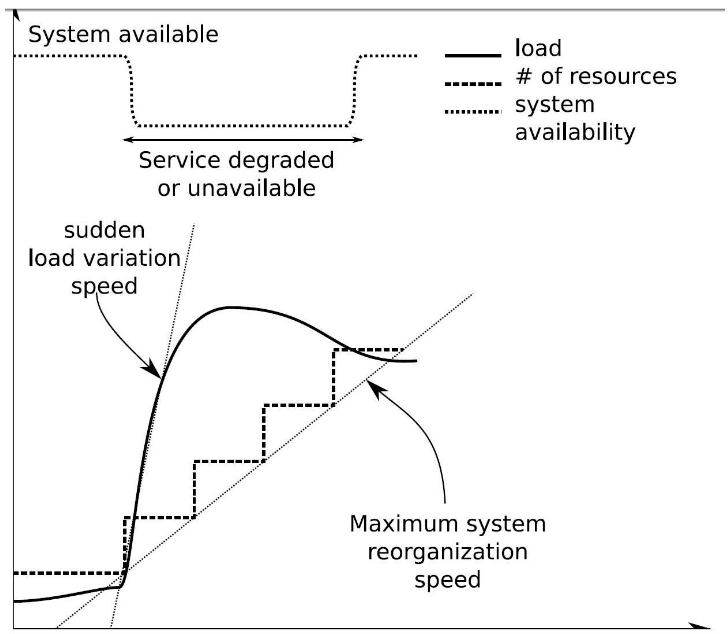

figure 7.Sudden load variation

Gradual Load Variations. In a system undergoing gradual load variations, the capacity planning of the system can be continuously adjusted through architectural reconfigurations as simple as adding or removing resource units one at a time. Indeed, the gradual load variation assumption ensures that the system provisioning will be updated promptly enough to absorb and follow the load variation. We implemented a self-optimization manager able to handle gradual load variations. The manager relies on low-level resource usage system observations (such as cpu, memory, disk or network) and, based on these observations, triggers single node addition or removal to the system. We experimented and evaluated this manager on a clustered Internet service implementing a multi-tier e-commerce web application that was submitted to gradual load variation. The Internet service was able to self-optimize its behaviour according to the changing workload.

Sudden Load Variations. In a system submitted to sudden load variations, updates to the system capacity planning may require addition or removal of multiple resources at a time, so as to absorb the load peaks. In case of sudden load variation, it is preferable to factorize and parallelize architectural reconfigurations operations, thus increasing the overall speed of system reorganization. The challenge here is to determine as accurately as possible how many nodes to add or remove in one step. We implemented a self-optimization manager that copes with sudden load variations. It is based on low-level resource system observations (cpu, memory, disk and network) as well as higher application-level observations (such as the number of concurrent transactions running in a server). Application-level observations allow the construction of heuristics functions to determine the optimum capacity planning of the system. We identified a heuristics function that calculates the optimum capacity planning as linearly proportional to concurrent transactions in the system. Based on the result of this function, multiple nodes are assigned or released in parallel. We implemented this self-optimization policy and applied it to a cluster of Internet services that implements an e-commerce web application. In the presence of load spikes, the e-commerce web application was able to efficiently self-optimize.

2.3 System oscillation management

Another issue of self-optimization is that it may introduce system instabilities during which sensors may report meaningless information. Thus, interpreting these signals is likely to be irrelevant and leads to erroneous decisions. Indeed, dynamic resource provisioning of an Internet service may induce multiple concurrent provisioning operations that are actually not necessary and would, as a result, hurt the overall Internet service performance. For instance, in a multi-tier Internet service composed of a front-end web server and a database back-end organized as a pipelined system, the database back-end might become a bottleneck and induce an underload on the front-end web server (which then waits for responses from the back-end tier).

In such a situation, the self-optimization could trigger provisioning operations, increasing the amount of resources on the database back-end tier on one hand, while reclaiming unused resources on the front-end tier. Obviously, the latter un-provisioning on the front-end tier is a consequence of the dependency between the front-end web server and the database back-end that leads to un-necessary operations, and therefore to system oscillations. To prevent system oscillations, we introduce a technique that (i) first automatically calculates inter-dependencies between sub-parts of the system, and then (ii) automatically prevents system oscillation occurrence.

The system oscillation management relies on a description of the system that allows the manager to determine dependencies between parts of the system. More precisely, the manager is given a representation of the system in terms of pipelined and partitioned sub-systems. Thanks to this knowledge, the manager infers a dependency function defined as follows: (i) a sub-system Si depends on a sub-system Sj if Si and Sj are parts of a pipelined system, and (ii) a sub-system Si always depends on itself. Indeed, a pipeline materializes the dependency between sub-systems, while a partition materializes their independency.

Notice that in pipelined sub-systems, the workload of one of the sub-systems may have a side-effect on another sub-system in pipeline. This is due to the fact that client request processing may flow through all or part of the pipelined sub-systems. While in case of partitioned sub-systems, the workload of the different sub-systems are independent from each other; each partition being responsible of processing requests independently from the other partitions. Thus, based on the inter-dependency function and on the knowledge of the system architecture, the SSIS manager is able to automatically identify inter-dependent parts of the Internet service. To prevent oscillations from occurring, the SSIS manager ensures that during a self-optimization operation on a part of the system, self-optimization is inhibited on any inter-dependent part (during a given delay). Once the inhibition delay has expired, new self-optimization operations are allowed for execution again.

We implemented the system oscillation management for a self-optimized e- commerce web application hosted by a two-tier Internet service where each tier of the Internet service was replicated and dynamically provisioned. The two tiers of the Internet service were identified as a pipelined system. Thus, all system reorganization happening on the first or the second tier of the system triggered an inhibition that blocked any new reorganization on the first and on the second tier.

3 Self-optimization for self-benchmarking

The principle of self-benchmarking has been detailed already in previous deliverable [6]. We give here a short reminder in order to introduce the specific use of self-optimization in this context.

3.1 Introduction to benchmarking and optimization requirement

Benchmarking typically consists in comparing the performance of possible alternative computing elements (software/middleware, hardware). For instance, one may be interested in comparing the performance of a number of Java Virtual Machines, databases, application servers, etc. To do this, it typically takes:

- a reference application that uses these alternative elements, in order to have something to measure (e.g. application response times or computing resources consumption);
- together with a workload specification that defines the flow of requests that the reference application will be subject to.

For example, in the context of Web applications, the Rubis benchmark [1] provides a number of implementations of an on-line auction web application that can be run on a variety of JVMs, applications servers, databases, etc., together with an HTTP traffic generation utility that defines a special mix of different HTTP requests. By measuring the performance of this application with different JVMs (or databases or...), the tester can compare the performance of these alternatives and make the best technological choices.

One of the key issues with benchmarking is that alternatives must be tuned, or, in other words, configured to get their optimal performance, in order to get meaningful, comparable measures. To go on with the Rubis benchmark example, JVMs, applications servers, databases, etc. all have specific parameters whose settings may (or may not) influence the overall application performance. Of course, the optimal settings of one alternative are typically closely dependant on the application.

To conclude, the benchmarking activity typically relies on looping on the following steps:

- finding the performance limits of a tested element with a given configuration (observation)
- tuning the element configuration for better performance (feedback)

3.2 Towards autonomic benchmarking

3.2.1 Principle

Starting from the observation that benchmarking activities involves a human feedback loop in a complex technical infrastructure (tested element, workload generation system, network...), our idea is to apply Selfware's architectural approach to autonomic management of complex systems in the field of performance benchmarking.

Roughly speaking, it consists in supporting both the lookup of saturation step and the performance optimization step as autonomic processes. The first step typically relies on a self-regulated load injection system, autonomously adapting the workload level to the observed performance of the target element. This step must finally provide a performance metric for the tested configuration. The second step consists in changing the tested element's configuration in order to improve its performance. We develop these two steps in the following two sections.

3.2.2 Self-regulated load injection

The self-regulated injection aims at looking the saturation limits of the tested element in terms of performance. The concepts of performance and saturation must be resolved into measurable metrics and Service Level Objectives that are meaningful to the tester. As a matter of fact, testers may consider different metrics and SLOs. Let's mention for instance:

- metrics: response times experienced from the user point of view, request processing time in the system, system load (processor, memory, disk transfer... or an arbitrary combination), network bandwidth usage, request rejection or error rate...
- SLOs: acceptable values of these metrics, in terms of average, maximum, standard deviation, distribution...

The self-regulated load injection system can be represented as an autonomic system with the Selfware approach, with a control loop:

- measures (observation),
- SLO check (decision),
- adapt the workload level (reaction).

3.2.3 Self-optimization

Self-optimization consists in generating possible configurations (basically in terms of parameter settings) of a tested element in a way that improves its performance within the current benchmark (reference application and workload). This process is based on the performance observation resulting from the self-regulated load injection process. Then, a reconfiguration decision is made and applied to the tested element. Here again, Selfware's control loop-based autonomic architecture applies.

Self-optimization relies on a classical generate-evaluate process which is supposed to explore a multi-dimensional space of solutions. Roughly speaking, the tested elements can be configured by setting a number of parameters, with a range of possible values. A number of issues arise then:

- some parameter values may be incompatible with each other;
- some parameters may be correlated;
- the number of parameter values combinations may be far too huge to be able to explore them all.

Factor analysis statistical techniques may be used in order to identify and eliminate parameters that don't influence performance. Heuristics may also be introduced in order to guide the exploration in an efficient way. Finally, a sufficiently good configuration, albeit not the best configuration, shall be found.

3.2.4 Self-benchmarking

Finally, self-benchmarking combines both self-regulated load injection and self-optimization. This combination requires an upper-level control in order to drive the load injection and the self-optimization processes in a consistent manner with regard to the benchmarking motivations:

- getting a final metric that rates the performance of a given configuration,
- getting a final conclusion about the best configuration.

Generally speaking, there are two main approaches to combining the load injection-based performance evaluation process and the self-optimization process.

1. We could dynamically change the configuration while a given workload is being applied to the system under test in order to see if performance changes (e.g. response time or resource consumption evolution). This approach requires being able to change a configuration while the system is running, very quickly in order to avoid disturbing measures from the load injectors and probes. This is almost impossible with the kind of computing systems that are in the scope of Selfware. Most of the time, reconfiguring will require to stop and restart the reconfigured element. Should dynamic reconfiguration be possible, it would take a significant piece of time anyway. So, in any case, it is necessary to stop the load injection and to ignore measures from probes (or even possibly stop some probes if they are bound to the reconfigured element) while the reconfiguration takes place. Furthermore, the load injection should be started with an initial ramp-up workload so that the reference workload for performance comparison is not applied at once: it is very likely that either the load injection system or the tested system would not to stand it. Finally, another important drawback lies in the issue of defining the reference workload for performance evaluation: if it is not heavy enough, it might be impossible to significantly compare the performance of the different

configurations. But, if the workload is too heavy, the tested system might saturate and become unstable. Getting a tested system out of control is actually a risk in the self-benchmarking approach because it makes it harder to drive the autonomic benchmarking campaign. So, it ends that the only practically possible option is the following.

2. We'd better go through well-separated reconfiguration-performance evaluation steps. Instead of evaluating performance for a given reference workload, we will take advantage on the self-regulated load injection in order to measure the maximum workload the tested system can stand with regard to specific SLO. For each configuration of the tested system, the self-regulated load injection will start with a minimal workload, and will progressively increase the workload to reach the limit of the SLO satisfaction. The workload may also be decreased if the SLO is violated, and so on. Stability is one of the key issues.

3.3 The Autobench architecture

3.3.1 Overview

The Autobench architecture is based on Fractal components bound together to implement both control loops. In the Selfware architecture terminology, we have two control loops, with two Autonomic Managers (AM), one of them being controlled by the other one (see figure 8):

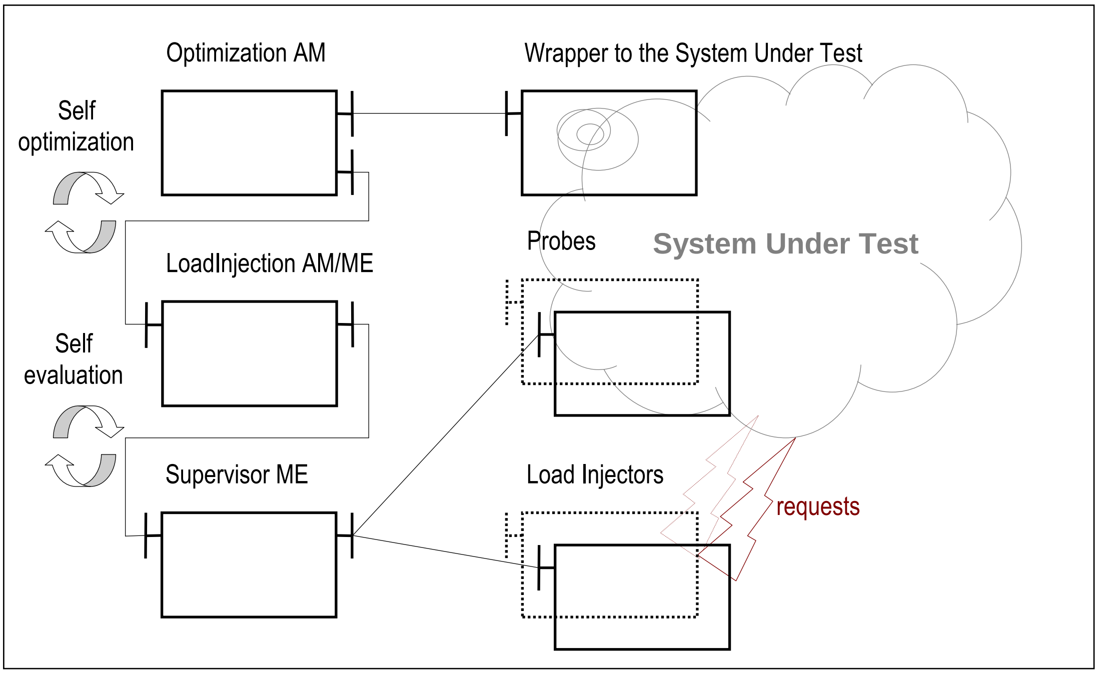

figure 8. Overview of the Autobench architecture for self-benchmarking

The *load injection AM* implements the decision logic for the self-regulated load injection system. The Managed Element for this AM is the CLIF Supervisor component that is actually a front-end for controlling an arbitrary number of load injector components from the CLIF load injection framework [1]. Through the TestControl interface, the AM can control the activity of all the load injectors (start, stop, suspend, resume, etc.) and dynamically adjust the number of virtual users for each load injector. Moreover, this interface also gives access to monitoring statistics about load injector components (response times, throughput, error rate...) and probe components from the monitoring service (see [5]). These probes may deliver consumption measures about arbitrary resources: system load, CPU load, memory usage, disk transfer rate, network bandwidth usage, or any other resource usage measured through JMX or SNMP standards, or in an ad hoc manner (e.g. parsing some log files or interrogating some system tables in a database).

The *optimization AM* that generates configurations for the system under test, and controls the activity of the load injection AM in order to evaluate these configurations. To enable this evaluation and finally compare different configurations, the load injection AM provides the optimization AM with a performance metric. As a result, the load injection AM is also a ME for the optimization AM, since it is set up and monitored by the optimization AM.

3.3.2 Focus on the load injection self-regulation

The generated workload is based on the definition of a virtual user behavior, using CLIF's ISAC scenario tool (Isac is a Scenario Architecture for Clif). This behavior is supposed to represent the typical behavior of a user of the tested element. Whenever necessary, Isac's probabilistic branching statement shall be used in this behavior to represent a given mix of users with different behaviors. The initial, minimal workload for each configuration evaluation consists in activating a single virtual user. Then, the number of virtual users is successively increased (or decreased), typically depending on the distance between current performance measures and the target SLO satisfaction limit. For instance, if the SLO is defined by a system load less than 80%, the workload increment may be higher if the system load measure is 20% than if it is 60%. Conversely, if the system load measure becomes greater than 80%, the workload must be decremented.

In control theory terms, we will use a linear feedback loop with stability guardrails, as shown in the algorithm depicted by figure 9. This algorithm applies to a metric that linearly evolves in a non-saturated situation. Stability guardrails limit the feedback level in order not to go brutally into the saturation zone, while still keeping a chance of converging toward – but always below – the saturation limit. We typically want to avoid two difficult situations:

- persistent instability with a succession of big load increments and decrements that never converges to a stable load;
- true saturation which would make the system under test become unstable or merely crash. By unstable we mean a non deterministic behavior that would make the metric irrelevant.

```
metric_max is given as a input parameter
vusers = 1 // initial number of virtual users
loop
        wait for stability (e.g. constant delay)
        get metric
        delta = (metric - metric_max)/metric_max // relative distance to the target metric
        increment = vusers * delta // linear feedback (simple rule of three)
        // since the system under test is non-linear when approaching
        // the saturation limit, we truncate the increment through 3 rules:
        // rule 1) try to consume just half of the apparently available power
        increment = increment / 2
        // rule 2) never do more than double the load
        if increment > vusers then increment = vusers
        // rule 3) never remove more than half of the load
        if increment < -vusers / 2 then increment = -vusers / 2
        vusers = vusers + increment
        apply the new load with vusers virtual users
```

figure 9. linear feedback algorithm with stability guardrails for self-regulated load injection

At the end of this self-regulated load injection process, we must get a single metric evaluating the performance of one configuration. But, with regard to the instability issue, when can we consider the experiment is finished? In other words, when can we decide that the overall autonomic system (load injection + tested system) is in a (sufficiently) stable situation? As a matter of fact, basic experiments with actually used technologies (see for instance [3]) have shown that the concept stability is not straightforward to characterize. A simple and practically efficient stability definition consists in defining a sufficiently long time frame, long enough with regard to the tested system activity. Then, the performance metric that may be defined is the total number of requests actually served (possibly with a maximum response time) during the experiment.

3.3.3 Focus on self-optimization

Key issues about self-optimization have been sketched already in sections **Erreur ! Source du renvoi introuvable.** and 3.2.3. In the context of Selfware, we don't plan to produce contributions in the field of operational research or artificial intelligence, since we focus on developing and illustrating an architectural approach and a framework for autonomic computing. Thus, we don't go into complex illustrations which would combine several configuration parameters of a variety of types. We will typically consider a single numeric parameter and perform a dichotomic search within a given range. For a given system under test, we will choose a duplet {performance metric, tuning parameter} that makes sense, i.e. where the tuning parameter actually influences the performance metric.

4 Conclusion

This document has described the solutions that were experimented in the Selfware project regarding self-optimization purposes.

The first self-optimization manager has more precisely focused on self-optimizing Internet services. The proposed approach, based on resource usage observations and on simple architectural reconfigurations, is attractive thanks to its simplicity. Its basic design confers a generic behavior to the self-optimization manager, allowing its appliance to many different Internet services with minimal effort. Of course, genericity is reduced when self-optimization makes use of application-level heuristics, like it was the case for tackling sudden load variations (such observations being application-dependent). A major open issue of this work concerns tuning and configuring the self-optimization manager itself. Indeed, configuration parameters may have the form of min and max thresholds used to guide dynamic resource provisioning, inhibition delays used to prevent system oscillation. Configuring these parameters must be carried carefully since it conditions the efficiency of the self-optimization manager. In our experiments, we manually tuned these parameters of the self-optimization manager, based on an observation of the behavior of the underlying Internet service.

The second self-optimization manager is dedicated to self-benchmarking activities. The idea of self-benchmarking is to introduce autonomy in performance evaluation, as well as in optimization. A full architecture has been described: it combines load injection and probe components from the open source CLIF load injection framework with two managers: the load injection manager that looks for the saturation limits of the system, and the self-optimization component that generates possible configurations. This results in combining two autonomic control loops, which is a very challenging issue in the general case. So, we proposed simple algorithms through a simple hierarchical organization of both loops. The motivations for self-benchmarking in the Selfware context is to deploy pre-optimized services platforms and applications, instead of taking the risk of deploying badly configured systems with poor performance. Of course, this does not alleviate the need for runtime self-optimization capabilities, since the runtime workload is subject to possibly dramatic changes. Actually, self-benchmarking could also be used to assist, and possibly automate, the configuration of the parameters of the first self-optimization manager, to make the usage of self-optimization easier and more efficient.

5 References

[1] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, Willy Zwaenepoel, "Performance Comparison of Middleware Architectures for Generating Dynamic Web Content", 4th ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003.

[2] Dillenseger (B.), Flexible, easy and powerful load injection with CLIF version 1.1. Fifth Annual ObjectWeb Conference, Paris La Défense, January 2006.

[3] Harbaoui H., Dillenseger B., Vincent JM, "Performance characterization of black boxes with self-controlled load injection for simulation-based sizing". *6ème Conférence Française sur les Systèmes d'Exploitation*, Fribourg, 11-13 February 2008.

[4] Selfware Architecture, Livrable SP1-Lot1, May 2007.

[5] Selfware Architecture, Livrable SP1-Lot2, November 2007.

[6] Selfware Java EE management scenarios, Deliverable SP3-Lot1, August 2007.

[7] Jing Xu; Sumalatha Adabala; Fortes, J.A.B, ICAC 2005. Second International Conference on Autonomic Computing.