A JMX benchmark

Laurent Andrey, Abdelkader Lahmadi, Julien Delove

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

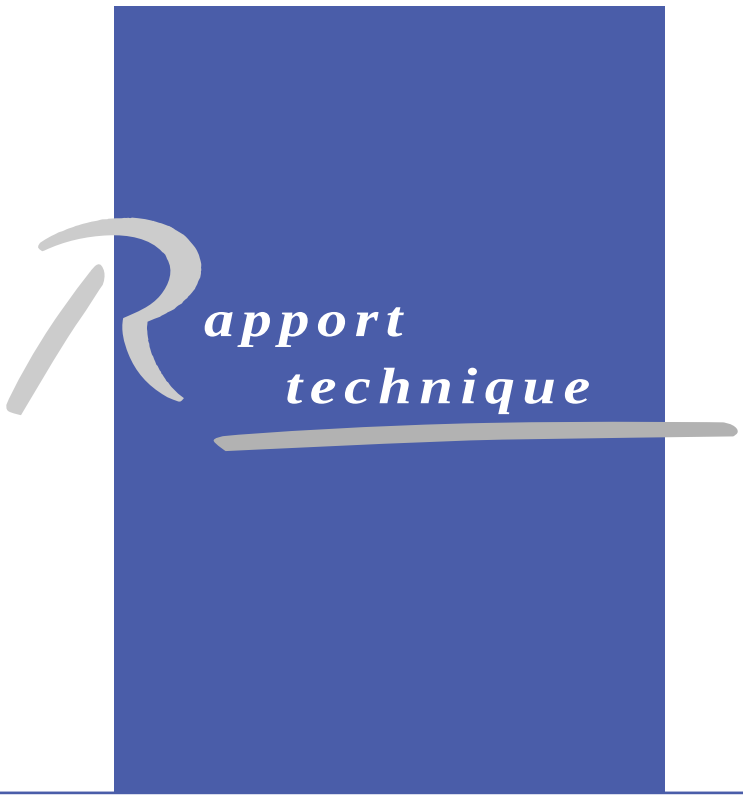
A JMX benchmark

Laurent Andrey — Abdelkader Lamadi — Julien Delove

N° ????

November 2005

Thème COM



ISSN 0249-0803

A JMX benchmark

Laurent Andrey , Abdelkader Lamadi , Julien Delove

Thème COM — Systèmes communicants
Projet Madynes

Rapport technique n° ???? — November 2005 — 26 pages

Abstract:

This rapport describes a performance test suite (benchmark) for JMX the *Java Management eXtention* the Java framework dedicated to Java applications and services management. This suite mainly target scale factors for JMX agents. The injection scenarios (workloads) are synthetic, they do not try to perfectly mimic a real management traffic. The test suite includes: -a system under test (a configurable set of configurable MBeans); -some loads injectors ; -scripts to start tests and collect measures ; -scripts for a *post-mortem* analysis of measures.

This work has been supported by a French Amarillo RNRT's project grant (`http://www.telecom.gouv.fr/rnrt/rnrt/projets_anglais/amarillo.htm`).

Key-words: benchmarks, JMX, networks and services management

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Une suite de tests de performances pour JMX

Résumé :

Ce document décrit une suite de tests de performances pour JMX *Java Management eXtention*, l'infrastructure de gestion d'applications et services en Java. Cette suite de tests vise surtout des facteurs liées au passage à l'échelle des agents JMX. Les *scenarii* d'injection sont synthétiques, ils ne cherchent pas à mimer parfaitement un trafic réel de supervision.

La suite de tests se compose de - systèmes sous test (un agent configurable) ; - injecteurs de charge configurable ; - scripts pour démarrer un test (mesure) et collecter ses résultats ; - scripts pour l'analyse *post-mortem* d'un ensemble de mesure et obtenir la visualisation selon certaines métriques.

Ce travail a été partiellement réalisé et financé dans le cadre du projet RNRT Amarillo (`http://www.telecom.gouv.fr/rnrt/rnrt/projets_anglais/amarillo.htm`).

Mots-clés : test de performances, JMX, gestion de réseaux et services

Contents

I Benchmarks design and organization 4

1 Goals and context 5
1.1 Pre-requisite . . . 5
1.2 Facts and Goals . . . 5

2 Test characteristics 6
2.1 Services Under Test . . . 6
2.1.1 Further studies . . . 6
2.2 Systems under test . . . 6
2.3 Metrics . . . 7
2.4 Factors . . . 7
2.4.1 JMX implementations . . . 7
2.4.2 Types of MBean . . . 8
2.4.3 Number of attributes . . . 9
2.4.4 Number of mbeans . . . 9
2.4.5 Number of request injected per second . . . 9
2.5 Parameters . . . 10
2.5.1 Kind of remote connectivity and related parameters . . . 10
2.5.2 Overall hardware, operating system . . . 10
2.5.3 Java Virtual Machine . . . 11
2.5.4 Java compilation and libraries packaging . . . 11
2.5.5 Network . . . 11
2.5.6 Data types . . . 11
2.5.7 Service interaction complexity . . . 11
2.5.8 Service invokation and proxy . . . 11

3 Tests suite architecture and organization 12
3.1 What is a test ? . . . 12
3.2 Simple get polling scenario . . . 12

4 Load injectors 13
4.1 Load injection issues . . . 13
4.2 Ramp-up . . . 17

5 Agent under test 17
5.1 Support for $TypeJmx$ factor . . . 19
5.2 Support for $TypeMBean$ and $NbAttr$ factors . . . 19
5.2.1 *std* and *dyn* cases . . . 19
5.2.2 *model* case . . . 20

5.3 Support for *NbMbean* 20

6 Console, Tests starting and execution **20**

7 *Post-mortem* analysis **21**

II User manual **21**

8 System and hardware requirements **21**
8.1 System and sofwares requirements 21
8.2 Getting the suite sources 22
8.3 Shell configuration and user rights on test nodes 22

9 Running a test, getting a measure **22**
9.1 A full example 22
9.1.1 Step one: running one injection 23
9.1.2 Step two: *Post-mortem* and visualization 24

10 Other notes **24**
10.1 Using eclipse for coding injector and agents 24

Notations

In the following sections:

1. `$JMXBENCHROOT` stands for the top level directory of the tests suite distribution;

2. `$USERHOME` stands for the user's home directory. "User" is the unix account under which any test scripts or programs are run.

3. When a class name is referenced as `org.objectweb.jmxbench....` the corresponding java source file is found in `$JMXBENCHROOT/java/src/org/objectweb/jmxbench/...`

Part I
Benchmarks design and organization

1 Goals and context

1.1 Pre-requisite

We assume that the reader has enough knowledge around the following keywords: JMX[SUN02, SUN03], Java, bash, perl.

We will as much as possible try to use concepts and terminology of performances analysis domain as found in [Jai91, Bur04]. These terms are highlighted using sans serif font.

1.2 Facts and Goals

JMX is to be the choice management and supervision framework for Java applications. Typically the question a java application or middle-ware developer is immediately asking about JMX is: "*How much does all this cost ?*". A second question could be "*And how does it scale ?*".

To get a first idea of what could be the answers we decided to conduct a couple of performance tests ("bench marks") and we selected:

- a rate metric: number of get request per second a MBean server can reply to;
- some overheads to measure (utilization metrics): %CPU, %memory and network traffic;
- simple scenario for load injection (workload) and scaling studies.

It seems there is no study about analysis and qualification of traffic generated by JMX management applications. Some are available for the Internet's *Simple Network Management Protocol* (SNMP) [Pat01]. We do not intend such study in this report. There are ever less studies about JMX agents resources consumption.

We focus on supervision which usually relies on many variables polling from a set of network nodes (the "managers") at various rates. But usual workloads are supposed to mimic real applications. As we do not have precise idea of the associate traffic we will try to conduct tests in an exhaustive way of get a raw picture of how costly polling is.

Once enough tests and measures will be conducted we would be able to have indications about how JMX scales against a couple of factors and how much resources it needs to handle a target polling load .

The developed workloads and test suite utilities will be freely available to able anyone to replay tests on its own infrastructure. A first example of use of this test suite can be found in [LAF05] (in french).

The next sections of this part makes explicit the various factors and parameters introduced to define the workloads.

2 Test characteristics

2.1 Services Under Test

For now only one JMX services under test is supported: the basic but commonly used "get" service. This choice reflects usual usage for JMX: variable(s) polling using the `Object javax.management.MBeanServer.getAttribute( ObjectName name, String attribute)` method. We assume therefore that the representativeness of the "get" of service is good enough.

2.1.1 Further studies

Further studies still around monitoring with other JMX services should be conducted. Let's cite:

- comparison of `Object javax.management.MBeanServer.getAttribute(ObjectName name, String attribute)` vs. `Object javax.management.MBeanServer.getAttributes(ObjectName name, String[] attributes)` ;
- comparison of polling vs. JMX higher level monitoring service (Gauges, Counters with asynchronous notification).

Types of MBean (standard, dynamic, model...) could be seen as services, but for a pure test point of view: the three cases can not be differentiated by an external manager. So the type of MBean is defined as a test factor (see section 2.4.2).

2.2 Systems under test

The System Under Test (SUT) could vary according to injection scenarios. In the case of simple study of how a JMX server behaves against "get" service solication the SUT is composed of:

- the JMX agent with all its components (adaptors, connectors), and Mbeans
- the Java Virtual Machine (JVM) supporting this agent
- the operating system supporting the Java Virtual VM (with its IP stack)
- the hardware supporting the operating system, englobing the network interface (i.e Ethernet Card)

To be really precise the whole network between users (workload injectors) and these previous elements should be added in SUT to some point. Indeed, for some metrics (request latency by example), the way to measure them catches also network behavior (some probes

and log are on injector side). In this case the network must be taken in a wide sense: true IP network, plus local network interface, plus IP stack, plus JMX connector on injector (JMX client) side.

2.3 Metrics

The metrics the proposed tests suite intends to support are:

- CPU, network, memory (Utilization metrics) ;
- throughput: real injection return, or in other words numbers of correct requests completed per second (Productivity metric) ;
- service request latency.

That means that the tests suite provides the proper probes and analysis tools (scripts) to get visualization for these metrics.The former metrics and some measure data allow to elaborate more sophisticated metrics as:

- knee capacity usually considered as the optimal operating point for the SUT;
- maximum capacity : the point after which one considers that the SUT does not provide the service under test anymore;
- even more elaborated: true scalability indicators using a $Production/Consumption \times Qos$ scheme as proposed in [JW02].

The tests suite does not provide help to calculate or to visualize the last three metrics for now.

2.4 Factors

Let us now list which test factors one can vary using the tests suite.

2.4.1 JMX implementations

We only investigate JMX implementations what are "free enough" and conformant to JMX 1.2:

- Sun reference implementation. This is the Reference Implementation (RI) for jsr03 and jsr160. The related **precompiled** jar files are available at sun's site: `http://java.sun.com/products/JavaManagement`. The tested distribution have version numbers: `1.2.8` for the core `jmxri.jar` file and `jmxremote-1_0-b23 2003.07.21_15:35:48_MEST` for the `jmxremote.jar` file.

- MX4J. The related precompiled jar files are available at `http://mx4j.sourceforge.net`. This version features an implementation of jsr160. The tested distribution have version numbers: `2.0` for both of the core `mx4j.jar` and the `mx4j-remote.jar` files.

  - BCEL issue. If the BCEL library [1] is available for class loading, MX4J activates an optimization for requests forwarding to standard MBeans. If this library is not available requests forwarding uses regular, not so performant, java reflection. So this is a sub-factor for MX4J implementation and we unfold it together with all other JMX implementation cases.

To sum-up, we define the following factor name and its set of possible values:
Factor *Types (vendors) of the JMX implementation*

$$TypeJmx = \{ri1\_2, mx4j{-}jsr160, mx4j{-}jsr160\_bcel\} \quad (1)$$

Remark 1: The cases $mx4j, mx4j\_bcel$ are also available, they are corresponding to an earlier mxj4 pre-jsr160 version using a proprietary rmi connector.

2.4.2 Types of MBean

JMX offers three different kinds of MBean which can be vue as three values of a factor which could affect performances. Let us detail these three cases:

- **standard**. They are made of one class and one interface, coded in an *ad hoc* way following a precise naming scheme (class, interface andmethods names). Usually JMX implementations use java introspection to achieve actual calls (get/set) which is supposed to be bad for performances. Naming scheme applies at java source code level.
- **dynamic**. They are made of one class coded in an *ad-hoc* way. The class must implement the JMX `DynamicMBean` interface. That results in three general methods to code: one getter and one setter common to **all** Mbean attributes and one method to forward any `MBean` operation. It is up to the programmer to code in each of those three methods the proper dispatch mechanism accordingly to attributes or actions names. Naming scheme applies at java source code level.
- **model**. `Model MBeans` are little different: there is no class to implement. In this case, test code is reduced to - the creation of one `Model MBean` descriptors and then -the feeding of `Model MBean` support with this descriptor to get pseudo instantiation of the wanted `MBean`. A basic support for Model MBean is mandatory by [SUN02][`RequiredModelMBean` page 72], therefore our tests use the one shipped with each of the tested JMX implementations (see § 2.4.1).

[1] `http://jakarta.apache.org/bcel/index.html`.

Remark 2: [SUN02][pages 61–70] is also introducing the *Open MBeans*. They are basically `Dynamic MBean`. *Open MBeans* differ from regular `Dynamic` ones only by the data types (java classes) used to describe attributes, operation parameters and operation results. As we do not intend to make data types a factor (see § 2.5.6) we do not dissociate these two cases.

To sum-up, we define the following factor name and its set of possible values:
Factor *Types of the* `MBean`

$$TypeMBean = \{std, dyn, model\} \tag{2}$$

2.4.3 Number of attributes

One can suspect that a large number of attributes in the same `MBean` server may introduce performance problems. So we introduce the following integer factor:

Factor *Number of attributes per MBean*

$$NbAttr \in [1..maxAttributes] \tag{3}$$

Tests suite user can choose $maxAttributes$ at will.

2.4.4 Number of mbeans

One can suspect that a large number of `MBeans` in the same Mbean server may raise performance problems. So we introduce the following integer factor:

Factor *Number of* `MBeans` *per* `MBean` *server*

$$NbMbean \in [1..maxMbean] \tag{4}$$

Tests suite user can choose $maxMbean$ at will.

2.4.5 Number of request injected per second

Indeed this factor is also a workload parameter. To test the *get* service it is necessary to be able to vary injection rate if one needs to measure the knee capacity or maximum capacity of

one MBean server. So we introduce the following factor:

Factor *Request injection rate (actual injected get requests per second)*

$$InjectionRate \in [0..maxInjectRate] \tag{5}$$

Tests suite user can choose $maxInjectRate$ at will. We have conducted tests up to 1000 requests per second as this value is higher than any maximum capacity of any of our test configurations. I have used a 100 requests per second step to build series.

2.5 Parameters

Test parameters are characteristics what have effect on performances but than can not be explicitly changed in the test suite. A parameter can be promote to a factor after analysis of some measures. One can easily figure out the parameters detailed in the next section.

When running measures one must be sure than no parameter varies, or in other words that experimental conditions are really stable from one measure to another.

2.5.1 Kind of remote connectivity and related parameters

Only RMI/JRMP connectors (mx4j proprietory) or jsr160 conformant [SUN03][chapter 4] are used. This is no a major restriction: JSR160 for RMI/JRMP is commonly available and used (RMI connector, over JRMP or IIOP, is mandatory by [SUN03][page 33]). Rmi/JRMP have various parameters (i.e: time out for TCP connections management). We have conducted tests with default sun jdk1.4 JVM configuration. One can find details about properties of sun RMI implementation at: `http://java.sun.com/j2se/1.3/docs/guide/rmi/sunrmiproperties.html`.

RMI/JRMP uses TCP as transport layer. So parameters (factors indeed) as the *number of TCP connections/overall injected throughput* ratio could have a major impact on tests. This ratio can vary then an important workload (*injectionRate* has to be supported by several processus on several nodes. The system parameters as the TCP window size can also have some impact. One can refer to [MR01] for an extended discussion about RMI performance and configuration.

2.5.2 Overall hardware, operating system

Obviously node parameters like:

- Architecture type
- Memory Size
- Type of O/I devices

- Operating System
- Overhead introduced by basic system services
- ...

have direct effects on performances. We do not intend here to make an exhaustive list and a finer classification. We repeat measures in a stable hardware and system environment.

2.5.3 Java Virtual Machine

The vendor of the JVM used to support measures and its run-time parameters form a large set of parameters. There is a typical issue about objects heap size at run-time.

For the moment all our measures have been done using basic sun jdk 1.4.2 HotSpot **client** JVM (see the `-client` option for the `java` command).

2.5.4 Java compilation and libraries packaging

All the java support (jdk) and libraries under tests (jmx support) are the one coming from the original sites or with Linux packages in a precompiled format.

Local recompilations of:

- native code level for jdk implementation (or other jvm)
- java code of jmx support with peculiar java options

can have some effect on overall performances.

2.5.5 Network

We have conducted our test on a nearly "isolated" switched 100baseTX Ethernet network. So we assume that network is not a bottleneck.

2.5.6 Data types

The data type a JMX get retries is a simple test parameter. We generate tests with simple `Strings` with a length inferior to 10 characters.

2.5.7 Service interaction complexity

Complexity of the JMX interaction: mainly the numbers of requested attributes in one service interaction is set to one. This is an arbitrary choice for this parameter.

2.5.8 Service invokation and proxy

All call (`get`) are done using directly the to `MBeanServer`.

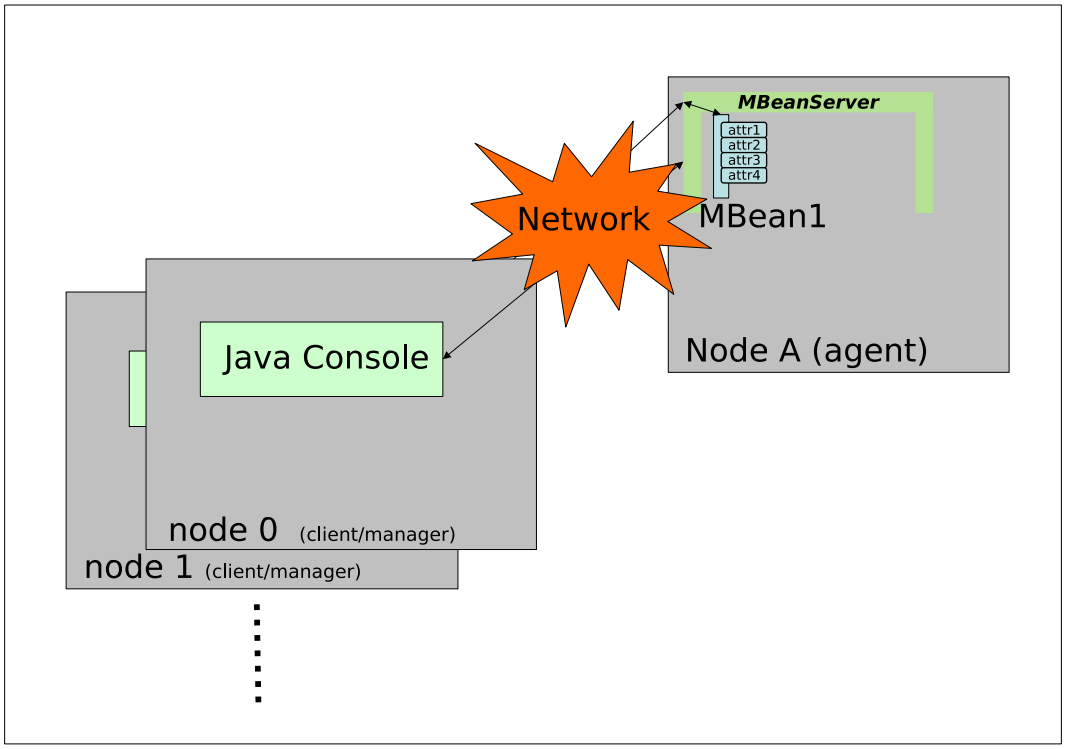


Figure 1: Many managers/one agent

3 Tests suite architecture and organization

3.1 What is a test ?

To obtain a certain visualization of a (or several) metric for some fixed test factors one must first make a serif of unitary measures or *runs* in a stable environment (test parameters must not vary). Usually several runs using the same factors are analyzed (one could find a discussion about how to do this analysis in [Jai91][chapter 12]) to get one result, the result of the test. One factor can be varied and then a series of tests is achieved.

Afterward, the series must be graphically displayed to allow analysis an interpretation.

3.2 Simple get polling scenario

The test provides a simple test scenario: one (or several) manager (client) sends get requests to one JMX agent. Manager and agent are configurable to cover all factors ranges described in (see section 2.4). Figure 1 gives an idea of the overall test organization.

In this scenario the manager is a simple get request injector using a given rate. It generates a (local) log file containing informations as: generated requests according to time, completed requests according to time, requests latency according to time.

A log of cpu, memory, network usage is generated on agent node using the SAR deamon [God04].

The execution of one measure is coordinated by a test "console" which:

1. Creates and starts a SAR deamon on agent node.
2. Starts one agent under test.
3. Creates one or several clients (injectors).
4. Starts requests injection.
5. Stops requests injection. Time between start and stop is the test duration.
6. Collects log files (SAR's logs and injectors logs).
7. Launches scripts (or commercial tools) to process logs files and get basic values for simple metrics, average of correct responses per second by example. This last step is a "post-mortem" analysis. This step feeds the last one: visualization of results.
8. Cleans up all processes and temporary files.

Figure 2 sums up these steps.

4 Load injectors

In this section we shortly describe how the manager (client) is designed to generate injection for the simple get polling scenario. We do not indent here to explain how to modify or extend this manager. It is configurable to support all manager related factors described in section section 2.4. This configuration uses java properties. Values (i.e TCP port number, number of MBeans, ...), and factory class names as well can be defined via these properties. One can take a look at `org.objectweb.jmxbench.clients.ClientProperties` class to get an overview of all available manager properties.

4.1 Load injection issues

The most important test factor supported by a load injector is the injection rate (see section 2.4.5). This test factor is obviously the base to get measures where the *throughput* (see section 2.3) metric applies. One must be careful that the wanted number of requests per second is really injected at the SUT (the JMX agent) border. The load injector must have enough ressources (cpu, memory, network bandwidth) to fulfill the injection. Currently in our test suite the only mecanism to ensure what one injector is running as expected is to use

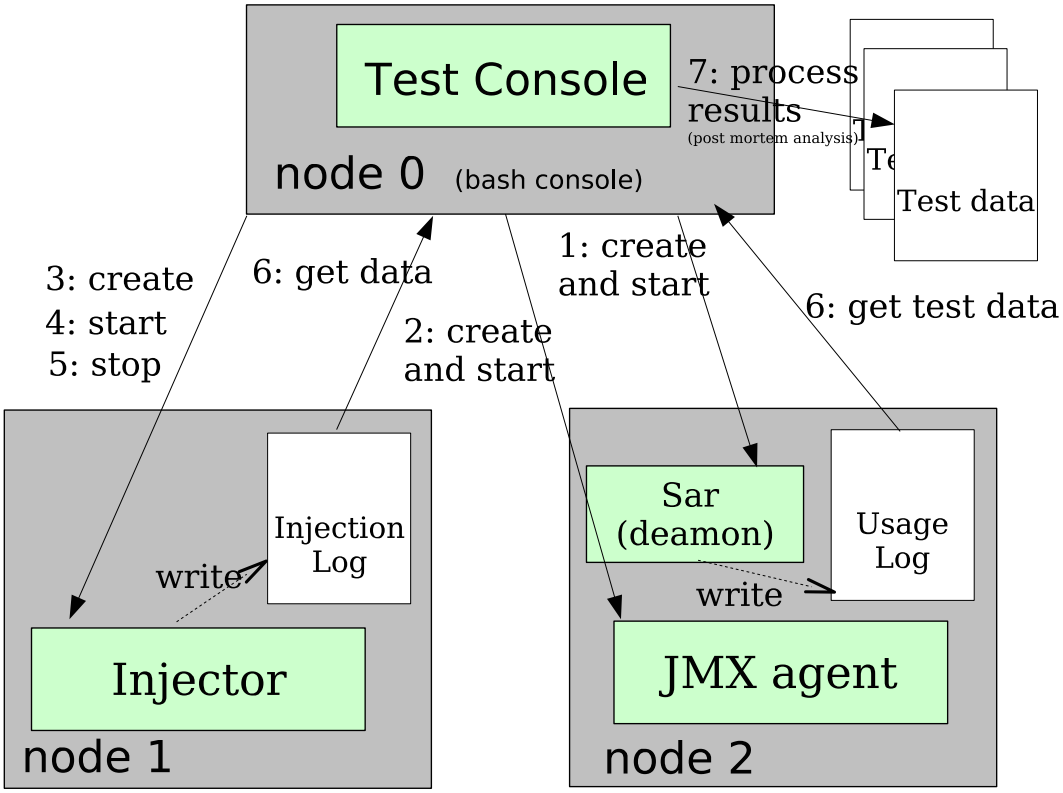


Figure 2: Test steps

SAR on the injection node. It is **up to the user** to check if the injector runs in a proper area of cpu, memory... It is also up to the user to set up a test bed where the network does not introduce bottleneck between injector(s) and SUT. Typically the test bed can be made of nodes connected by a 100 Mbps full duplex Ethernet switch.

To achieve a target request injection rate without been obliged to use too much client injectors, injectors must be coded in a proper way. Actually a JMX get request is a true client/server request at the JMX API level. From a client point of view such a request is a simple **synchronous** remote invocation to a remote `MBeanServer`'s method (`MBeanServerConnection.getAttribute`). Let us defined *rtd* as the actual round trip delay for one invocation (so expressed in *seconds/request*). In the case of sequential invocations, $InjectionRate \times rtd$ (which has no dimension $(requests/second) \times (seconds/request)$) must stay under 1. On in simpler words $InjectionRate$ is limited to $1/rtd$ $requests/second$ in sequential way.

To overcome this limitation the obvious way is to use threads. After a couple of tries we have adopte a simple scheme: one thread injects requests at a one request per second rate (we assume that *rtd* is smaller that 1 second in an operational environment), therefore the number of instanced threads corresponds to the target $InjectionRate$ factor. Figure 3 depicts this idea.

The main reason to do so is the lack of resolution of the `sleep` call in java. Indeed this call has some drawbacks:

1. Even if the parameter is a milliseconds value (even nanoseconds !), the minimum actual duration that `sleep` can ensure is quite higher than one millisecond and depends on underlaying infrastructure (JVM, Operating System, hardware).

2. Even if the parameter given to `sleep` is a realistic value, the actual duration can be a little different that the expected one.

3. With the same parameter the actual duration can vary between two calls of `sleep`. These fluctuations can not be neglected.

Some other ways to get a kind of timer (`Thread.wait`, `Thread.join` java methods) do not give better results.

One can take a glance at small test programs into `org.objectweb.jmxbench.utilities.testsleep` package which come with the test suite distribution. `Test2` program can help to figure out how a real system behaves face these problems. `Test1` program gives an idea of the `System.currentTimeMillis` method resolution (usually around 1 millisecond).

So there is **no synchronization** between injectors located on separated computers (nodes) nor between injection threads within the same injector in our injection scheme. A workload is mainly constituted by a rather long steady injection period (more than twenty minutes see following section 4.2). This simple way to realize load injection is good enough to ensure a target $InjectionRate$ factor value in our context. Injectors logs enable us to evaluate the gap between a target $InjectionRate$ and the actual average value.

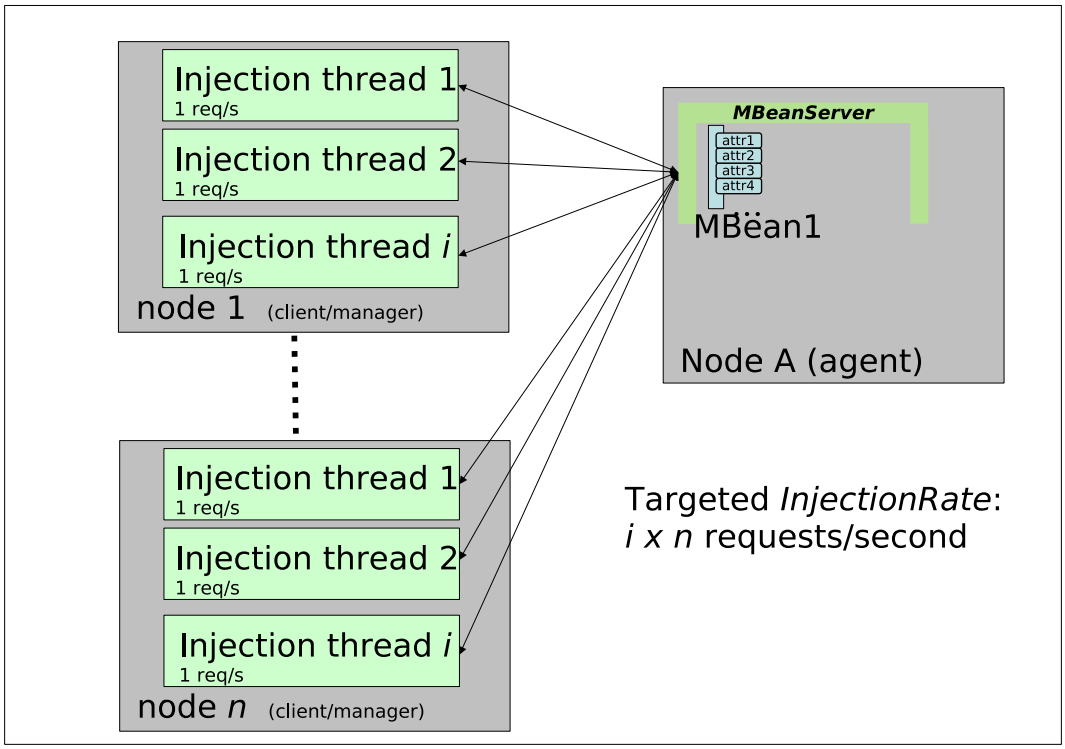


Figure 3: Injectors, injectors threads

4.2 Ramp-up

Traditionally load injection processus is split in three phases as explained in [ACC+02, CMZ02, CCE+03]:

1. a ramp-up or warm-up phase. This phase allows to start injection until it reaches a steady-state throughput level (target *InjectionRate* factor in our context) without overflooding injectors or SUT.

2. a measurement phase where injection rate is maintained at its target value and where all measures logs are really taken into account.

3. a ramp-down phase where injection rate is still maintained at its target value but where measures logs are neglected. The idea is to avoid measuring the ultimate phase when injectors and SUT are stopping. For example this ramp-down phase avoids measuring unachieved requests due to agent shut-down.

In our case the `jmxbench_client_upRamptime` client property defines the ramp-up duration for injector. A pseudo linear injection ramp-up is achieved by starting one injection thread (see section 4.1) every:

$$\frac{\texttt{jmxbench\_client\_upRamptime}}{(InjectionRate/Number\,of\,requests\,injected\,by\,a\,thread\,per\,second)}$$

As we have chosen 1 *request/second* for the *Number of requests injected by a thread per second* (see section 4.1 again), we simply start one thread each `jmxbench_client_upRamptime`/*InjectionRate* second.

Figure 4 depicts the ideal effect of this ramp-up approach for injection rate evolution during a test.

Obviously, to reach this situation the start duration allocated for one injection thread must be long enough to allow:

- its creation and initialization ;
- injection of its first request.

One can guest than the `jmxbench_client_upRamptime` property must grow accordingly to the *InjectionRate* factor as the number of threads grows accordingly to this factor.

5 Agent under test

In this section we shortly describe how the agent is designed for the simple get polling scenario. We do not indent here to explain how to modify or extend this agent. This agent is configurable to support all agent related factors described in section section 2.4. This configuration uses java properties. Values (i.e TCP port number, number of MBeans, ...),

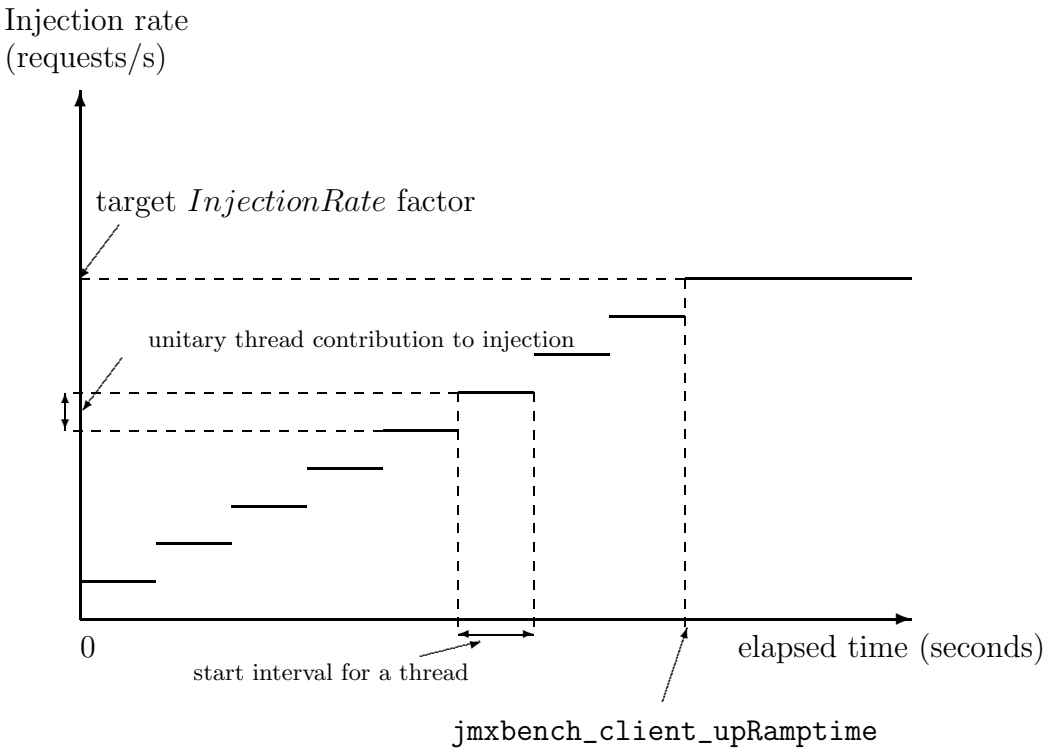


Figure 4: Ramp-up effect on injection rate

and factory class names as well can be defined via these properties. One can take a look at `org.objectweb.jmxbench.agents.AgentProperties` to get an overview of all available agent properties.

5.1 Support for $TypeJmx$ factor

At the java source level everything has been done to get test code independent from the JMX implementation under test. For the MBeans under test the independence is strict there is no condition as "*if mxj4 then ...*" in code. At agent level the only point to handle is the kind of connector used between manager and agent. All peculiar code related to connector creation must be keep in a class implementing the `org.objectweb.jmxbench.agents.ConnectorMaker` interface. The name of such a class is passed to the agent into the agent `jmxbench_agent_connectormakerclassname` property. Two concrete classes are provided within the jmx test suite:
`org.objectweb.jmxbench.agents.connectors.jsr160rmiJsr160RmiConnectorMaker` for connector initialization conformant to JSR160 and
`org.objectweb.jmxbench.jmxbench.agents.connectors.mx4j.Mx4jConnectorMaker` for connector initialization of pre jsr160 version of Mx4j.

Obviously the agent under test must load the classes (jarfiles) of the implementation under test and all other supporting libraries. To achieve this in an simple but bulletproof way: we simply copy all necessary jarfiles in a unique run time directory (`lib` at the current directory where agent is started) and build a java classpath from it. So the BCEL option of the $mx4j\_jsr160\_bcel$ factor is achieved by copying or not copying the BCEL jarfile in this runtime `lib` directory.

5.2 Support for $TypeMBean$ and $NbAttr$ factors

Instantiation and registration of one MBean in the agent under test is made abstract by defining the `org.objectweb.jmxbench.agents.mbeanservers.MBeansMaker` java interface. The concrete factory classes are using the `jmxbench_agent_numberofattributes` agent property which is indeed the $NbAttr$ factor at runtime.

5.2.1 *std* and *dyn* cases

Instanciable standard `MBeans` comes from concrete java classes which implement a dedicated interface. A name convention links the `MBean` class and is interface (see [SUN02][pages 36–37]). Instanciable dynamic MBean comes from a concrete java class which must implement the conventional JMX `DynamicMBean` interface (see [SUN02][pages 40–41]). If such a class is available the factory
`org.objectweb.jmxbench.agents.mbeanservers.UniqueMBeanMakerFromClass` implementing the `MBeansMaker` interface instanciates one MBean from a class name given in the `jmxbench_agent_mbeanservers_UniqueMBeanMakerFromClass_mbeanclassname` agent property.

Support of *NbAttr* is a little peculiar for these two cases. Indeed such `MBean` classes are supposed to be coded at source level by a programmer, then compiled, and loaded at runtime. But in our case we want be able to run series of tests by varying the *NbAttr* factor. To remplace the "human programmer" we are using the "Java-based template engine" Velocity[2]. The templates supporting this generation are located in `$JMXBENCHROOT/agents/mbeanservers/onembean/std` and `$JMXBENCHROOT/agents/mbeanservers/onembean/dyn` directories. They allow to generate a `Mbean` with a chosen number of `String` attributes. Each JMX attribute is supported by a simple instance variable but is not link to a real "managed resource". The `$JMXBENCHROOT/java/build.xml` for the `ant`[Fou05] utility features two targets: *generateonembeanstd* and *generateonembeandyn* to generate and compile java files for the corresponding cases. The wanted number of attributes (factor) is given as an `ant` property (`-D` option, or in an `ant` properties file) at generation and compile time.

5.2.2 *model* case

This case is simpler. There no class or interface to produce. A model `MBean` is instanciate by filling a descriptor and then giving this descriptor to the mandatory JMX support to get instanciation (see [SUN02][pages 71–72]. The `org.objectweb.jmxbench.agents.mbeanservers.UniqueModelMBeanMaker` factory initializes such a descriptor, accordingly to the value of *NbAttr* factor and create one model `MBean`. In this this factor is implemented at run-time by `jmxbench_agent_numberofattributes` agent's property.

5.3 Support for *NbMbean*

To get the target number of MBeans an unitary factory is simply called several times. This factor is given to the agent via the `jmxbench_agent_numberofmbeans` property. Unitary factories as described in paragraphs 5.2.1 and 5.2.2 have a proper factory instance variable to generate different names for all these `MBean`.

6 Console, Tests starting and execution

The test console (see figure 2) is a simple bash script (`$JMXBENCHROOT/scripts/tests/getinjector/start-console.sh`). All test factors values and test informations as test duration, ramp-up duration, nodes names for injectors and agent are grouped in a text file. Such a test configuration file is given as a parameter to the console. All processus (SAR monitoring deamons, agent under test, injectors) are started using "remote shell" (`rsh`) together with some helper scripts. All test data files (SAR logs, injectors logs) are collected to a unique directory ("report directory") using "remote copy" (`rcp`). This report directory is specified in the test configuration file.

[2] `http://jakarta.apache.org/velocity`.

This choice for simple scripts has been mainly taken for portability issues: bash, rsh, rcp are widely avalaible on most unix systems. It is quite probable that rsh, rcp must be replaced by their secured (ssl) equivalents.
Remark 3: Some (perl) scripts has been writen to help user generate series of configuration files to reflect the variation of one factor.

7 *Post-mortem* analysis

Post processing of unitary measures (runs) a made using perl. This choice is still quite portable and anyway only one node has to be set-up to support this step. Indeed, results generated during the previous step can be moved to this node.

This analysis can have a large range of targets according to the wanted visualization. For now we just provide merge of individual runs to get one result per metric and per factor set and then visualization for series using a varying *InjectionRate* factor. Currently this visualization is generating HTML pages with graphics. Graphics are created using the gnuplot free software [WK05].
Remark 4:
A discution could stand at this point: how runs for a same factors set are merge or sumarized ? Guideline to choose the right method can be found in [Jai91]pages 187–200. In our case, for a small number of runs it seems than simple (arithmetic) mean is correct.

Part II
User manual

8 System and hardware requirements

8.1 System and sofwares requirements

The suite has been test under various linux distributions (debian, mandrake).

The following softwares must be available:

- **SAR** deamon [God04]
- **perl** (usually coming with basic set-up for most Linux distributions). Perl is used for *post-mortem* analysis, so this software should only be avalaible on the computer where test data are analysed.
- **jdk 1.4** (the test suite has not been tested under jdk 1.5). See: `http://java.sun.com/j2se/1.4.2/index.jsp`.

- **ant** the make like utility for java. See: `http://ant.apache.org`.
- **rsh** is available on each test nodes and correctly configurated to allow access to the unix user which runs the tests (`start_console.sh` in the following).

8.2 Getting the suite sources

Sources are avalaible in two different ways:

- using cvs (anymous access):
  ```
  cvs -d :pserver:cvs@cvs-sop.inria.fr:/CVS/NetworkMngtbenches
        co -ramarillo jmx-bench
  ```
  This command will create a `jmx-bench` in the current directory, it will be the `$JMXBENCHROOT` directory.
- getting the archive file at: `http://madynes.loria.fr/jmx-bench/jmx-bench-amarillo.tgz` or `http://potiron.loria.fr/projects/madynes/jmx-benches/tar` and untaring (`tar xvzf jmx-bench-amarillo.tgz`) anywhere wanted. The extraction will create a top level `jmx-bench` directory what will constitute the `$JMXBENCHROOT` directory described in this document.

8.3 Shell configuration and user rights on test nodes

The test suite relies upon `bash`. Some configurations have to be done on any test nodes except the console. The console (scripts: `start_remote_agent.sh`, `start_remote_client.sh`) starts processes on nodes (agent, injection) using `rsh` so a couple of environment variables must be set on each nodes:

1. `PATH`

Usually those variables are set and export in the file: `.bash_profile` in the home directory of the user running the tests.

9 Running a test, getting a measure

9.1 A full example

We detail here how to run the *getinjector* workload. This is a two steps process using a couple of bash and perl scripts.

9.1.1 Step one: running one injection

To get **one measure** one must follow these points:

1. change directory to `$JMXBENCHROOT/scripts/tests/getinjector` ;
2. edit the test configuration file which can be anywhere. Let's say: `$USERHOME/mytest.cfg`. User must define here values for the test factors (see section 2.4) and other tests characteristics (test duration, ramp-up duration, ...). This text configuration file has a simple format close to java properties files.

   a commented sample test configuration file is available as: `$JMXBENCHROOT/tests/getinjector/test-example.cfg`. Test configuration files are very simple properties files.
3. start the test by typing start_console.sh <*path to configuration file*>. The script `start_console.sh` does all the job accordingly to properties specified in the configuration file given as parameter. The console:
   - starts agent under test on node specified in `jmxbench_console_agentnode` property ;
   - starts one or several injectors on nodes specified in `jmxbench_console_clientnodes` property ;
   - starts the resource monitor (SAR) on each node (agent and injectors nodes) ;
   - waits for the test duration specified in configuration file (`jmxbench_client_upRamptime` + `jmxbench_client_sessionRuntime`) milliseconds (see section 4.2 for ramp-up discussion);
   - stops all process's: agent, injectors, SAR...
4. all test data locally generated on nodes are copied to the directory specified in `console_mainReportDir` property:
   - a test directory is created in the console directory using the date/time of test launching ;
   - a sub directory `CONFIG` is created and a file equivalent to the test configuration file
   - in this test directory the SAR log of the agent and the injector(s) are copied (`Agent-`<*agent node name*>, `Client-`<*client node name 1*>, ...) are copied ;
   - the injector response logs: requests latency vs. time: Client-<*client node name 1*>.rspt.dat... and injector number of completed requests vs. time and approximation and the numbers of requests per second vs. (the derived of the first column): `Client-`<*client node name 1*>.cmpt.dat ;
   - `Client-`<*client node name 1*>`.xml` file gives a sum-up of all the measure data for each injection node.

5. the last point can be replay to get:
   - several measures for a given test factors set (test configuration file is not changed)
   - other measures with a changed factor

9.1.2 Step two: *Post-mortem* and visualization

Post-mortem analysis is made using several perl scripts:

1. generate-final-result.pl <*path to a main report directory*>: to process result of several runs. Run with the same factors set are grouped using arithmetic mean (see section 7). As a result we obtain 5 files, one for each metrics in: cpu usage, memory usage, transmitted bytes, received bytes, number of correctly served request per second. Those files are simple text array which can be visualized to plot a metric against *InjectionRate* (see section 2.4.5) factor.

2. The script generatereport.pl generates html files with graphics to visualize a metric against time for one unitary run. This visualization is given as a tool to detect incorrect runs.

Remark 5: One must note:

1. `nodeDir` property must not point to a shared directory (NFS), node name is not added when building temporary directories names. Only one injector (client) can run on one node for similar reasons. But, one agent and one injector can stay on the same node.

2. The console logs its message in `$JMXBENCHROOT/scripts/tests/getinjector/console.log` ;

10 Other notes

10.1 Using eclipse for coding injector and agents

Eclipse[3] (2 or 3) can be used to edit, compile and manage jmxbench sources.

To compile (not run) jmxbench sources under Eclipse environment some jarfiles must be imported into the Eclipse project (Project Name→Properties→Java build path →Libraries tab). For jarfiles coming with jmxbench distribution you have to do a simple import ("Add JARS"), but others external jarfiles you have to use "Add external JARS"

- set the default source folder to `$JMXBENCHROOT/java/src`. Use Project Name→Properties→Source tab →Edit.

[3] www.eclipse.orf

- set the default output folder to `$JMXBENCHROOT/bin`. Use PROJECT NAME→PROPERTIES→SOURCE TAB.
- For the JMX definitions (`javax.management`) : import `$JMXBENCHROOT/java/external/jmximplementations/sunri/v.../jmxri.jar` and `$JMXBENCHROOT/java/external/jmximplementations/sunri/v.../jmxremote.jar` for JSR160 related packages.
- To be able to compile pre-jsr160 adaptor for MX4J, import `$JMXBENCHROOT/java/external/jmximplementations/mx4j/v.1.1.1/mx4j-tools.jar` as project's jarfile.
- To compile ant task for Velocity (a macro processor used to generate some source level `MBeans` (standard, dynamic)): add ant core support by importing `ant.jar` as an **external** jar file because ant distribution is not included into jmxbench. The location of this file jarfile depends of your ant installation or package manager. You also need to add the `$JMXBENCHROOT/java/external/benchgeneration/velocity-core-1.3.1.jar` and `$JMXBENCHROOT/java/external/benchgeneration/commons-collections.jar`

References

[ACC+02] Cristiana Amza, Emmanuel Cecchet, Anupam Chandaand, Alan Coxand, Sameh Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani, and Zwaenepoel Willy. Specification and implementation of dynamic web site benchmarks. Technical report, WWC-5, Rice University, TX, USA, november 2002. http://www.cs.rice.edu/CS/Systems/DynaServer/wwc5-spec_dyna_bench.pdf.

[Bur04] Mark Burgess. *Analytical Network and System Admistration*. WILEY, 2004. ISBN : 0-470-86100-2.

[CCE+03] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *4th ACM/IFIP/USENIX International Middleware Conference*, juin 2003. http://rubis.objectweb.org/download/Middleware-2003.pdf.

[CMZ02] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of ejb applications. In *Oopsla'02*, 2002. http://rubis.objectweb.org/download/perf_scalability_ejb.pdf.

[Fou05] The Apache Software Foundation. Apache ant 1.6.5 mannual. http://ant.apache.org/manual/index.html, 2005.

[God04] S. Godard. Sar manual page. see: `http://perso.wanadoo.fr/sebastien.godard/use_sar.html`, february 2004.

[Jai91] Raj Jain. *The art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc, 1991. ISBN : 0-471-50336-3.

[JW02] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on parallel and distributed systems*, 11(6):589–603, june 2002.

[LAF05] Abdelkader Lahmadi, Laurent Andrey, and Olivier Festor. Performances et résistance au facteur d'échelle d'un agent de supervision basé sur jmx : Méthodologie et premiers résultats. In *Colloque GRES 2005 : Gestion de REseaux et de Services, Luchon, France*, volume 6, pages 269–282, mars 2005. ISBN : 2-9520326-5-3.

[MR01] Ashok Mathew and Mark Roulo. Accelerate your rmi programming- speed up performance bottlenecks created by rmi. `//http://www.javaworld.com/jw-09-2001/jw-0907-rmi.html`, september 2001.

[Pat01] Collin Pattinson. A study of the behaviour of the simple network management protocol. In *DSOM (International Workshop on Distributed Systems: Operations & Management)*, October 2001. see: `http://www.loria.fr/~festor/DSOM2001/proceedings/S9-2.pdf`.

[SUN02] SUN. Java^{TM} management extensions, instrumentation and agent specification, v1.2. http://jcp.org/en/jsr/detail?id=3, october 2002. Maintenance Release 2.

[SUN03] SUN. Java^{TM} management extensions(jmx™) remote api 1.0 specification. http://www.jcp.org/en/jsr/detail?id=160, october 2003. Final Release.

[WK05] Thomas Williams and Colin Kelley. gnuplot - an interactive plotting program. http://www.gnuplot.info/docs/gnuplot.pdf, 2005.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803