Monitoring Scheduling for Home Gateways

Stéphane Frénot, Yvan Royon, Pierre Parrend, Denis Beras

▶ **To cite this version:**

Stéphane Frénot, Yvan Royon, Pierre Parrend, Denis Beras. Monitoring Scheduling for Home Gateways. IEEE/IFP network operations and management symposium, Apr 2008, Salvador de Bahia, Brazil. pp.411-416, 10.1109/NOMS.2008.4575162 . inria-00270941

HAL Id: inria-00270941

https://inria.hal.science/inria-00270941

Submitted on 7 Apr 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Monitoring Scheduling for Home Gateways

Stéphane Frénot, Yvan Royon, Pierre Parrend and Denis Beras
INRIA ARES / CITI, INSA-Lyon, F-69621, France
tel. +334 72 43 64 22 - fax. +334 72 43 62 27
{firstname.lastname}@insa-lyon.fr

Abstract— **In simple and monolithic systems such as our current home gateways, monitoring is often overlooked: the home user can only reboot the gateway when there is a problem. In next-generation home gateways, more services will be available (pay-per-view TV, games…) and different actors will provide them. When one service fails, it will be impossible to reboot the gateway without disturbing the other services.**
We propose a management framework that monitors remote gateways. The framework tests response times for various management activities on the gateway, and provides reference time/performance ratios. The values can be used to establish a management schedule that balances the rate at which queries can be performed with the resulting load that the query will induce locally on the gateway. This allows the manager to tune the ratio between the reactivity of monitoring and its intrusiveness on performance.

I. Introduction

During the last years, the Home Gateway market has evolved at a very fast pace. The business model has evolved from plain IP connectivity to triple play (voice, video and data). These 3 services are provided by a single entity: the internet access provider. In the close future, the business model will move to multi-play [1], and enable the management of smart homes [2]. The idea is that different sets of services, such as TV on demand, games, or home security, can be developed and deployed by various business entities other than the access provider. To push this idea further, different services will be delivered by various providers depending on the home user's choices.

Such a business model has a strong impact on their underlying technical infrastructures. Indeed, each service provider should be able to manage his own services. For instance, if the service is to monitor a pacemaker for an elder person, the hospital (or whoever is providing the service) should access the pacemaker's data as directly as possible. It is not acceptable that monitoring data is lost because the grandchildren are watching TV.

From the network point of view, the solution is to grant a certain Quality of Service to each provider, or even to each service. However, we must also look at the system point of view: home gateways have limited processing power, and management activities do have an impact on CPU utilization. Simply put, the more often a manager sends requests, the more accurate management data he will get. However, the CPU load will increase, until a threshold where services will not be able to run correctly.

We propose a management framework for home gateways that determines this threshold. It is implemented on top of Java/OSGi, using JMX. Section III gives a quick overview of JMX and OSGi. In section IV, we show how we can evaluate the cost of a `getAttribute()` request, namely a request to get the CPU usage on a gateway. Finally, section V describes a way to schedule management activities so that their cost in processing power are under control. Section VI concludes this work. [1]

II. RelatedWorks

We now present the state of the art in management of Java-based systems. An overview of the existing solutions is provided, and the JMX management opportunities, that emerges as one versatile solution for java applications. Until recently, the principal management technology has been the SNMP protocol. It brings with it a complete management framework, in particular data handling facilities with the MIB (Management Information Base). SNMP can also be used in home systems [3], as JMX is. However, no real integration with applications is provided, which makes the latter approach more relevant when high-level software systems must be managed. However, application management must be performed carefully, so as not to impair the system functions with management-related performance overhead. This question has been studied in the context of Web Services [4], and for EJBs [5], or with a focus on Operating System management [6]. The performance question for OSGi management will be discussed in detail in this paper. JMX is the Java Management Extension [7]. It is meant for managing and monitoring Java-based systems, through a so-called Agent that supervises probes. The probes are accessed through MBean interfaces. JMX is part of the Sun Java project, and is the subject of two complementary specifications: the SUN JSR 3, 'JavaTM Management Extensions (JMXTM) Specification' 2, and the 'SUN JSR 160: JavaTM Management Extensions (JMX) Remote API' 3. Several tools have been developed to exploit the JMX functionalities. The JConsole [8] is the sun monitoring and management tool. MX4J 4, which is an Open Source implementation of JMX, also provides a set of JMX-related tools. So as to provide a full control over the managed systems, which is often built out of several elements, JMX must be integrated in a suitable framework. Jasmine5 is such a management framework for enterprise applications, developed in the frame of the ObjectWeb Consortium. It aims at managing

[1]This work is partially supported by the IST-6thFP-507295 MUSE European Project

the various parts of N-Tiers systems ,such as Java EE, Message oriented Middleware, and Services Oriented Architectures). The intensive use of JMX for management brings scalability questions. This problem has been studied by [9], and will be further discussed in this paper. The specific instrumentation of OSGi platforms with JMX management facilities is more recent, and few powerful solutions exist. The Jasmine Project provides its own OSGi Console, which is rather simple, since it only lists OSGi Gateways and installed bundles. We developed a complete JMX Management Framework for the OSGi platform. The principle of our approach is presented in [10], and the core functionalities are shown is [11]. The current work introduces additional functionalities, and discuses performance optimization of JMX-based Management.

III. A Management Architecture for Home Gateways

In previous works, we have designed a JMX-based framework for home gateways [12]. It comprises a remote console, and software that runs locally on each gateway. In JMX, such software is:

- **JMX agent**: a singleton application that registers Java management interfaces called MBeans;
- **MBeans**: Java objects registered whithin the agent, and accessible by a public Java interface. Standard MBeans can provide 3 kinds of methods: get the value of attribute, set it to a new value, and invoke a method;
- **Connectors**: handled by the agent, they allow to access MBeans remotely, in our case using HTTP or RMI;
- **Probes**: feed management data to MBeans when performing get or set queries. Probes are implementation-specific, and are hidden by the MBean interface.

The role of the MBean is to provide information from the managed system to the remote manager. The information can be provided in two ways: through a solicitation from the manager (Response/Request), or through a notification from the MBean itself (Notification). The JMX specification enables various kinds of MBean (standard, dynamic, models and open) and various kinds of MBean behaviors (thresholds, relations, measurements); this study focuses on standard MBeans with get / set / invoke methods.

We have implemented these software parts as OSGi plug-ins (called bundles). The OSGi [13], [14] specifications define a framework that manages the life cycle of applications. These applications can be downloaded from a remote location, started, stopped, updated, removed. Finally, applications can also express dependencies towards other applications. The framework automatically checks that dependencies are met, and refuses to launch an application otherwise. The OSGi specifications do not specify any management architecture to control gateways. We have developed a JMX framework that enables the remote monitoring of OSGi based home gateways. The framework uses a JMX agent located on the framework, and an RMI (or HTTP) connector that enables the connection between an agent and a manager. We have also developed our own management console that connects to the agent. The entire framework is available in Apache Felix [15], an open source OSGi implementation. The management subproject is called MOSGi.

When managing a remote gateway, two interaction schemes are available: request/reply and notifications. With request/reply interactions, the manager periodically polls the gateway for values (memory, bandwidth. . .). With notfications, the gateway sends messages either on a timer basis or on a threshold basis. In both interaction schemes, management activities do have a cost in terms of processing power. This load corresponds to the various probes running on different threads and to the agent that maintains remote connections and a registry of current MBeans. Our concern is that the remote gateway has limited resources. These resources are shared between both user services (TV, voice, games) and management activities. This paper tries to elaborate a way to determine the volume of management activities that can be imposed on a home gateway without disrupting user services.

IV. Management Cost Evaluation

Managing home gateways has to cope with 2 opposite visions. When something goes wrong the gateway operator should be warned as early as possible. But in order to achieve this, the gateway needs to handle an extra management activity that burdens the gateway. The question is simple: supposing that a figure under 5% of resource consumption is not intrusive, how often can a manager query management data? The quantity is expressed as the number of requests per second (or minutes) that can be sent. This measure reflects the response time an administrator can expect on a remote device. A similar question has been addressed in [16] for managing networking equipment.

A. CPU measurement

As a first approach we are working in request/response mode. Figure 1 shows the CPU consumption that is induced when querying the CPU utilization ratio.

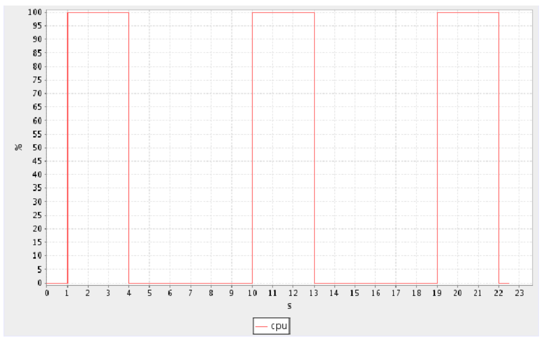


Fig. 1. Cpu Monitoring

In this example the polling is made every 9 seconds and the probe returns its value in 3 seconds. If the gateway does nothing else than reacting to this query, it should present a load of 33%. It is easy to understand that the load average is directly related to the execution time (in CPU cycles) of the `getCpu` method. In the Unix world, the `/proc/stat` file holds data needed by the getCPU probe. The generic pseudo-algorithm of the probe is the following:

```
public int getCPU(){
  open(/proc/stat);
  int currentIdleValue=readIdleValue();
  int cpu=(currentIdleValue-oldIdleValue)/Duration;
  oldIdleValue=currentIdleValue;
  close(/proc/stat);
  return cpu;
}
```

Listing 1. Pseudo-code for the getCPU probe

B. Determining the cost of the CPU probe

In order to manage a remote device, we need to know at which rate we can request data from it, and which quantity of data we can get in each request. The more requests by minutes, the fastest we can identify a fault, but the more load we put on the remote device. The theoretical curve obtained is shown in figure 2 (smooth curve). This is a direct way of obtaining the request rate the manager can apply. According to these estimations, if the manager asks for the cpu every 250ms he will generate a cpu load of 17%. If he asks every 2s the rate drops down to 2%. Of course if the observation is made every 2s, the system cannot identify the failure and the end-user can feel a service disruption.

The next section is a proposal to automatically obtain this curve. Knowing this curve, a manager can elaborate a management schedule among the various equipment available. This management planning should enable a fast decision making without disrupting the user's services.

V. A framework for management scheduling

A. Determining the cost of a management probe

As was presented in the previous section, we want to find a way to establish the consumption curve. The curve establishes the CPU rate involved by the management system. We have developed two approaches: the first one is empirical, and tries various requests rates; the second one is theoretical and finds significant points that allow to deduce the curve.

In the empirical approach, we define a series of rates at which the remote equipment is solicited. After setting these rates, the management console triggers a value request for each of them. This test, run on a LinkSys NLSU2 equipment, gives the second curve (not smooth) on figure 2. The values start at 97% which means that whatever the manager does, he will not be able to load the remote system more than 97%. We also see that there is an asymptote at about 0.5% which represents the average load without any activity. This load is achieved for a probing period of approximately 2s. So if we want not to disrupt the load of the managed system, we can make one request every 2s. A more important information is that we can see that the curve has the $\frac{\alpha}{x} + \beta$ form. Below we try to find the theoretical curve in a faster way, in order to determine the probing period that match 5% of system load (which can be considered as negligible) efficiently.

$\frac{\alpha}{x} + \beta$ is a trivial curve. The β parameter represents the load without activity and the α parameter represents the peek load that the management system can put on the remote equipment. For instance, if the period is 1s for a 100% load, it is trivial that for a 2s period the load should be 1/2 which correspond to 50% load (with the β parameter approximation). So the problem is to find a way to extract the α and β parameters of this curve. We have build an application that makes two measures and extrapolates the curve. For the first point, the application sends requests as fast as possible and for the second point, the application tries a long period of requests.

Figure 2 compares the two curves previously presented.

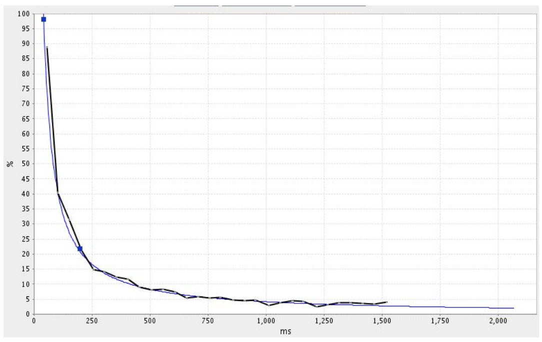
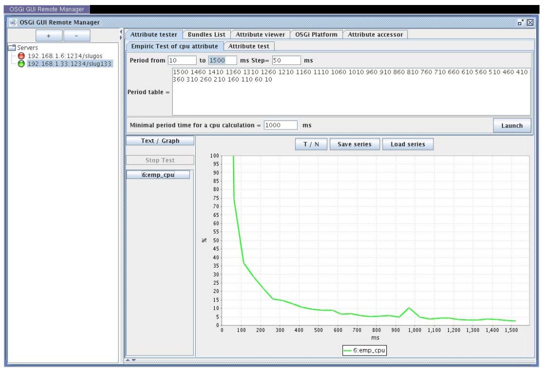


Fig. 2. Estimated load curve and empirical results

The smooth curve is the one deduced from the significant points. The other one represents the empirical measurements. We see that testing all values or taking only two measures lead to similar results. The next step is to choose the 2 points that allow to quickly calculate the estimated curve.

B. A management tool for evaluating the cost of a probe

We have integrated the tool to make these evaluation in our management architecture. It relies on a management agent residing on the remote equipment which reacts to requests from the management console. The management console has the following user interfaces:



Period from 10 to 1500 ms Step= 50 ms

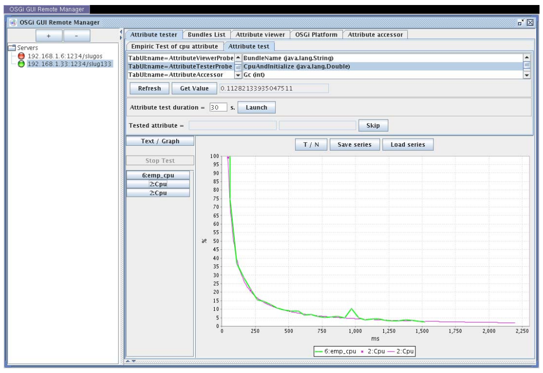
Period table = 1500 1460 1410 1360 1310 1260 1210 1160 1110 1060 1010 960 910 860 810 760 710 660 610 560 510 460 410 360 310 260 210 160 110 60 10

Minimal period time for a cpu calculation = 1000 ms

Fig. 3. Management Console: Empirical tab

The left part of the user interface (figure 3) enables the selection of the remote equipment. The right top part of the interface enables the input of parameters. In the sample the

user wants to test request periods from 1500ms to 10ms of delay with a step of 50ms. When he validates the value, a series of testing periods is provided (the top Java TextArea). The minimal probing period is an important value, it tantamounts to the maximal rate of information retrieval by the manager. If periods are too short, tests are biased with the time the probe needs to perform the calculation. One bias is linked to the access to `/proc/stat`: the delay can interfere with measurements. The other bias is linked to management timers: Linux cannot provide precise data if the querying is too fast. In order to absorb these two bias we define the Minimal period time for a CPU calculation value, which force the request/reply cycles at least to this value. The value corresponds to the minimum delay at which the system will be loaded for a period of time. Finally if the user clicks on the 'Launch empiric' button it will generate a series of queries for each period, and provide the corresponding curve.
The second tab "CPU attribute test" of this user interface drives to the $\frac{\alpha}{x} + \beta$ curve production. Figure 4 shows this interface. The upper zone enables the user to choose the tested



TabUI:name=AttributeViewerProbe
TabUI:name=AttributeTesterProbe
TabUI:name=AttributeAccessor

BundleName (java.lang.String)
CpuAndInitialize (java.lang.Double)
Gc (int)

0.11282133935047511

Attribute test duration = 30 s.

Tested attribute =

Fig. 4. Management Console: $\frac{\alpha}{x} + \beta$ tab

probe and the time duration of the test. When he presses the *launch* button, the system chooses automatically two periods to test and determines the α and β parameters from a curve extrapolation.

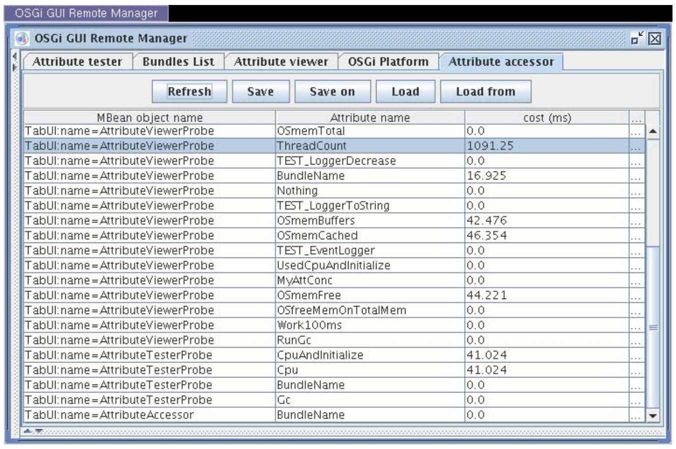
We implement some heuristics in order to choose these two periods:

- The second result should be at least 70% lower than the first one. If the first value generates 97% of CPU load, the second point should be evaluated somewhere where the result is lower than 29%.
- The same number of requests must be executed for the points determination. This balances the garbage collect activity.
- The system should control that the activity without management is stable during the tests. This means that before, between and after the tests, the activity is stable.

As a conclusion, our approach enables a fast evaluation of the management cost of the CPU probe. The curve can be determined with two significant points. In the next section, we extend this to the evaluation of all available probes.

C. *A generalisation of scheduling for any management probe*

In the previous section we showed that we could evaluate the cost of the CPU probe for a low end system. Since the remote equipment is managed in a multi-service environment we think that not only the CPU cost should be evaluated. In many business scenario the CPU load is not relevant and many other values can be more important. Significant values can be either "classical" ones, like available memory or bandwidth, but it can also be more specific ones such as "how many services are deployed" (platform management data) or "how many beers are in the fridge" (application level data). Provided we have the corresponding MBean , the framework is are able to evaluate the cost for any probe. We automate the process of evaluating any remote probe. Figure 4 shows the associated user interface. The service manager selects the probes he wants



MBean object name	Attribute name	cost (ms)	...
TabUI:name=AttributeViewerProbe	OSmemTotal	0.0	...
TabUI:name=AttributeViewerProbe	ThreadCount	1091.25	...
TabUI:name=AttributeViewerProbe	TEST_LoggerDecrease	0.0	...
TabUI:name=AttributeViewerProbe	BundleName	16.925	...
TabUI:name=AttributeViewerProbe	Nothing	0.0	...
TabUI:name=AttributeViewerProbe	TEST_LoggerToString	0.0	...
TabUI:name=AttributeViewerProbe	OSmemBuffers	42.476	...
TabUI:name=AttributeViewerProbe	OSmemCached	46.354	...
TabUI:name=AttributeViewerProbe	TEST_EventLogger	0.0	...
TabUI:name=AttributeViewerProbe	UsedCpuAndInitialize	0.0	...
TabUI:name=AttributeViewerProbe	MyAttConc	0.0	...
TabUI:name=AttributeViewerProbe	OSmemFree	44.221	...
TabUI:name=AttributeViewerProbe	OSfreeMemOnTotalMem	0.0	...
TabUI:name=AttributeViewerProbe	Work100ms	0.0	...
TabUI:name=AttributeViewerProbe	RunGc	0.0	...
TabUI:name=AttributeTesterProbe	CpuAndInitialize	41.024	...
TabUI:name=AttributeTesterProbe	Cpu	41.024	...
TabUI:name=AttributeTesterProbe	BundleName	0.0	...
TabUI:name=AttributeTesterProbe	Gc	0.0	...
TabUI:name=AttributeAccessor	BundleName	0.0	...

Fig. 5. Attribute CostResult

to evaluate. For each probe the framework evaluates the delay (minimal time between each probe request) that corresponds to an increase of 100% of the remote equipment load.

Figure 5 shows results for some tested probes. For example the OSmemFree probe costs 44.221ms which means that requesting OSmemFree probe every 44ms will increase gateway CPU activity by 100%. So If we want a 5% load, the corresponding period is *i.e.* 884ms. The 0.0 values just indicates that probes are not tested yet.

These data can be used to elaborate a management plan. This plan indicates which and when each management probe can be requested. For instance if the manager asks the OSmemFree value every 884ms and the Cpu probe value every 820ms we will have an increased CPU load of 10%. This management plan is a fundamental concept where the manager needs to find the right balance between the reaction time he wants and the load put on the remote equipment. The more reactive he wants to be, the more load he puts on the remote site. Establishing the plan a-priori can lead to a better anticipation of the system evolution.

D. *Results*

We validate our framework on three classes of systems: 1) an Intel dual core running a desktop environment, 2) an EPIA

1000Mhz simulating a high-end multimedia home gateway, and 3) a LinkSys NLSU2 representing a lightweight home gateway.

All these system runs Gentoo Linux with the JamVM virtual machine and Felix/MOSGi as the management system[2]. The first question to answer is : ”is the theoretical curve compliant with empirical tests”. Figures 6, 7 and 8 represent these two curves for the 3 test environments. The curve with square points represent the ”empirical” approach the curve with round points is the curve deduced from α and β parameter measures.

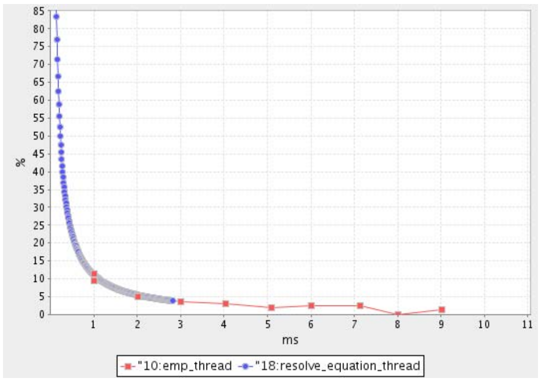


Fig. 6. Dual Core CPU cost

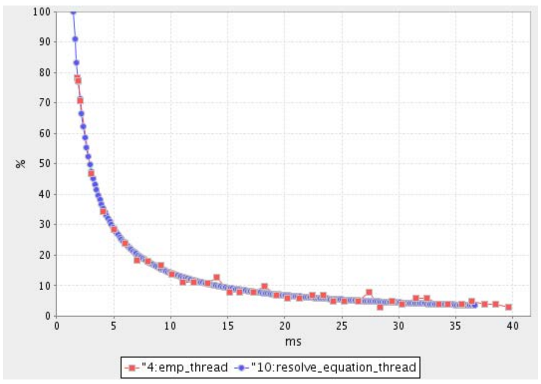


Fig. 7. Via Epia 1000 CPU cost

The empirical approach for the dual core figure stops at 12% load. This means that the manager cannot stress the remote equipment to a big load. This is due to the fact that the dual core is too powerful for the manager to be loaded. For the three figures we see that the empirical and the α / β approaches produce the same results. The latter is far faster since all the curves are produced from two reference measures.

E. Monitoring

Once the service provider has determined which data he wants to monitor and at which rate he wants to get information, he sets these values on the monitoring management panel. Figure 9 shows an example of remote monitoring panel. In this

[2]All the code is available as opensource at http://cwiki.apache.org/FELIX/index.html, and http://mosgi.gforge.inria.fr/

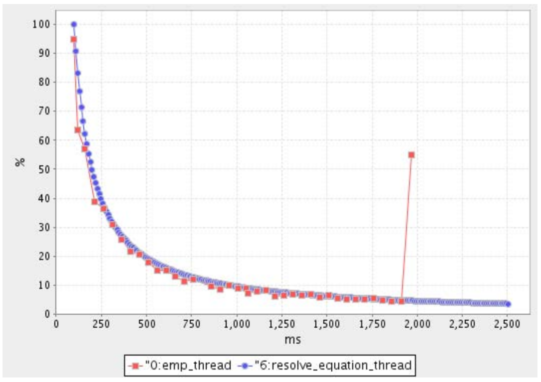


Fig. 8. LinkSys NLSU2 CPU cost

example the panel is targeted to a human administrator but it can be aimed at an automated system based on thresholds and alarms.

The monitoring console shows 2 periods (PI, PII). They display monitoring curves for 4 probes (c1, c2, c3, c4). From upper to lower curve :

- The c1 curve is the OSmemFree probe,
- The c2 curve is the OSmemBuffer probe,
- The c3 curve is he Cpu probe,
- The c4 curve is the OSmemCached probe.

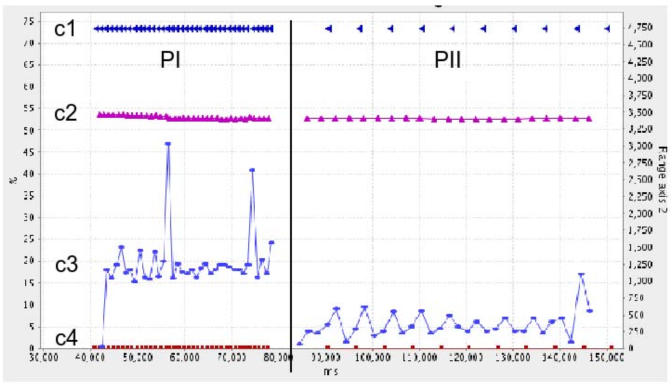


Fig. 9. Slug monitoring platform

In PI all probes request periods are set to 1s. So our framework predicts a CPU load for c1 to c4 respectively of 4.4%, 4.2%, 4.6% and 4.1%. We can see that the Cpu probe returns an average value of about 17.5% which is about the sum of the c1 to c4 probes cost. In PII we want to specify a request period for Cpu probe (c3) of 3s and for OSmemFree (c1) of 2s which, according to our framework, generates a CPU load of 3.6% total. So if we do not want to generate more than 5% CPU load with our management plan, we set request periods for the 2 remaining probes in order to generate a maximum of 1.4% CPU load. In the figure we choose *i.e* 0.7% for c2 and 0.7% for c4 the system calculates a resulting delay of 6069ms for c2 and 6623ms for c4.

A last use case of our system is to make benchmarking between systems. We use our system both on a Felix OSGi framework and on a Concierge OSGi framework. We observed that the mean response time of most probes are slightly faster on Concierge implementation than on Felix which confirms that Concierge is optimized for small environments. These benchmarking issues are rather easy to set up and the results are directly observable with the monitoring platform.

Another use case we examined is to evaluate the quality of probe implementations. Our system can compare CPU consumption of similar probes in order to compare their performance.

VI. Conclusion

We present in this article a monitoring framework that aims at finding the balance between the frequency of remote queries and the resulting CPU load on the managed equipment. The monitoring process starts with an evaluation period where monitored system probes are queried in order to establish their characteristics. A probe characteristics is modeled with a curve that represent the induced load in function of the query pace. The curve has an $\frac{\alpha}{x} + \beta$ shape.

After having evaluated the curve a manager can elaborate a query plan. This plan fixes for each probe the query pace thus anticipating the resulting load induced by the management layers on the managed system. The overload is then due to other running applications. Of course one goal of the system is to guarantee that the management layers do not cost too much and that they stay below a certain limit.

This framework is aimed at home gateways; these run services and applications that should be remotely managed. Since those devices have quite limited resources, the management overload needs to be carefully tuned. The faster the management queries are made the faster the service provider can react to a perturbation, but the highest load he puts on the gateway.

References

[1] MUSE Project, “IST-507295 FP6,” http://www.ist-muse.org/, 2004.

[2] Baskar Sridharan and Aditya Mathur, “Infrastructure for the Management of SmartHomes,” Software Engineering Research Center Tech Report SERC-TR-177-P, January 2002.

[3] Radu State, Olivier Festor, and Isabelle Chrisment, “Context-Driven Access Control to SNMP MIB Objects in Multi-homed Environments,” in *DSOM 2003, Self-Managing Distributed Systems*, vol. LNCS 2867, 2003.

[4] R. Levy, J. Nagarajarao, G. Pacifici, A. Spreitzer, A. Tantawi, and A. Youssef, “Performance management for cluster based web services,” in *IFIP/IEEE Eighth International Symposium on Integrated Network Management*, 2003.

[5] Markus Debusmann, M. Schmid, and M. Kröger, “Generic Performance Instrumentation of EJB Applications for Service-Level Management,” in *NOMS 2002, Florence, Italy*, ser. Lecture Notes in Computer Science, M. Brunner and A. Keller, Eds. Springer, April 2002.

[6] Sujay Parekh, Kevin Rose, Joseph Hellerstein, Sam Lightstone, Matthew Huras, and Victor Chang, “Managing the Perfomance Impact of Administrative Utilities,” in *DSOM 2003, Self-Managing Distributed Systems*, vol. LNCS 2867, 2003, http://www.research.ibm.com/PM/RC22864.pdf. [Online]. Available: http://www.research.ibm.com/PM/

[7] H. Kreger, “Java management extensions for application management,” *IBM Systems Journal*, vol. 40, no. 1, pp. 104–129, 2001.

[8] M. Chung, “Using jconsole to monitor applications,” Sun Whitepaper, December 2004. [Online]. Available: http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html

[9] Abdelkader Lahmadi, Laurent Andrey, and Olivier Festor, “Performances et resistance au facteur d’échelle d’un agent de supervision basé sur JMX : Méthodologie et premiers résultats,” in *GRES*, 2005.

[10] Eric Fleury and Stéphane Frénot, “Building a JMX management interface inside OSGi,” Inria RR-5025, Tech. Rep., 2003.

[11] Yvan Royon, Stéphane Frénot, and Frédéric Le Mouël, “Virtualization of Service Gateways in Multi-provider Environments,” Component-Based Software Engineering, 2006.

[12] Yvan Royon and Stéphane Frénot, “Multiservice home gateways: Business model, execution environment, management infrastructure,” *IEEE Communications Magazine*, vol. 45, pp. 122–128, October, 2007.

[13] OSGi Alliance, “http://www.osgi.org/.”

[14] Dave Marples and Peter Kriens, “The Open Services Gateway Initiative: an Introductory Overview,” IEEE Communications Magazine, December 2001.

[15] Apache Software Foundation, “Felix OSGi R4 Service Platform implementation,” http://felix.apache.org/. [Online]. Available: http://svn.apache.org/repos/asf/incubator/felix/

[16] P. Moghe and M. Evangelista, “grap - rate adaptive polling for network management applications,” in *NOMS 98*, pp. 395–399.