Implementing Probes for J2EE Cluster Monitoring

Emmanuel Cecchet, Hazem Elmeleegy, Oussama Layaida, Vivien Quéma

LSR-IMAG Laboratory (CNRS, INPG, UJF) - INRIA
INRIA Rhône-Alpes, 655 av. de l'Europe, 38334 Saint-Ismier Cedex, France
First.Last@inrialpes.fr

1 Introduction

Clusters have become the de facto platform for large J2EE application servers. In production environments, it is necessary to constantly monitor the state of the system to detect failures or performance degradations that may lead to violations of Service Level Agreements. The LeWYS project (http://lewys.objectweb.org) is an open source initiative aiming at building such monitoring infrastructure. It relies on Julia, a Java implementation of the Fractal component model. In this paper, we focus on the implementation of efficient probes in Java that reifies the state of various J2EE cluster components with a controllable intrusiveness. We first describe in section 2 the overall architecture of the LeWYS framework. Then, section 3 presents the design and implementation of hardware and operating system probes for Linux and Windows. J2EE specific components are probed using the JMX probe introduced in section 4. Section 5 provides a performance evaluation and section 6 discusses related work. Section 7 concludes the paper.

2 The LeWYS framework

LeWYS (*LeWYS is Watching Your System*) is a component-based framework for distributed system monitoring. LeWYS is based on Fractal [1], a Java-based component model featuring hierarchical composition, component sharing and component binding. LeWYS allows building different kinds of monitoring systems, for various configurations of distributed systems.

Figure 1 depicts the main components commonly found in a monitoring system built using LeWYS: *observers* receive monitoring data from a set of monitored nodes. A *monitoring pump* is deployed on each monitored node. Monitoring pumps allow observers to subscribe to different *probes* that are in charge of collecting monitoring data on both hardware and software resources. The monitoring pump timestamps monitoring data and sends them to observers using dynamic *event channels*. Event channels are made of communication components (e.g. marshallers, TCP sockets) and processing components (e.g. filters, aggregators). Note that LeWYS provides a particular observer, called *monitoring repository*, that store monitoring data in databases. Data can be retrievied from the repository and processed later (e.g. to correlate various indicators).

3 Hardware and OS probes

3.1 Linux probes

Linux, as many other Unix operating systems, reifies the kernel state by the way of a virtual filesystem directly mapped onto its in-memory data structures. This filesystem is commonly mounted under `/proc` and is referenced as the `/proc` filesystem.

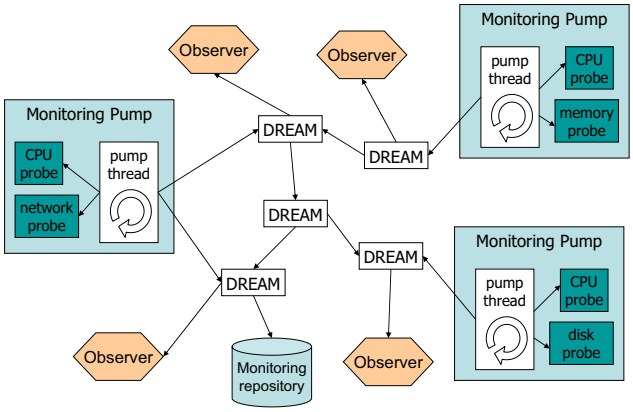

Fig. 1. Architecture of a monitoring system built with LeWYS.

[2] describes the benefits of several optimizations to gather information from the `/proc` filesystem. All LeWYS Linux probes follow the same design. When the probe is loaded, the relevant files of the /proc filesystem are parsed to find the list of available resources and the indices of the values within the files. The various optimization includes keeping the file descriptor open between reads, reading the /proc files as a whole block (values are regenerated on each access) and processing raw bytes using pre-computed indices without conversion through UTF strings.

We have implemented Linux specific probes that report monitoring data on the CPU consumption (user, nice, kernel, idle), on the memory (total, used, free, shared, cached, etc.), on disks (issued reads and write, merged reads and writes, read and write sectors, etc.), on the netwok (received bytes, received packets, collisions, etc.), and on the kernel (interrupts, context switches, etc.).

Note that cpu information is reported for each processor in the system and network usage is reported for each network interface handled by the kernel. Most information is retrieved from few files (one file per probe except for the memory probe) but units remain resource specific.

Resources that report a monotonically increasing value will require further processing to compute throughput. For example, network throughput can be computed from 2 samples (s1 and s2) of *bytes received* and *bytes transmit* divided by the time elapsed between the 2 samples:

$$throughput = \frac{(s2_{bytes\_received} + s2_{bytes\_transmit}) - (s1_{bytes\_received} + s1_{bytes\_transmit})}{s2_{timestamp} - s1_{timestamp}}$$

3.2 Windows probes

Windows provides a performance monitoring infrastructure ensuring that applications have up-to-date information on performance. Performance data concerns: (i) physical components, such as processors, disks, and memory, (ii) system objects, such as processes, threads and interrupts, and (iii) application-level services that use Windows mechanisms to provide information about them. Performance data is structured in three hierarchy levels called *objects*, *instances*, and *counters*:

– An *object* refers to any resource, service or application providing performance data. Examples of performance objects are disk, processor and memory.

– An *instance* is a unique copy of a particular object type relative to a sub-part of this object. For example, the network object may have an instance for each available network interface.
– Each object (or instance) has several *counters* that are used to measure various metrics. For instance, metrics of a network *instance* are: bandwidth utilization, packet-loss ratio, number of active connections and so on.

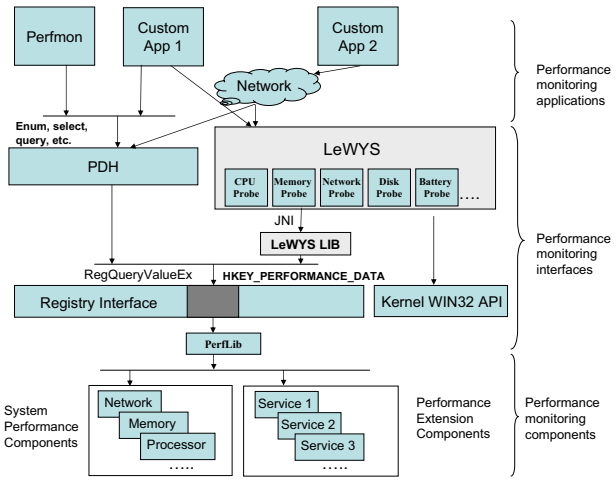

Fig. 2. Overview of performance monitoring structure in Windows.

As shown on Figure 2, several system performance components are responsible of system and resource monitoring. On top of these components, Windows provides a registry interface to collect performance data using the Windows registry API. The data is not stored in the registry database; instead, querying the registry key HKEY_PERFORMANCE_DATA causes the system to collect the data from the appropriate performance components. The system returns a uniform data structure containing the requested set of performance data, structured as objects, instances and counters values.

Based on this interface, LeWYS implements Windows specific probes used to gather all performance data made available by the system. On the registry interface, a native LeWYS library (LeWYS LIB on figure 2) provides generic routines to collect performance information, which are then made accessible to LeWYS probes through the Java Native Interface (JNI).

4 JMX probes

The purpose of JMX [3] probes is to collect monitoring information about the software applications running in J2EE environments. Unlike other probe types, this information can not be obtained from the underlying operating system, but rather from the applications themselves. The JMX architecture is basically a 3-tier architecture, in which the back-end tier (known as the instrumentation level) represents the managed applications; the middle tier (or the agent level) is constituted by the JMX agent

(including the MBeans server and the services it provides); while the front-end tier (referred to as the distributed services level) represents the JMX clients and their communication channels with the agent.

Our JMX probes represent the clients in the architecture described above. Each JMX probe is responsible for a single JMX agent. During its initialization, it inquires from the agent about all the MBeans registered in it, together with their attributes. JMX probes use RMI connectors to communicate with the agent, so they can be deployed either on the same machine as the agent, or on a remote machine. Currently, LeWYS incorporates both the generic JMX probe and a specialized probe that is responsible for monitoring the C-JDBC [4] middleware, as its JMX interface is based on retrieving values through special operations rather than getter methods.

5 Performance

5.1 Linux probe performance

Table 2 reports the average time needed to collect resource usage for each LeWYS Linux probe. The measurements were performed using Sun JVM 1.4.2 on a Pentium4 1.8GHz machine, 512 MB RAM, a 40GB IDE disk with 6 partitions, and running Linux kernel 2.4.20.

Probe	**Number of resources**	**Average time to probe 1 resource**	**Average time to probe all resources**
CPU	8	22.9μs	23.4μs
Memory	13	40.3μs	40.7μs
Disk	66	30.4μs	31.5μs
Network	48	25.3μs	27.8μs
Kernel	3	23.0μs	23.0μs

Table 1. . LeWYS Linux probes collecting time

The dominant cost is reading the /proc filesystem and we perform this read only once regardless of the number of resources to probe. Therefore, there is few difference between gathering the values of all resources or only one. The gathering cost ranges from 23 to 41μs to probe all resources of a probe. The memory probe costs more than the other probes because it has to collect information from 2 different /proc files. The probe cost allows for a single thread to gather all hardware information of a node in 146μs.

5.2 Windows probe performance

Table 2 reports the average time needed to collect resource usage for each LeWYS Windows probe. The measurements were performed using Sun JVM 1.4.2 on a Pentium4 2 GHz machine, 512 MB RAM, a 40GB IDE disk with 2 partitions, and running Windows 2000.

The cost to collect one resource varies from 332 μs for memory probe to 12093 μs for network probe and is relatively high compared to Linux probes. This cost is mainly due to JNI calls but also to the access to performance data through the registry interface. However, the variation between different probes is due to the fact that some

Probe	Number of resources	Average time to probe 1 resource	Average time to probe all resources
CPU	10	661 μs	725 μs
Memory	9	332.8 μs	354.7 μs
Disk	42	1939.1 μs	1956.2 μs
Network	32	12093 μs	12265 μs
Kernel	9	1184.4μs	1251.6μs

Table 2. . LeWYS Linux probes collecting time

performance components in Windows (e.g. network) are costly in terms of processing and memory requirements. We also observe a slight difference between probing one resource or all of them, and this thanks to some optimization features applied in LeWYS. Indeed, because Windows provides thousands of performance objects, retrieving all object data is too costly in terms of processor and memory requirements. Experiments made in Windows 2000 have revealed that retrieving the whole performance data generates more than 95 Kilobytes of data. This is due to the fact that querying the performance registry key causes the system to collect data from all monitoring components, which is costly. LeWYS reduces the cost of native monitoring routines by minimizing access to the registry. Only the required subset of information is required from the registry. For instance, the CPU probe retrieves only counter values of the processor object, which required less than 700 bytes of data. Probes requiring several objects are grouped within the same request so that one request is made on the registry.

6 Related work

Closely related to our work, the WatchTower [5] system provides a monitoring framework specific to Windows. WatchTower builds upon PDH (Performance Data Helper), an abstract API that hides the complexity of the registry interface. Although this interface gives access to all performance data, it is oriented more towards operations on a single counter rather than groups of counters (each counter value requires an individual access to the registry). Using this interface to collect a large amount of performance information would be too costly. In LeWYS, this cost is reduced by accessing the raw registry interface in an intelligent way.

The JMX specification [3] defines a monitoring service that does not match the requirements of our framework, which aims at capturing the complete behavior of the monitored resources. In particular, the three offered types of monitors have the following inconsistencies with our goals: the *String Monitors* are not of interest to us since they do not monitor numerical data; the *Counter Monitors* are suitable only for monitoring integer-typed continuously increasing variables, which is a special case that can not be guaranteed, especially if we did not know the nature of the monitored resources beforehand; the *Gauge Monitors* only send notifications when the value of the monitored resources gets out of a predefined range. However, no information is obtained regarding the behavior of the resource inside or outside that range.

Ganglia [6] is a toolkit aimed at monitoring clusters and clusters of clusters. It relies on the use of monitor daemons (gmond) and meta daemons (gmetad). A gmond, running on every monitored node, probes a fixed set of hardware resources at a fixed sample rate and multicasts the values on the network to gmetad. Ganglia does not offer any probe for J2EE monitoring. Moreover it is not as flexible as LeWYS.

Researchers at Charles University have developed Xampler [7], a benchmarking suite that focuses on extensive evaluation of CORBA environments. It provides probes for Solaris, Windows, and Linux operating systems. LeWYS is complementary to Xampler and targets J2EE environments.

7 Conclusion and future work

In this paper, we have briefly introduced LeWYS, a Java-based monitoring framework targeting J2EE clusters. We have described the implementation of hardware and operating system specific probes, as well as J2EE generic probes (based on JMX). Our preliminary experiments show that it is possible to implement efficient Java-based probes that collect exhaustive resource usage within a J2EE cluster. We are currently experimenting the overhead of the monitoring infrastructure on real e-commerce applications running on J2EE clusters.

Availability LeWYS is freely available under an LGPL license at the following URLs: `http://lewys.objectweb.org`.

References

1. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An Open Component Model and its Support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2004)*, Edinburgh, Scotland, 2004.
2. C. Smith and D. Henry. High-Performance Linux Cluster Monitoring Using Java. In *Proceedings of the 3rd Linux Cluster International Conference*, 2002.
3. Java Management Extensions (JMX) specification version 1.2, Mars 2004. Sun Microsystems, http://java.sun.com/products/JavaManagement/.
4. E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *Proceedings of Freenix*, 2004.
5. M. Knop, J. Schopf, and P. Dinda. Windows Performance Monitoring and Data Reduction Using WatchTower. In *Proceedings of the Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, New York City, 2002.
6. Ganglia Toolkit 2.5.0, September 2002. University of California, Berkeley, USA, http://ganglia.sourceforge.net/.
7. Middleware Benchmarking, 2004. Distributed Systems Research Group, Charles University Prague, http://nenya.ms.mff.cuni.cz/.