

# A CPU Resource Consumption Prediction Mechanism for EJB deployment on a federation of servers

Stéphane Frénot

INRIA Arès

INSA-CITI

Bat. Léonard de Vinci

69621 Villeurbanne Cedex, France

+33 4 72 43 64 22

stephane.frenot@insa-lyon.fr

Tudor Balan

INRIA Arès

INSA-CITI

Bat. Léonard de Vinci

69621 Villeurbanne Cedex, France

tudor\_balan@yahoo.com

## ABSTRACT

In this paper, we describe a model for characterizing EJB performances in terms of CPU load. Our model aims at deploying EJB on a federation of heterogeneous servers. Deployed EJB are provided by external clients that want them to be hosted on the federation with a guaranteed execution time. That guarantee has a cost which is defined by the model. In our model, candidate EJBs are firstly benchmarked in order to characterize their consumption in CPU. Then, having an idea of their consumption there is a negotiation with the client and a contract that guarantees an execution response time for a certain amount of concurrent clients. We show in this paper that it is possible to evaluate the performance of an EJB on a dedicated server and deduce the performance on a real production server. We also show that the response time for an EJB is a factor of its own characteristics and the total number of clients accessing the application server on which it is deployed.

## Categories and Subject Descriptors

### General Terms

Management, Performance, Verification.

### Keywords

Performance prediction, Deployment, RMI, CPU consumption, EJB

## 1. INTRODUCTION

This research paper takes place in the context of a large project called DARTS (Deployment and Administration of Resources, Treatment and Services) [1] which is currently developed at INSA LYON laboratories. DARTS aims at building a framework allowing deployment and administration of services and software components in the GRID [2] computing context.

The main purpose of this paper is to introduce an intelligent management of EJB components [3] on a federation of application servers. The servers federation is a collection of candidate application servers that can accept to host new EJB components. Taking into consideration the CPU load the EJB component consumes and the available federation hardware resources, our system finds the most appropriate server the EJB will be deployed on. The direct implementation of our system would be a commercial application where EJB components are hosted for charge. The charge reflects in a direct proportional manner the resources consumption of the EJB to host.

Since EJB components are externally accessed through the java RMI layer we will show that the round robin behavior of RMI enables a simple CPU load prediction model that only relies on the total number of concurrent clients accessing the server. That standard result of the round robin queuing system enables us to build a simple deployment management mechanism on a federation of servers.

Section 2 presents the overall constrains we are dealing with for our federation of servers. Section 3 presents the CPU load estimation model we have defined. Section 4 presents our implementation algorithm of that model. Finally, section 5 concludes and proposes some possible evolutions of the model.

## 2. System constrains for CPU load evaluation

In this section we give the more precise context in which we apply our model. This context is a federation of application servers executing EJB components related to GRID computing. In that context there are constrains for the federation of servers and for the execution model that relies on Java RMI framework.

### 2.1 Servers federation

A servers federation is a collection of heterogeneous hardware of different capacities. Each physical server hosts an EJB container that can execute EJB components on behalf of some concurrent accessing clients. Each server presents a specific behavior as more and more clients access the system simultaneously.

EJB are proposed by providers that wish to host them on the federation. Providers must indicate which EJB method is the most CPU intensive. The federation benchmarks the EJB and proposes hosting cost depending on the number of concurrent clients and response time the client wishes. Of course the more the providers pays, the more clients will be tolerated simultaneously. Since different EJB will be hosted on the same server with different execution contracts we need to guarantee that each contract stays coherent with the growing potential clients. In that context we are working on an upper limit that guarantees response time whatever the number of clients (beneath the contract number) are accessing the different EJBs. In other word if each EJB is used at its maximal load the server still responds in a bounded time.

Since we are in the GRID context, we consider that EJB are not used in the "classical" e-business approach, but in a more CPU consuming approach. For a first evaluation, we only take into account session beans (stateless and statefull).

Because each GRID computation should execute at its maximum power, we do not want to interfere with the execution on the federation. Thus all predictions and evaluations must be held out of the core of the federation.

## 2.2 RMI-EJB server

RMI [4], [5] is a framework which can be viewed as an interface between outer clients and EJB execution runtime. The role of RMI server in this context should be viewed as a Remote Procedure Call mechanism. It guarantees that the client method calls are sent to the EJB server and that return values from the EJB server reach the client. In order to establish our model we need to explain some details of the RMI framework.

On EJB server side, there exists a finite pool of application threads. On client side, the RMI server opens a RMI-Connection thread for each client. Then it matches each RMI-Connection thread with an application thread. If the number of RMI-Connection threads exceeds the number of available application threads, RMI-Connection threads will be permuted in a round robin manner in order to get access to application threads. All application threads are of the same priority and there is no preferential treatment for any of them. However, application threads are Java threads, not OS threads. They are mapped into so called "lightweight processes" of the JVM process. These "lightweight processes" obey the OS specific thread policy but usually they are run in a round robin manner [6].

All concurrent clients are finally mapped into EJB server threads which in turn are mapped into OS threads of the same priority which will be executed in a round robin manner most of the times. Thus, each thread will benefit from the same CPU access.

Consequently, we can model an EJB server and its N concurrent clients as a queuing system as illustrated in Figure 1.

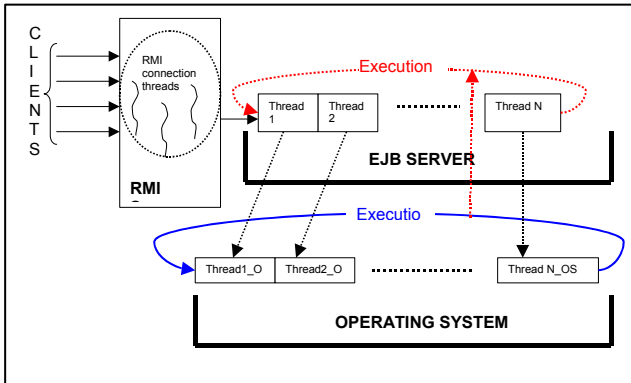


Figure 1: RMI Thread mapping

These threads represent a method execution of one of the deployed EJB components of EJB server. We make no assumption on the number of clients for a certain EJB component and on the number of threads corresponding to a certain EJB component.

It is this threads mechanism which stays at the basis of our prediction approach which is detailed in the following section.

## 3. CPU execution time model for EJB

In this section we will show that the CPU load evaluation model is the same as any standard queuing system since it relies on the RMI framework. We show that the response time of one EJB on a application server depends only on its own features and on the total number of connected clients. If we know a specific execution time when one client executes the EJB, we can infer the execution

time when many clients are connected simultaneously, even if they do not execute the same EJB.

### 3.1 CPU execution time estimation model

The goal of CPU analysis is to identify the influence of this essential resource on EJB server and EJB components performance and to provide a mathematical model which will be integrated in the evaluation process.

We consider that we have N concurrent clients accessing the server at full time, this means that as soon as one thread finishes its execution, a new one will take its place. This assumption is made in order to insure that we always analyse the worst performance time with N permanent clients. This means that the execution of a thread is done from its first until its last quantum in the presence of other N-1 threads.

One important element to remember is that N represents all clients connected on all active EJB and not only the clients on the evaluated EJB.

Let us introduce the following notations:

$k \rightarrow$  index variable spanning the threads:  $1 \leq k \leq N$

$s_j \rightarrow$  a CPU quantum length for server j ( $s_{isolated}$  represents that value for a specific isolated server from the federation)

$i_{kj} \rightarrow$  the number of cycles the  $k^{th}$  thread needs to complete on server j. ( $i_{kisolated}$  represents that value for a specific isolated server from the federation)

As soon as the queue is in a stable state, the time needed for the  $k^{th}$  thread to complete in the presence of other N-1 threads is  $t_k = i_{kj} * s_j * N$ . Even if we do not know  $i_{kj} * s_j$ , this product represents in fact the execution time of the  $k^{th}$  thread in isolation conditions (executed alone, without any other concurrent thread). This product  $t_{kisolated} = i_{kisolated} * s_{isolated}$  is evaluated on a out of core server and is used as the base value for the load prediction. These two formulas are rather trivial and are standard results of queuing system. They mean that :

1 - If we know the most consuming method of an EJB, we can find a standard measure that represent intrinsically that EJB. That measure can be evaluated on an isolated server. If we also know the scale factor between  $s_j$  and  $s_{isolated}$  we can deduce the execution time of that EJB on any server of the federation.

2 - The prediction time for the execution depends on the total amount of connected clients on the server. That result is used to evaluate the worst execution time. For example if  $ejb\_1$  has negotiated a maximum number of 100 clients,  $ejb\_2$  has negotiated a maximum of 150 and  $EJB3$  as negotiated 150 clients then its worst execution time is  $t_{kj} = i_{kj} * s_j * 400$  ( $i_{kj} * s_k$  is a product known on server k).

In order to negotiate the cost of the hosting of a new EJB, we need to define the transaction concept.

### 3.2 Transaction

Defining the **transaction** as a completed client request, and considering that the  $k^{th}$  client permanently issues the same request to the server, then the number of transactions (Tx) that may be completed for k clients in interval T is

$$Tx_k = T / (i_{kj} * s_j * N) = T / (t_{k\_isolated} * N)$$

Denoting  $T/t_{k\_isolated}$  with  $C_k$ , becomes:  $Tx_k = \frac{C_k}{N}$

Figure 2 illustrates that the number of transactions for a client served by an EJB component evolves with the total number of clients. We define  $V_{k\_limit}$  as the minimum value of  $Tx$  for which a client is satisfied by the response time.  $N_{k\_limit}$  represents the total number of clients for which  $V_{k\_limit}$  may be still achieved and for which the EJB provider will pay for.

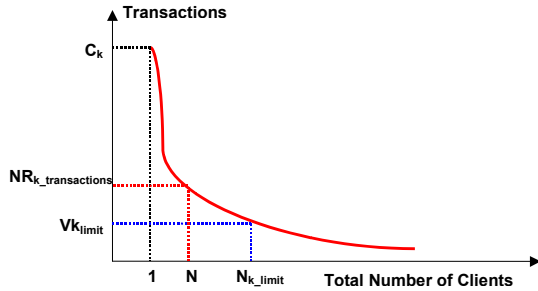


Figure 2: Transaction evaluation

In order to prove that the transaction time are directly influenced by the total number of client connected we have made some validating tests.

### 3.3 Empirical CPU consumption

We have defined a benchmarking architecture that is able to validate the previous model. Though the description of that architecture is out of the scope of that paper, it's worth noting that the most complex problem of the benchmarking architecture is to build a real concurrent stressing system since RMI is rather difficult to stress concurrently. We have used some standard tools [7], [8], [9] in order to collect instrumentation data. The benchmarking architecture we implemented resembles that of [10].

We have made 3 experiments in order to prove that the execution time of an EJB is  $t_1=N*i_1s$ . Where  $N$  represents the total number of clients accessing all active EJB on the system.

In the first experiment we consider that 20 concurrent clients ( $N=20$ ) stress a unique EJB component (ejb\_1). We are interested in determining the real execution time of each of the concurrent clients. Theoretically, having a unitary execution time (execution time for one isolated client) of  $t_0$ , an  $N*t_0$  value should be obtained for any of the  $n$  concurrently running clients.

In the first stage, we consider a single client which issues a single method invocation on ejb\_1. The business code is executed  $k=30$  times. Computing the average execution time we obtain the value of 352ms. Figure 3 represents the unitary execution times and the average unitary execution time curves.

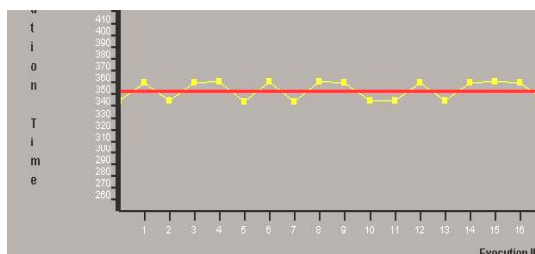


Figure 3: 1 client executing 30 times a CPU consuming method

In the second experiment we consider all  $n=20$  clients concurrently assaulting ejb\_1 and executing the same method. For each client  $i$  we obtain a set of values  $V_i$  representing the  $k=30$  execution times. Figure 4 represents three curves. The bottom curve represents the nominal execution time of 352ms for 1 individual client. The two upper ones represents respectively the real and the average measures obtained with 20 concurrent clients ( $t=20*t_0$ ).

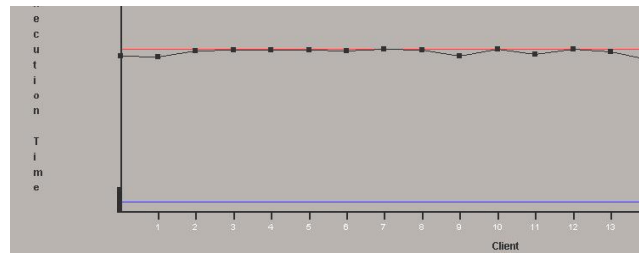


Figure 4: 20 clients executing 30 times a CPU method

In the last experiment we want to show that the total execution time is influenced by the total amount of clients connected on the system independently from which EJB the request is made on. We have made another EJB that is accessed by our concurrent clients. We consider that the  $n=20$  clients will stress ejb\_1 and ejb\_2 with an equal distribution, namely: the first 10 clients will stress ejb\_1 and the other 10 clients will stress ejb\_2. Obviously for each of the  $n=20$  clients we obtain a set of values  $V_i$  representing the  $k=30$  execution times performed during a specific method invocation. The curves on figure 5 show that the response time of the EJB is only influenced by its execution time and the total number of concurrent clients.

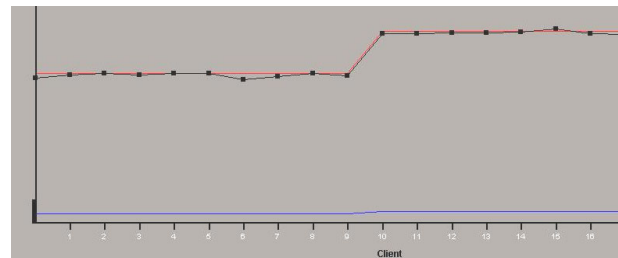


Figure 5: 20 clients accessing simultaneously ejb1 and ejb2

Having validated our execution time prediction model, we have defined an architecture that implements that model in order to deploy EJB on a federation of servers. That deployment algorithm must respect transactions constraints specified by the EJB provider. We present that algorithm in the next section.

## 4. A Server federation deployment algorithm

The goal of our research efforts is to architect a performance prediction based system, able to host EJB components in an optimized way. Our deployment algorithm mechanism which governs the federation relies on the following entities:

- A Server federation: each server member of the federation must host an appropriate number of EJB types. Each hosted EJB type has a guaranteed transaction value. Independently of the number of client accessing it (provided every EJB stays in its contract).

- A Benchmarking unit: each new type of EJB willing to integrate the federation must be benchmarked in order to be hosted on one of the server of the federation.
- Deployment Management: that mechanism aims at selecting the “best” server that will host the EJB for a certain range of clients accessing it.

A new type of EJB willing to be hosted will be evaluated under the following procedure.

#### 1. Benchmarking process

That first step aims at providing a benchmarking value for the CPU consumption. In the EJB architecture every method that should be invoked is known through either the home or the remote interface. Thus the most consuming method is one of these. In our approach, the EJB provider delivers the jar containing the EJB he wants to be hosted. Therefore, before being benchmarked, any EJB component provided must declare its most CPU consuming method. Knowing that method, the EJB is run alone on a benchmark server. That execution provides the  $t_{isolated}$  measure.

#### 2. Negotiation process

The  $t_{isolated}$  measure is used to calculate transaction curves for different servers capacities of the federation through the use of the scale factor. Each server knows its scale factor with the *isolated* server. That factor is calculated at the server installation through a server benchmarking process. Knowing various transaction curves, the EJB provider chooses for its EJB a maximum number of clients that should access it and a willing minimum transaction value (Max response time).

#### 3. Selection process

The transaction value previously selected defines which servers of the federation are able to host the new EJB (it is the minimal value above which the client won't be satisfied). Then we calculate for every selected member of the federation the perturbation induced if the new EJB would be hosted. That calculation integrates for every currently hosted EJB the fact that there may be N simultaneous new clients. Each server of the federation maintains a table of active EJB and its associated contract. The process controls that every active EJB still conforms to its negotiated contract with the total amount of potential concurrent clients (including the new one). If one or many selected server tolerates the new EJB, we can choose on which server the EJB will be deployed. The most powerfull (the EJB provider pays more) or the less powerfull (the EJB providers pays less).

#### 4. Deployment process

That last process is the standard one of any EJB deployment.

### 5. Conclusion and future work

In this article we have presented a global architecture for EJB deployment on a federation of application servers. The deployment mechanism mainly relies on a benchmarking process that provides some initial values that characterize the EJB. We have shows and validated that on RMI framework the execution time for one EJB only depends on the total number of clients accessing any EJB and on specific values characterizing that EJB. Nevertheless our architecture needs to be improved in the following ways :

Other parameters influence: actually our implemented architecture launches the same deployment algorithm for the memory measure. When a new EJB is to be deployed, he declares its most consuming CPU method and its most memory consuming method. However the total amount of memory used by a session bean only depends on the total number of simultaneously connected clients on that specific EJB. So the selection mechanism is much more simple since it is sufficient to control that the total memory amount requested by all concurrent clients on the EJB is available. The last parameter that should be benchmarked is the network related stuff such as the bandwidth that the EJB consumes. We do not have yet tackled that value.

Sincerity Control mechanism: Since the EJB provider indicates which of the EJB methods consumes the most, there can be errors if its not the case. Each federation member needs to trigger a sincerity control mechanism that verifies that every EJB component stays in its negotiated values. Since the application server knows the number of current accessing client, it is able to calculate the response time of every active EJB. If one response time is above its prediction the sincerity mechanism can exclude the EJB.

Finally it is worth noting that the system is not exclusive to the EJB architecture, since the model is only based on the RMI behavior. It is possible to apply this system to every system that can expose its external public methods and which is accessed in a round robin queuing method. For instance web services are conformed to that model.

### 6. ACKNOWLEDGMENTS

This work is supported by the french ministry of research under the ACI GRID funding program.

### 7. REFERENCES

- [1] Darts Project, INSA Lyon, <http://darts.insa-lyon.fr>
- [2] The Global Grid Forum, <http://www.gridforum.org>
- [3] The EJB specification, <http://java.sun.com/ejb>
- [4] The Remote Method Invocation, <http://java.sun.com/rmi/>
- [5] A. Wollrath, R. Riggs and J. Waldo, “A distributed object model for the java System”, 2<sup>nd</sup> conference on object technologies and systems (COOTS), 1996
- [6] Edward Harned, “Using an asynchronous process manager to contain your RMI server applications”, <http://www-106.ibm.com/developerworks/java/library/j-rmiiframe>
- [7] The java profile J-Profiler, ej-technologies, <http://www.ej-technologies.com/products/jprofiler>
- [8] Optimizelt, Borland, <http://www.borland.com/optimizelt>
- [9] PerformanceAnalyzer, a graphical view of hprof, <http://developer.java.sun.com/developer/technicalArticles/Programming/perfanal/>
- [10] Adrian Mos, John Murphy, “A framework for performance monitoring and modeling of enterprise java beans applications”, 16 Annual ACM SIGPLAN Conference on Object-Oriented Programming, System, Languages and Applications (OOPSLA, Tampa Bay, Florida, USA)