Visualisation of Distributed Applications for Performance Debugging

F.-G. Ottogalli[1], C. Labbé[1], V. Olive[1], B. de Oliveira Stein[2],
J. Chassin de Kergommeaux[3], and J.-M. Vincent[3]

[1] France Télécom R&D - DTL/ASR, 38243 Meylan cedex, France
{francoisgael.ottogalli,cyril.labbe,vincent.olive}@rd.francetelecom.fr
[2] Universidade Federal de Santa Maria, Rio Grande do Sul, Brazil
benhur@inf.UFSM.br
[3] Laboratoire Informatique et Distribution, Montbonnot Saint Martin, France
{Jacques.Chassin-de-Kergommeaux,Jean-Marc.Vincent}@imag.fr

Abstract. This paper presents a method to perform visualisations of the behaviour of distributed applications, for performance analysis and debugging. This method is applied to a Java distributed application. Application level traces are recorded without any modification of the monitored applications nor of the JVMs. Trace recording includes records from the JVM, through the JVMPI, and records from the OS, through the data structure associated to each process. Recorded traces are visualised post mortem, using the interactive Pajé visualisation tool, which can be conveniently specialised to visualise the dynamic behaviour of distributed Java applications. Applying this method to the execution of a book server, we were able to observe a situation where both the computation or the communications could be at the origin of a lack of performances. The observation helped finding the origin of the problem coming in this case from the computation.

Keywords: performance analysis and debugging, distributed application, Java, JVMPI, meta-ORB.

1 Introduction

The aim of the work described in this article is to help programmers to analyse the executions of their distributed programs, for performance analysis and performance debugging. The approach described in the following includes two major phases: recording of execution traces of the applications and post mortem trace-based visualisations. The analysis of distributed applications is thus to be done by programmers, with the help of a visualisation tool displaying the execution behaviour of their applications.

In the presented example, a distributed application is executed by several Java™[1] Virtual Machines (JVM) cooperating through inter-objects method calls on a distributed infrastructure.

[1] Java and Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. The authors are independent of Sun Microsystems, Inc.

This work was carried out in the context of the Jonathan project [6] developed at France Télécom. Jonathan is a meta-ORB, that is a framework allowing the construction of object-oriented platforms such as ORBs (*Object Request Brokers*). Jonathan could be specialised to be CORBA or a Java RMI (*Remote Method Invocation*) style. In the following we will be concerned with the CORBA specialisation.

Performance traces are recorded at the applicative level of abstraction, by recording the method calls through the Java Virtual Machine Profiling Interface (JVMPI) [1,14,15]. Additional information needs to be recorded at the operating system level, in order to identify inter-JVMs communications. Execution traces are then passed to the interactive visualisation tool Pajé [4]. Pajé was used because it is an interactive tool – this characteristic being very helpful for performance debugging – which can be tailored conveniently to visualise the execution and communications between JVMs.

The existing Java monitoring and visualisation tools cannot be used conveniently for performance debugging of distributed Java applications. *hprof* [12] and *Optimizeit* [7] provide on-line cumulative information, without support for distributed applications (monitoring of communications). Although this information can be very useful to exhibit performance problems of sequential Java applications, it is of little use to help identifying the origin of performance problems in the distributed settings. The JaVis tool [10] can be used to trace the communications performed by Java Virtual Machines. However the recording is performed by modified JVMs while we decided to stick to standard JVMs.

The main outcome of this work is the ability to observe and visualise the dynamic behaviour of distributed Java applications **including communications**, without any modification of the application programs. Moreover it allows a temporal analysis of the hierarchy of methods invocations.

The organisation of this paper is the following. The recording of execution traces is described in Section 2. Section 3 describes the visualisation tool Pajé and its specialisation for the visualisation of the behaviour of distributed Java applications. Then comes the conclusion which also sketches future work.

2 Recording Traces of the Execution of Java Applications on Distributed Platforms

Performance analysis is performed off-line, from execution traces which are ordered sets of events. An event is defined as a state change in a thread of a JVM. Two types of events will be considered: the infrastructure events, corresponding to state changes associated with the JVM machinery and to the standard JDK classes used, and the applicative events, associated with the execution of the methods of the application classes. In order to take into account the state changes of internal variables, it is possible to build specific recording classes, to be observed as well as applicative classes.

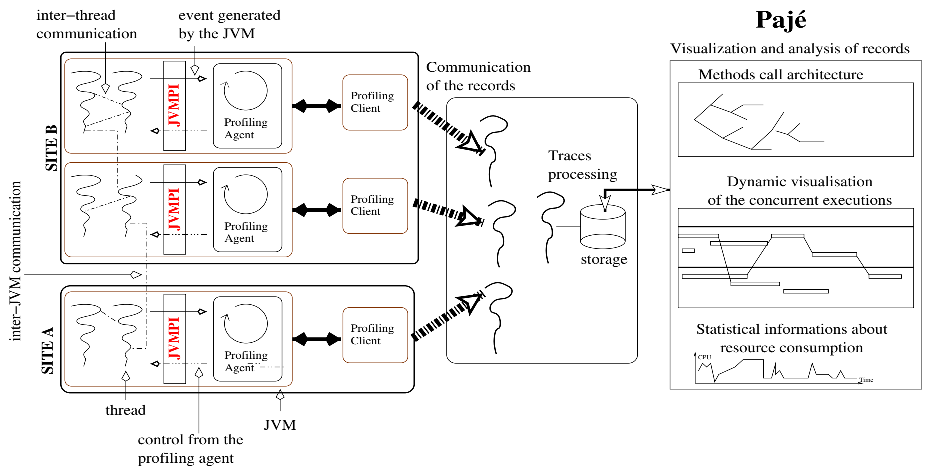
inter–thread communication
event generated by the JVM
Pajé
Visualization and analysis of records
Methods call architecture
Communication of the records
SITE B
JVMPI
Profiling Agent
Profiling Client
Traces processing
Dynamic visualisation of the concurrent executions
storage
inter–JVM communication
SITE A
Statistical informations about resource consumption
CPU
Time
thread
JVM
control from the profiling agent

Fig. 1. ***Global architecture for analysis and performance debugging.*** Profiling agents are used to observe events and interact with the JVM. Profiling clients connect profiling agents to the applications dedicated to collect and process the data traces. Then, the data traces are visualised with Pajé.

To each recorded event a date, identification and location are assigned so that the execution path of the application as well as the communications can be reconstructed post-mortem by the analysis tools.

2.1 Application Level Recording Traces (Figure 1)

Events are recorded using the JVM Profiling Interface (JVMPI) defined by SUN and implemented in the Linux_JDK_1.2.2_RC4. This functional interface allows event occurrences and control operations to be observed [14]. The JVMPI is accessed by implementing an observation agent [1,15], loaded during the initialisation of the JVM. The following events are observed:

- loading or unloading of a class;
- beginning or termination of a method call;
- creation, destruction or transfer of an object in the execution stack;
- creation, suspension, destruction of a thread;
- use of monitors (waiting and acquisition);
- user defined events (program annotations to observe specific events).

Observing these events is used to reconstruct the execution from the JVM point of view. It is used to construct several representations of the execution. One, based on thread execution, displays the execution path of the methods (Figure 4). Another, not shown in this paper, represents the execution in terms of object and method execution.

Constructing an event trace requires to date, identify and localise each event. A unique identifier is associated to each loaded class, and each method defined in these classes. Similarly, Java threads are given a unique identifier. Since the events of a thread are sequentially ordered, it is possible to reconstruct the causal order of an execution by grouping the records by thread.

Using a JVMPI to observe events can potentially produce an important volume of recorded events. It is therefore necessary to provide **filtering** functions to limit the size of the recorded traces. For example (see Section4), filtering an execution trace by a mere exclusion of standard Java classes from the recording, divided by a factor of 32 the size of the execution trace.

Additional information, relating to communications and use of system resources will be recorded at the operating system level of abstraction.

2.2 Information Needed for Communications Observations

To observe the communications between JVMs, we need to identify the links created between them. Calls to the methods performing the communications are recorded by the JVMPI. However, communication parameters, necessary to reconstruct the dynamic behaviour of the applications, are lost in our records.

Accessing these parameters through the JVMPI would not have been simple since calling parameters cannot be accessed without delving into the execution stack of the JVM; this would be the case for example with the communication parameters, when recording a communication event. The approach used instead to obtain the parameters is to observe the messages sent and received at **the operating system level of abstraction**. This choice was driven by two major reasons :

– to have a direct access to the parameters associated to the communications;
– to obtain information about operating system resources consumption [13].

We assume that "sockets" are used to establish connections between JVMs. The identification of the sockets used for inter-objects communications is performed at the operating system layer. In the case of the Linux operating system, the sockets used by a process or a thread are represented as typed "i-nodes". The data structures associated to these i-nodes can be used to identify a given link by the IP address of the remote host and the port number used.

Once all the data have been collected, as described in figure 1, they can be visualised with Pajé.

3 Visualisation Using the Pajé Generic Tool

Pajé is an interactive visualisation tool for displaying the execution of parallel applications where a (potentially) large number of communicating threads of various life-times execute on each node of a distributed memory parallel system [2,3,4]. The main novelty of Pajé is an original combination of three of the most desirable properties of visualisation tools for parallel programs: extensibility, interactivity and scalability.

In contrast with passive visualisation tools [9] where parallel program entities – communications, changes in processor states, etc. – are displayed as soon as produced and cannot be interrogated, it is possible to inspect all the objects displayed in the current screen and to move back in time, displaying past objects again. Scalability is the ability to cope with a large number of visual objects such as threads. Extensibility is an important characteristic of visualisation tools to cope with the evolution of parallel programming interfaces and visualisation techniques. Extensibility gives the possibility to extend the environment with new functionalities: processing of new types of traces, adding new graphical displays, visualising new programming models, etc.

3.1 Adapting Pajé for Visualising Distributed Java Executions

Extensibility is a key property of a visualisation tool. The tool has to cope with the evolutions of parallel programming models – since this domain is still evolving rapidly – and of the visualisation techniques. Several characteristics of Pajé were designed to provide a high degree of extensibility: modular architecture, flexibility of the visualisation modules and genericity. It is mainly the genericity of Pajé which made possible to specialise it for visualising distributed Java applications.

The Pajé visualisation tool can be specialised for a given programming model by inserting an instantiation program in front of a trace file. The visualisation to be constructed from the traces can be programmed by the user, provided that the types of the objects appearing in the visualisation are hierarchically related and that this hierarchy can be described as a tree (see Figure 2). This description is inserted in front of the trace file to be analysed and visualised.

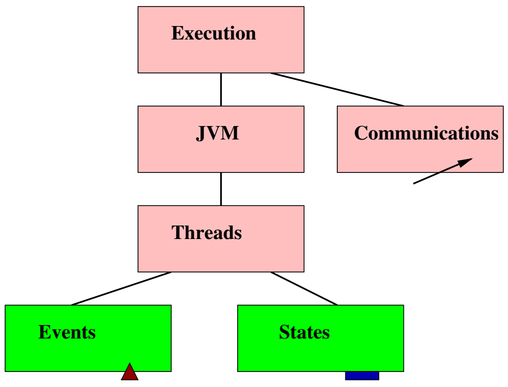

Fig. 2. ***Example of type hierarchy for visualising Java distributed applications.*** Intermediate nodes are containers while the leaves of the type hierarchy are the types of the elementary visual entities. Method calls are represented as state changes of the displayed threads.

3.2 Trace Visualisation with Pajé

The Pajé command language, allowing users to discribe how traces should be visualised, is based on two generic types: containers and entities. While entities can be considered as elementary visual objects, containers are complex objects, each including a (potentially high) number of entities. An entity type is further specified by one of the following generic types: event, state, link or variable. Another generic type used to program Pajé is the value type which can describe the type of one or several fields of complex entity types. Java abstractions were mapped to the Pajé generic command language according to the type hierarchy of Figure 2.

The visualisation is similar to a space-time diagram, where the time varies along the x-axis while JVMs and threads are placed along the y-axis (see Figure 3). The execution of the Java methods are represented as embedded boxes in the frame of each of the threads. Inter-thread communications appear as arrows between the caller and the callee communication methods. It is also possible to visualise, on the same diagram, the evolution of several system variables and the use of monitors, in order to draw correlations between application programs and consumption of system resources (not shown in this paper).

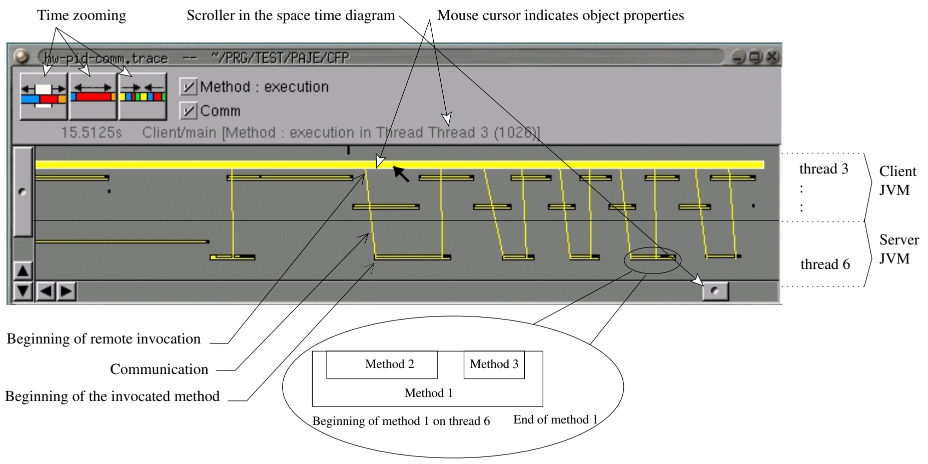
Time zooming
Scroller in the space time diagram
Mouse cursor indicates object properties
hw-pid-comm.trace -- ~/PRG/TEST/PAJE/CFP
Method : execution
Comm
15.5125s Client/main [Method : execution in Thread Thread 3 (1026)]
thread 3
Client JVM
thread 6
Server JVM
Beginning of remote invocation
Communication
Beginning of the invocated method
Method 2
Method 3
Method 1
Beginning of method 1 on thread 6
End of method 1

Fig. 3. ***Visualisation of the Book Server program execution using Pajé.*** The interactivity of Pajé allows programmers to select a period of execution and to "zoom" with respect to the time scale in order to have more details on the period of interest displayed on the screen. Similarly, it is possible to filter the displayed information, to restrict the amount of details to be visualised. For example, it is possible to filter the communications or the calls to some methods.

4 Observation of a Book Server Execution

The main purpose of this observation is to highlight the ordering of events corresponding to the beginning and the end of methods execution as well as communications. This will help identifying performance bottlenecks in terms of processing and network resources consumption.

The target application, an electronic book server, is an important application of the “net economy”. In addition it is representative of a whole set of on-demand data server applications in the domains of multimedia, video and music. Such applications can be characterised by massively concurrent accesses to the provided resources. In this example, the objective was to optimise the global performances of the application. One situation was identified where a bottleneck resulted from the excessive execution time of some methods and not from the communication delays. This drove us to conclude that optimisations should first concern these methods and their scheduling on the threads of the application.

4.1 Description of the Experiment

The displayed trace represents the execution of two clients and of the “book” resource: these entities perform most of the computations and communications. Since our objective is to analyse and improve performances, these entities have to be analysed in priority.

The execution trace was realized during an experiment involving three different locations. The resources and the resource server are hosted by the first site while two clients are located on the two other sites and access the same resource simultaneously. The three computers are interconnected by a 10 MBit Ethernet network; they all run Linux - kernel 2.4.0 of Debian woody - and use the 1.2.2 RC4 JDK from SUN.

This execution environment does not have a hardware global clock. Therefore, a global clock had to be implemented by software [11,8,5].

4.2 Trace Processing and Interpretation

Trace files need to be processed to convert local dates into global dates using Maillet’s algorithm [11]. Matching of Java and system threads is then performed, allowing system and application level traces to be merged.

Figure 4 exemplifies the functionalities provided by Pajé to visualise execution paths and communications. It is a detailed view showing the beginning of a communication.

Figure 5 displays a global view of the experiment described in Section 4.1. The execution of the book server, two clients as well as the communications between one of the clients and the server are visible. This representation makes it clear that the communication phase is very short with respect to the entire duration of the experiment.

Figure 6 is used to identify the origin and destination of communications (JVM, thread, method) as well as the interleaving of the computation and communication phases of threads.

Thus it is possible to observe a pattern (gray area in Figure 6) composed of four communications occurring five times during the visualised time interval.

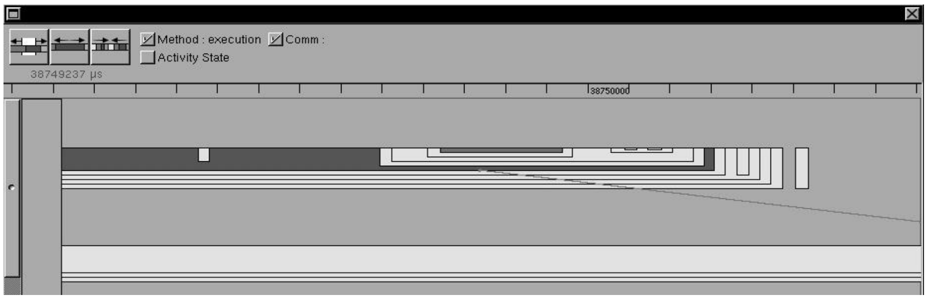

Fig. 4. ***Visualisation of the execution path of a thread.*** This visualisation represents the execution of two threads inside the same *JVM*. The embedding of clear and dark boxes represents the embedding of method calls. Dark boxes represent selected boxes. The line originating from the upper thread represents the beginning of a communication.

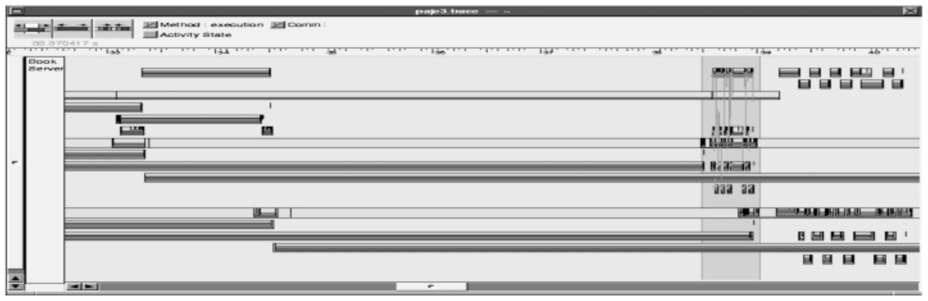

Fig. 5. ***Visualisation of the execution of three JVMs.*** This figure represents the execution of three *JVMs* on three different sites. An area, representing a period which includes several communications between two *JVMs*, was selected (in gray) to be magnified, using the zooming facilities of Pajé (Figure 6).

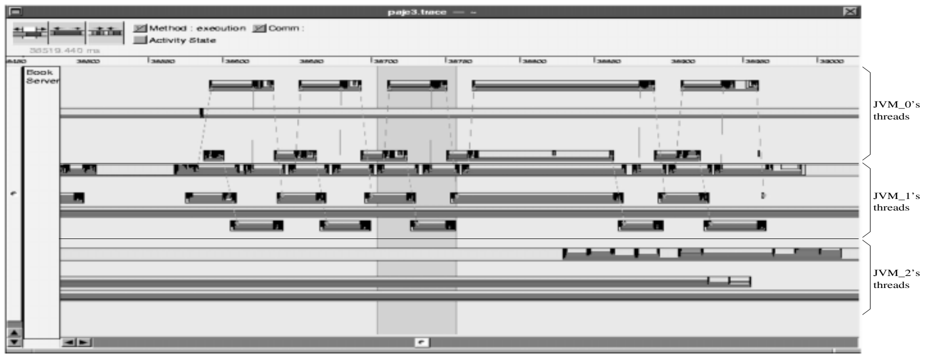

Fig. 6. ***Observing a series of communication sequences between two JVMs.*** *JVM* 0 represents the execution of the "book" resource while *JVM* 1 and 2 represent the executions of the first and second clients. Only communications between the "book" resource and the first client are displayed.

This pattern can be divided in two phases including each two communications: the first one from the client to the server and the second one from the server to the client. The execution time of this pattern is fairly constant, except for the forth occurrence. A trace analysis indicates that the whole execution is slowed down by the excessive duration of a thread (third from the top). The remaining of the sequence is indeed dependent on the sending of a message from this thread.

On the contrary, communication delays are low and constant with respect to computation times. In such a situation, improving the performances of this application is better done by optimising some specific methods that are identified to be time consuming than by trying to reduce the communication delays.

5 Conclusion

This paper presents a method to perform visualisations of the behaviour of distributed applications in the scope of performance analysis and performance debugging. Application level traces are recorded without any modification of the monitored applications nor of the JVMs. Trace recording includes recording of the method calls at the application level by the JVMPI as well recording communication information at the socket level of the operating system. Recorded traces are visualised post mortem, using the interactive Pajé visualisation tool, which can be conveniently specialised to visualise the dynamic behaviour of distributed Java applications. This method has been applied to a Java distributed application. It was thus possible to discriminate between two possible origins of a performance problem, ruling out the hypothesis of inefficient communications.

This work is still in progress and is amenable to several new developments, some of them concerning the tracing activity while the other extensions concern the visualisation tool.

First of all, the tracing overhead should be analysed in order to be able to assess the quality of the traced information and therefore of the visualisations of the executions of the traced programs. Further work will aim at evaluating the system resources required by on-line analysis of the traces: this analysis could help evaluating the system resources to be provided to analyse on-line distributed applications while keeping the analysis overhead low. Another perspective is the use of the observation and visualisation environment for system “black-box” analysis.

For the Pajé visualisation tool, the perspectives include several extensions in order to integrate the operating system resource consumption observations – such as the use of processors, memory or the bandwidth of the network – with the application visualisations and therefore help programmers to relate both. In addition, Pajé could be extended in order to take into account several problems related with the visualisation of embedded systems, such as the display of the synchronisation mechanisms.

References

1. Java virtual machine profiler interface (JVMPI). Technical report, Sun Miscrosystems, 1999.
2. J. Chassin de Kergommeaux and B. de Oliveira Stein. Pajé: an extensible and interactive and scalable environment for visualizing parallel executions. R.R. 3919, INRIA, april 2000. http://www.inria.fr/RRRT/publications-eng.html.
3. J. Chassin de Kergommeaux and B. de Oliveira Stein. Pajé: an extensible environment for visualizing multi-threaded programs executions. *Euro-Par 2000 Parallel Processing*, *LNCS* 1900, pages 133–140. Springer, 2000.
4. J. Chassin de Kergommeaux, B. de Oliveira Stein, and P. Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10):1253–1274, aug 2000.
5. J. Chassin de Kergommeaux, E. Maillet, and J.-M. Vincent. Monitoring parallel programs for performance tuning in distributed environments. *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments*, chapter 6. Nova Science, 2001. To appear.
6. B. Dumant, F. Dang Trang, F. Horn, and J.-B. Stefani. Jonathan : an open distributed processing environment in java. In *Middleware'98: IFPI Int. Conf. on Distributed Systems Platforms and Open Distributed Processing*, Sept. 1998.
7. G. Freedman. Common java performance issues and solutions. Technical report, Intuitive System Inc., 1998.
8. Y. Haddad. *Performance dans les systèmes répartis : des outils pour les mesures.* PhD thesis, Université de Paris-Sud, centre d'Orsay, Septembre 1988.
9. M. T. Heath. Visualizing the performance of parallel programs. *IEEE Software*, 8(4):29–39, 1991.
10. I. Kazi and al. Javiz : A client/server java profiling tool. *IBM - Systems Journal*, 39(1):82, 2000.
11. E. Maillet and C. Tron. On efficiently implementing global time for performance evaluation on multiprocessor systems. *Journal of Parallel and Distributed Computing*, 28(1):84–93, 1995.
12. N. Meyers. A performance analysis tool. Tech. report, Sun Miscrosystems, 2000.
13. F.-G. Ottogalli and J.-M. Vincent. Mise en cohérence et analyse de traces logicielles. *Calculateurs Parallèles, Réseaux et Systèmes répartis*, 11(2), 1999.
14. Sun Microsystems. *Java Virtual Machine Profiler Interface (JVMPI)*, Feb 1999.
15. D. Viswanathan and S. Liang. Java virtual machine profiling interface. *IBM - Systems Journal*, 39(1):82, 2000.