Transparent and Dynamic Code Offloading for Java Applications

Nicolas Geoffray, Gaël Thomas, and Bertil Folliot

Laboratoire d'Informatique de Paris 6
8 rue du Capitaine Scott, 75015 Paris France
firstname.lastname@lip6.fr

Abstract. Code offloading is a promising effort for embedded systems and load-balancing. Embedded systems will be able to offload computation to nearby computers and large-scale applications will be able to load-balance computation during high load. This paper presents a runtime infrastructure that transparently distributes computation between interconnected workstations. Application source code is not modified: instead, dynamic aspect weaving within an extended virtual machine allows to monitor and distribute entities dynamically. Runtime policies for distribution can be dynamically adapted depending on the environment. A first evaluation of the system shows that our technique increases the transaction rate of a Web server during high load by 73%.

1 Introduction

Some systems evolve in unpredictable environments: choices that were appropriate when launching an application might become inappropriate at a given time during execution. Pervasive computing and web servers belong to this kind of systems. Pervasive computing aims at bringing computer technologies into all kinds of surrounding entities: cars, highways, desktops, refrigerators, and so on. Embedded devices will evolve in different environments, from having plenty of external resources with good network bandwidth like in airports or hotels, to no external resources at all. If an application would like to take advantage of all these resources, it would have to take into account distribution issues and dynamicity of the environment. In the domain of Web servers, a hot-spot-of-the week effect [24] changes a regular situation with few connections to an exceptional situation where the server has to respond to a large amount of requests. One solution consists in dedicating a given amount of machines to balance the load, but this solution is far from optimal for a Web Server, because resources are underused during normal execution.

To face these unpredictable situations, we propose to inject on the fly an adaptable offloading system directly into the runtime when the local resources are overloaded and when new resources become available. Offloading ensures better performances and better responsiveness of the application. The offloading system is removed when unneeded. Our experience is based on an extended Java Virtual Machine (JVM), but it can be applied to other kinds of virtual machines. The main advantages of our solution are:

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1790–1806, 2006.

- The system loads the offloading code dynamically: it does not have to reboot the application in order to take into account distribution.
- The offloading code is independent from the application. An application which separates platform-dependent and platform-independent computations can benefit from a network of resources, even if it was not designed for.
- The application can dynamically adapt the distribution policies such as what to offload, or under which constraints (memory, computation). The offloading system is based on a flexible runtime that enables dynamic adaptation.
- The external resources are not dedicated to one application: they can be shared between many applications.
- External resources do not have to install a dedicated virtual machine. The system sends the offloaded code to standard environments.

Our solution relies on JnJVM [29] an adaptable Java Virtual Machine (JVM). JnJVM is enriched with a Java dynamic aspect weaver [12] that modifies applications during execution. We use JnJVM because (i) it is built on the same adaptable runtime as the offloading system and (ii) its aspect weaver is efficient and fully functional [19].

Section 2 describes the underlying architecture of the system that allows dynamic modification of the Java application, and dynamic adaptation of the distribution policies. Section 3 describes what is possible to offload in a Java program. Section 4 presents our offloading system. Section 5 evaluates the infrastructure. Section 6 shows related work and Section 7 concludes the paper.

2 The Execution Environment

Our offloading system relies on three different dynamic mechanisms: code loading, adapting the running system's code and adapting the application's code. The micro virtual machine (MVM) is an execution environment which offers the two first mechanisms. The JnJVM, which is constructed on top of the MVM performs the last. We describe both of these systems in this section.

2.1 The Micro Virtual Machine

The MVM is a dynamic compiler and a dynamic code weaver. It compiles components, whose architecture allows adaptation during execution. A component contains fields and methods. Two mechanisms are provided by the component architecture: replacing a component by an other, or modifying the implementation of a component's method. The dynamic code weaver is responsible for inserting the new code. The compiler of the MVM is described in [21]. The MVM is an enhacement of the system described in [22]. The MVM is also the execution environment of many projects, such as an adaptable Web cache [18], or a flexible micro object request broker [17].

In this project we use the MVM in order to dynamically load the offloading code, and adapt its policies during execution.

2.2 JnJVM

JnJVM is a full JVM that follows the J2SE specification 1.2 to 1.4 [9].[1] It is built on top of the MVM: it extends the MVM with Java-specific components such as a bytecode compiler (JIT) or a Java class loader. JnJVM offers the same performance as open source JVMs such as Kaffe 1.1.6 and is three times slower than industrial JVMs such as IBM. We use the JavaGrande Forum Benchmark Section 2 to evaluate JnJVM on PowerPC and Pentium. We expect that an engineering effort would yield better results, because our compiler and garbage collector are not optimized. The memory footprint of JnJVM is around 20 MB.

Dynamic aspect weaving in JnJVM: An aspect is an association between existing code of the application and new code. Aspect weaving is the act of modifying the existing code to invoke the new code. In this paper, we consider dynamic aspect weaving: the aspects are woven during execution.

We have designed a Java aspect weaver for JnJVM. This aspect weaver is separated from JnJVM and is reused for this work. The Java aspect weaver was presented in [19]. In its implementation, weaving an aspect on a method affects the next calls to the method. The aspect weaver does not perform on stack replacement.

3 Offloading in a Java Application: Discussion

Java manipulates different kinds of entities: classes, objects, methods and threads. Offloading an object means sending the object to a surrogate, offloading a class means offloading all the objects of this class, and offloading a method means executing it (and all the methods it calls) on a surrogate. These entities can all be candidates in our offloading system, but each one has its advantages and drawbacks described in the remainder of this section. Class and object migration is mostly for memory saving, whereas method migration is for computation. Finally, thread offloading means thread migration. Thread migration requires to have at each side (surrogate and server) a dedicated virtual machine, because standard JVM forbids capturing or reconstructing an execution stack. Having a dedicated virtual machine for thread migration is tedious to implement, and requires dedicated virtual machines.

3.1 Object, Class and Method Offloading

In this section, we discuss advantages and drawbacks of the different strategies. Table 1 compares the three approaches on (i) their purpose (processing or memory), (ii) the difficulty to analyze them statically (on the source code), and (iii) the difficulty to analyze them dynamically (during execution).

Object offloading: Choosing an object to offload can be done by analyzing the bytecode or the source code of the application or during the execution of the application. In the first case, algorithms exist [26] to determine the relations between

[1] JnJVM is based on GNU Classpath that follows the specification 1.2 to 1.4.

Table 1. Comparison between class, object and method offloading

Strategy	Purpose	Difficulty of static analysis	Difficulty of dynamic analysis
Method	Processing	*	*
Class	Memory	*	**
Object	Memory	*	***

objects of the program and therefore to know which objects are intensively used. The code of the application is then automatically changed to track all the chosen objects. In contrast, the execution analysis monitors the objects of the application and collects statistics on objects use. With these informations (collected statically or/and dynamically), a graph can be computed. The arcs between objects represent their interactions, and the weight of an object represents the other informations, such as the object's size. We can then apply graph partitioning in order to choose the objects that are the most interesting to offload. Static analysis has a major drawback compared to execution analysis: its informations are not as precise, and the objects interactions discovered can be misleading, for example when hot objects are used intensively at startup but rarely used after.

Class offloading: Offloading a class requires to trace all objects of the class, and to send all of them to surrogates. Choosing a class to offload can also be done by static analysis or during execution time. We can apply a graph partitioning heuristic to choose the classes to offload, depending on their interactions. This strategy suffers from adding to the virtual machine instructions to locate for one class all of its objects. It also suffers from having to send all the objects to surrogates, even if some of these objects are not much used in the program. The advantages of class offloading compared to object offloading is that it can be easier in the virtual machine to monitor classes than objects: the virtual machine does not have to track creation and destruction of the monitored objects.

Method offloading: Offloading a method is an easier way to offload an entity than the two previous ones. Static analysis computes call graphs. Execution analysis collects statistics on methods usage like number of times executed, execution time or data exchanged (size of the arguments and return). Compared to class offloading, the disadvantages of method offloading is that there are more methods than classes in a program. The analyzers have to do more work in method offloading, but the offloading might be more effective for performance: offloading a method concerns especially execution, not memory usage. The memory constrained devices generally use an interpreter, thus they do not lose memory for compiled methods. Offloading objects and classes target memory usage and execution. Finally, method offloading does not suffer from heavy communications between surrogates and the server: because all methods called by the method are sent to the surrogate, execution does not have to go back and forth between the surrogate and the server. With objects and classes, the method calls inside a class's method are sent back to the server. When offloading a method, the objects it uses are attached and the server waits for the result.

3.2 Static Analaysis of Method Migration

In our first prototype we implement method offloading. We are working on integrating static analysis for objects [26], or classes [10] in order to ease the execution environment on decisions of which objects or classes to offload.

Not all Java methods can be offloaded. In Java, in order for an object to be exchanged between Java virtual machines, its class has to implement the `java.io.Serializable` interface. An object that is not serializable can not be exchanged. Our solution targets any kind of applications, but the applications might have been conceived without distribution issues, and therefore, all of their objects are not serializable. There are also some classes in the core classes of Java that are not serializable, like the `java.net.Socket` class which represents a native socket of the operating system.

Our system sends objects as method parameters to the surrogates. Therefore these objects have to be serializable. We chose to modify the bytecode of an application in order to add the `Serializable` interface to classes that do not manipulate platform dependent objects, i.e. have real non-serializable fields. We expect to have an analyzer that will find these classes automatically.

Because we chose method offloading, we have to find which methods can be sent to surrogates. Candidate methods must not manipulate platform-dependent objects, and must not call methods that manipulate platform-dependent objects. For the moment, this analysis is done by hand. The informations given for a candidate method are: (i) the classes it needs for execution, i.e. the methods classes that it calls, and the classes of the arguments, (ii) the static objects accessed by the method and submethods, and (iii) which parameters and static variables might be modified (see Section 4.4). Because hand analysis is unconceivable for large applications, we are working on a tool that will make this analysis automatically.

4 Offloading System

The runtime environment must be enhanced with a system that offloads execution or memory to surrogates. In this section we present how the runtime is modified to be able to analyze execution of the offloading entities, and distribute them. Figure 1 gives an overview of the infrastructure.

The offloading system extends the MVM. Its main activities are:

- Dynamically monitor the application execution. In order to offload computation, the system has to know what is interesting to distribute (Monitoring aspect).
- Distribute Java entities. The platform must perform transparent distribution (Distribution aspect).
- Manage the resources. External resources (surrogates) can be removed or added dynamically. The platform has to take into account all modifications of the environment (Surrogate administration).

– Adapt the distribution by an agent. The agent analyzes the results of the application's monitoring, and decides what to offload and on which resources (Offloading agent).

Our platform does not need to modify the source code of an application, nor does it need to have surrogates executing a dedicated virtual machine. The execution is distributed by sending jobs that any JVM can execute, in order to have a larger amount of external resources possibilities. The following steps are done to realize offloading:

1. Static analysis of the application's bytecode gives a set of Java entities that should and/or can be offloaded,
2. Surrogates register and remove themselves to the platform,
3. An agent is launched inside the platform: it decides when offloading should operate,
4. The agent weaves a monitoring aspect on the entities given at stage 1, when the system is not overloaded,
5. Based on this monitoring and on the current situation of the application (presence of many surrogates, heavy load, etc), the agent weaves the distribution aspect that will send jobs to surrogates,
6. When offloading is not anymore necessary, it unweaves the distribution aspect. Execution continues with no overhead.

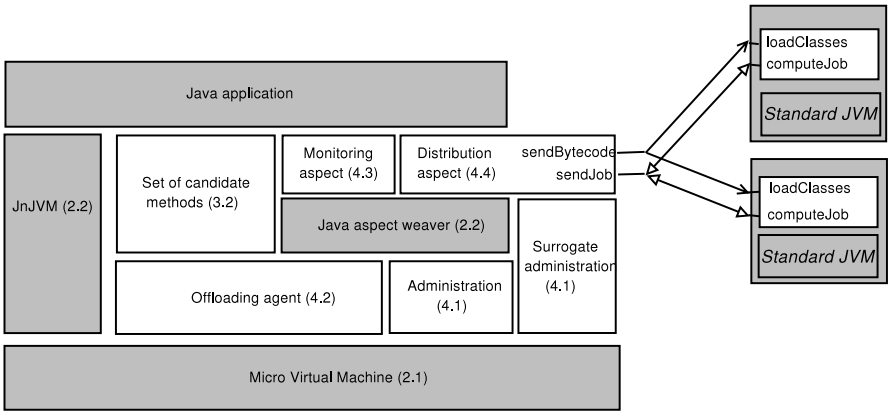

Fig. 1. Overview of the offloading architecture

4.1 Human Administration

There are two entries in the offloading system, each dedicated for two kinds of systems. The first entry targets limited resource devices. The second is Web servers. These entries serve two purposes: (i) administration for the offloading decisions, and (ii) surrogates register themselves dynamically. We are investigating on using a service discovery standard such as Jini [2]. Surrogates can be all kind of computers: grids, servers or simple personal computers.

Resource-limited devices administration: For limited devices, the system is enhanced with a communication socket. An administrator sends through the communication socket instructions for the offloading system: for instance it sends the set of methods that can be offloaded, or it forces a method to be offloaded.

Web server administration: We use servlets to administrate Web servers extended with our load balancing system. A servlet is a Java object that receives requests and sends responses. It is part of the J2EE specification. When used in a Web server, it dynamically creates a web page based on the contents of the request. A servlet is executed inside a servlet container. Our evaluation uses Tomcat (see Section 5), a servlet container. Therefore, instead of a GUI administration approach, we chose a servlet approach. The servlet is responsible for (i) communicating with the offloading system, by sending the instructions given by an administrator through a web page, (ii) printing on the web page the state (local or remote) and (iii) storing statistics usage of all methods. Instructions are sent to the offloading system through the servlet by using HTTP requests.

4.2 Autonomous Agent

The servlet and the communication socket provide entries for an administrator in order to operate on the offloading system. The administrator adapts the system depending on the current environment: heavy load, new peers, memory usage, etc. It decides when to offload a method. This task is not acceptable in the long term due to the permanent human presence it requires. For this reason, an autonomous agent inside the system should manage these tasks: the offloading agent. It is composed of probes and states, and executed as a MVM thread. We have implemented an agent for load balancing the application on top of a Linux system (the application is still in user mode): it probes the operating system's load average (eg, reads the `/proc/loadavg` file), and switches between two states, `charged` and `uncharged`. The `charged` state offloads methods which execution would benefit from being remote. The benefit is calculated by comparing the number of bytes to send versus the network's quality, and comparing the method's execution time versus the time to execute the Java distribution aspect. The `uncharged` state changes execution of all offloaded methods to local, i.e. the agent unweaves the distribution aspect.

Because it is implemented in the MVM, the agent can be dynamically modified by adding new probes or new states. Therefore, the selection of methods to offload can be adapted dynamically. For example, if memory is a constrained resource, probing the `/proc/meminfo` file could be added, as well as a new state `memory`. The agent would be in the `memory` state when there is not much memory left. In such case, it will offload methods with the highest compiled size and free memory taken by the method. Or it may switch to objects and class offloading. We will take into account lemory issues when the system will be enhanced with objects or classes offloading.

4.3 The Monitoring Aspect

We use dynamic Java aspect weaving to inject monitoring in the set of methods that can be offloaded. The monitoring aspect collects three statistics. First, the average execution time in order to know which methods cost the most at execution. Second, the number of times executed in a given elapsed time. This information gives the hot methods of the application. Third, the average size of the objects used (arguments, return, static variables). This last statistic gives an estimation of the distribution cost.

The system uses these runtime statistics in order to choose the most interesting methods to offload. An administrator or the agent can modify the monitoring aspect and add new statistics collecting for methods.

Our prototype collects statistics on Java entities execution, and records the characteristics of each surrogate. Network quality consideration is an improvement we are currently working on.

4.4 The Distribution Aspect

Offloading a method's execution to surrogates results in a distributed execution, where the surrogates (the clients) offer their processing capabilities to the server, and the server sends jobs to surrogates.

Our solution aims at sending jobs to standard environments: the owners of the surrogates do not have to install a dedicated execution environment. This facilitates usage of all kind of resources: it is easier to have a standard environment executing on different architecture platforms than with a specialized environment. The drawback is that it must use a standard distribution framework, not optimized for our kind of system. We chose RMI as a first framework for simplicity reasons. However, the system is not constrained with a unique framework, and we are currently investigating on using different ones.

Each surrogate executes an RMI object that exposes two methods: the first method, `receiveFiles`, receives the Java bytecode of the method and submethods to execute, and the second, `remoteMethodInvocation`, triggers the execution of the method.

Algorithm: The server executing the application holds references of all the peers available. A reference is a stub on which it can call the two RMI methods. When the system choses to offload a method, it weaves the distribution aspect on the method to:

1. Get the static objects used by the methods,
2. Put all arguments in an array,
3. Get a peer (i.e. get its stub) given by the agent,
4. Send the bytecode (call to `receiveFiles`) if the peer does not have it,
5. Invoke the method by calling `remoteMethodInvocation`,
6. Update the modified objects (see Section 4.4),
7. Return the result.

The call is synchronous: once a thread calls the `RemoteMethodInvocation` method, it must wait for the response. When the system decides that offloading the method is not anymore necessary, the distribution aspect is unwoven.

Distribution issues: Because we transparently distribute an application, we must face issues inherent to distribution. We now discuss how our system deals with some of these issues: remote call semantics, memory coherence and fault tolerance.

Remote call semantics: Some distribution frameworks like RMI have a call by copy semantic. This means copies of objects are sent and remote modifications done on these copies are lost when the remote method returns. This is not appropriate for our system, because we must have a transparent distribution, where a remote call becomes semantically like a local call. We used a call by copy-restore semantic to achieve such transparency.

The copy-restore algorithm is expensive: for an object Obj the remote call has to send back the new, modified 0bj and all reachable objects from the old, original Obj. The application that called the remote method receives these objects, and has to trace all references of the old Obj in order to compare them with the new Obj. It is better to analyze the application in order to know which objects must have a call by copy-restore semantic. The informations of a methods that can migrate contain therefore the parameters and the static variables modified in the method.

A non-standard implementation of call by copy-restore already exists for Java, NRMI [31]. We did not use this implementation because it changes the standard Java core classes, and JnJVM uses the GNU Classpath classes. Furthermore, the application code has to be modified in order to specify that an object has to be restored locally when sent to a remote call. Because we do not directly call a migrated method, but instead call a method of the RMI object with the information of what method to call, our solution can bypass modifications to the RMI implementation. If needed (there can be distribution frameworks that already have the call by copy-restore semantic) the call by copy-restore algorithm is inserted in the Java distribution aspect of the server. The implementation of the RMI object returns to the server an array of all objects that could have been modified by the method's execution.

Memory coherence: When a surrogate executes a method, the objects passed as arguments are duplicated on the surrogate. Therefore there can be two copies of one object. If the Java program is mono-threaded, execution is stopped locally, waiting for the result of the remote method invocation: no modifications on the objects can be done locally, and only the remote method can modify them. If the program is multi-threaded, the objects sent to the surrogate are manipulated locally and on the surrogate. This issue concerns the programmer, not the offloading system: if objects are not protected in the program (i.e. synchronized calls), whether the method is executed locally or remotely does not change the application semantic.

Synchronized methods can be migrated, but synchronizations inside the method need particular care: because the method is executed on a different virtual machine than the one that executes the application, a synchronization on a surrogate must be transmitted to the server. We are investigating on modifying

the method by adding remote calls that will inform the server of synchronizations. The modified method will then be sent to surrogates instead of the original method.

Fault tolerance: Method offloading allows our system to be partially fault-tolerant : if a peer executing a method crashes (detected by timeout), the method can be executed elsewhere, because all modifications done to objects on the dead peer are not recorded in the server. However, this solution is not possible if the method interacts with external entities, like databases. For this kind of problem, we are currently investigating on using rollbacks on databases when a surrogate crashes. When the system will be enhanced with object or class offloading, we will consider replications of distributed objects, in order to guaranty fault-tolerance.

5 Evaluation

Our evaluation compares execution of JnJVM without offloading and execution of JnJVM with offloading within a Web application. Comparisons with other JVMs without offloading are for the moment irrelevant: JnJVM and GNU Classpath are in their early stage, and they need engineering efforts to compete with commercial JVMs (see Section 2.2).

The application is a free implementation of TPC-W.[2] It is an interactive bookstore implemented with servlets, that interacts with a MySQL database. The servlet container we used is Tomcat 5.0.28. The server on which the application and MySQL are executed is a 1 GHz PowerBook G4 with 1 GB memory.

The application is composed of 29 classes, 15 of which are servlets, 49 methods and 5409 lines of code. A manual analysis of the code can detect 21 methods which execution can be migrated (most part of the other 28 methods manipulate specific servlet objects, which manipulate platform-dependent objects and are therefore not serializable). The majority of the methods that can be migrated follow this algorithm: (i) contact a local socket pool manager in order to aquire a connection to the database, (ii) make a query to the database and (iii) compute on the query's results. The Java class of the socket pool manager is serializable.

Tomcat is launched on the JnJVM without the offloading code. After startup, we load and install the offloading system in JnJVM. We give to the offloading system the set of methods chosen by our static analysis. The offloading agent starts, and begins to analyze execution of these methods. The surrogates add themselves to the offloading system: in the first experiment, a bi-processor G5 2 GHz and a 3 GHz pentium register to the offloading system. In the second experiment, a cluster of 20 bi-processors Xeon 2 GHz dynamically registers to the offloading system. The JVM executed on these machines are Sun JVM 1.4 for pentium and IBM JVM 1.4 for PowerPC. They all execute the RMI object presented in Section 4.4.

[2] http://jmob.objectweb.org/tpcw.html

Our experiment uses the `siege` utility.[3] `Siege` performs benchmarks for Web servers by creating a user defined number of simulated users that make requests to the server. The statistics reported by `siege` are: elapsed timed, data transferred, average response time, transaction rate, average throughput,average number of simultaneous connections, and availability (the number of requests that were satisfied by the servers).

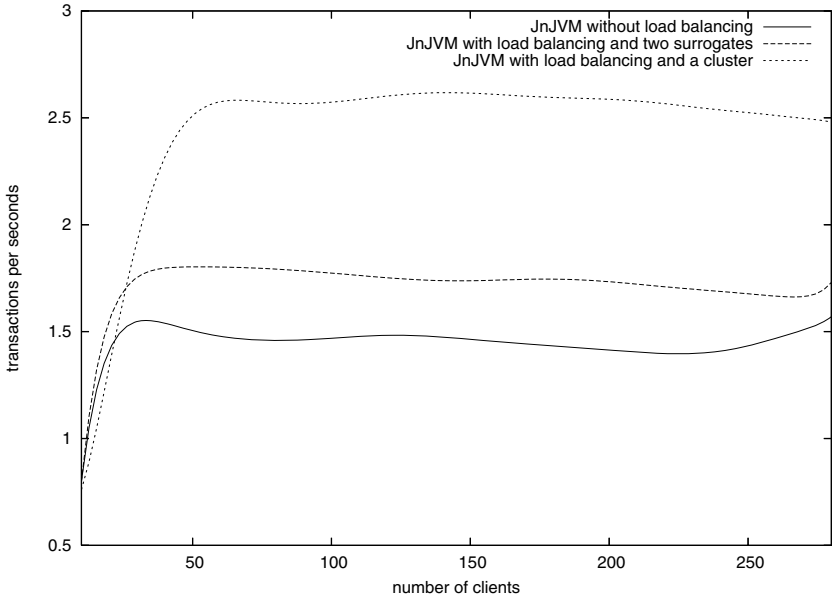

Fig. 2. Transactions per seconds

In the experiments, `siege` was launched for 10 to 400 simulated clients. The clients send requests to the TCP-W application. The first experiment uses JnJVM without load balancing, the second experiment uses JnJVM with load balancing and two surrogates, and the third experiment uses JnJVM with load balancing and the cluster. Figure 5 compares transactions per seconds. The results show that the number of transactions per seconds is increased by 17% with two surrogates, and 73% with the cluster. These numbers do not depend on the number of connections. Figure 5 compares average response time of the server. When 300 clients send requests, the average response time is decreased by 10% with two surrogates and 42% with a cluster. Figure 5 compares availability of the server between the three experiments. The availability of the server starts to decrease when 290 clients are connected, in contrast to 300 for two surrogates, and 360 with a cluster. These results show that our offloading system performs well, however note that increasing the number of surrogates does not lead to higher performance. The small improvement between two surrogates and a cluster of surrogates is mostly due to the fact that the server still has to dispatch the

[3] http://www.joedog.org/siege/

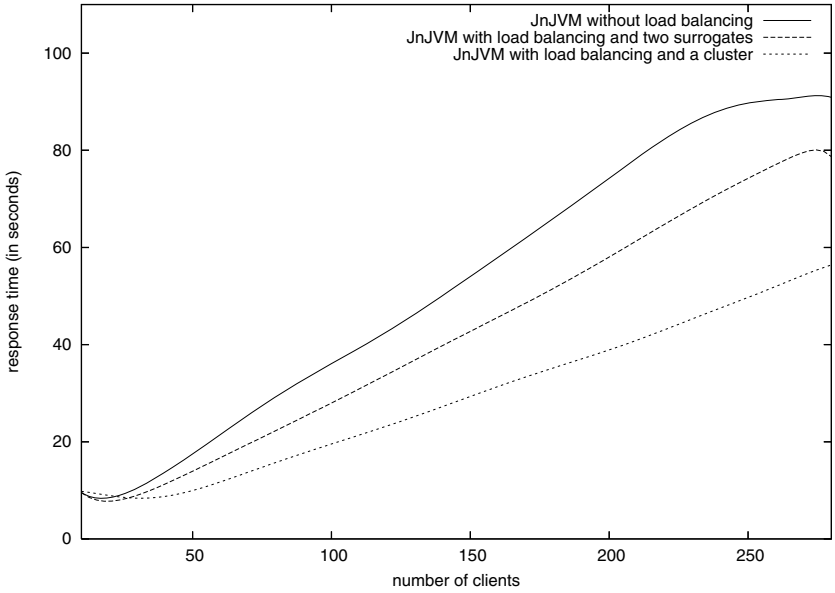

Fig. 3. Average response time

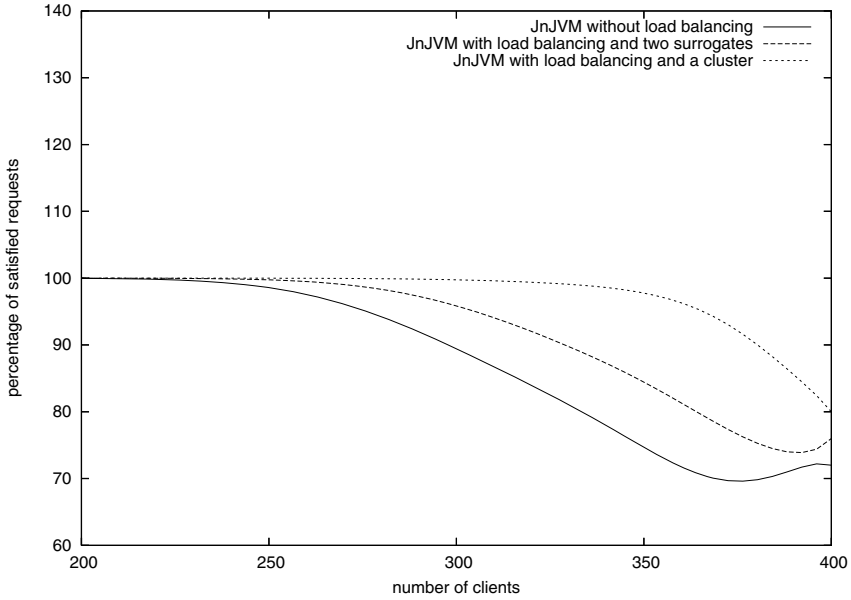

Fig. 4. Availability

requests. The performance of the server are the bottleneck. A solution would be to have a limited pool of servers, shared between many applications, dedicated only to receive requests and call offloaded methods.

6 Related Work

Our work aims at distributing on different resources a centralized Java application. There are mainly three related approaches to achieve this distribution: execution of the program on a dedicated cluster, static automatic distribution of the program, or transparent offloading on possible resources of Java entities during execution of the program.

6.1 Distributed Execution on Clusters

Distributed execution of Java programs on clusters can be achieved with thread migration, a distributed shared memory or a distributed virtual machine.

There are four approaches to thread migration: modifying the source of the application [8], modifying the bytecode [32,23], developing an extended virtual machine like MOBA [25], or using non standard functionalities of a Sun JVM [4]. Bytecode or source modifications adds computations inside the application. This enables the migration of a thread at any state. This additional computation consists in either try/catch clauses or tests and thus reduces performances. An extended virtual machine for thread migration requires to execute it at each node. Our solution aims at executing jobs on standard JVMs. Furthermore, thread migration is tedious and it is easier to implement method migration.

Distributed shared memory can be used to allocate and manipulate objects, like in Java/DSM [34], MultiJav [5], Javanaise [11] or Jessica [14]. The system executes over a cluster of machines, where each node participates in the shared memory, and therefore has an extended virtual machine. Removing or adding a node dynamically is not considered in these systems. This is not satisfactory for our solution, because nodes are here dedicated to the application, and can not remove themselves while the application is executing.

Finally, a distributed virtual machine like cJVM [3] or JESSICA2 [35] allows to execute an application over a cluster where each node participates in the extended JVM execution. These JVMs offer preemptive thread migration for dynamic load balancing. As for distributed shared memory, removing or adding nodes is not considered: nodes have to be present anytime during the application execution.

6.2 Automatic Distribution

JavaParty [20] extends the Java language with a new keyword `remote`, that developers add to classes which instances should be remote. A preprocessor is then applied to write the distributed code. Execution of the application is done in a standard virtual machine with a registry that enables locating objects. Our solution does not modify the Java language. Moreover, the distribution is decided statically in JavaParty and dynamically in our system. As JavaParty, we can offer distribution control with an instruction file that specifies which entities should be executed remotely. Remote Objects in Java [16], like in JavaParty, introduces a new keyword `remotenew` to create objects on another virtual machine. The

new keyword is translated in a new opcode, which can only be understood by an extended virtual machine, which limits the use of the bytecode.

Doorastha [6] allows a programmer to annotate a standard Java program with new keywords in order to specify which objects of the application should be distributed, and how each distributed object should be transmitted to methods. The Doorastha framework compiles the program and chooses where to place objects. Placement can be decided statically or dynamically. Our solution aims at executing existing applications and therefore does not modify the Java language.

Pangaea [27] uses an object graph algorithm [26] which operates on Java source code to find the objects that interact the most in the program. Based on the graph, a distributed code is generated to execute the application between different computers. The distribution code can either be in RMI, CORBA or Doorastha. Objects placement and number of participating processors are decided statically. J-Orchestra [30] and Addistant [28] modify the bytecode of a Java application to generate a distributed code, based on informations given by a programmer or user of the application. These informations specify which classes should have their instances distributed. In J-Orchestra, these classes are transformed to RMI classes. Addistant uses a master/proxy system, where proxies of an object communicate with the master object. Pangaea, J-Orchestra and Addistant modify an application before it is executed. Thus, dynamic adaptation is not considered, and objects that were not selected during bytecode migration can not be changed to distributed objects during execution. Participating resources are given statically, and can not be modified dynamically. Furthermore, these systems do not analyze the application during execution, and can therefore not adapt their distribution strategies at runtime.

6.3 Transparent Offloading

Lattanzi *et al.* [13] improve the performances of a FPGA with a Java interpreter by offloading CPU intensive methods to a co-processor. The Java virtual machine analyzes each method to find which method should be offloaded. The chosen method is compiled and migrated to the co-processor. This system uses shared memory and therefore does not deal with network distribution issues.

Azul Systems [1] performs dynamic load balancing for J2EE applications. The applications will be able to offload their execution to a dedicated grid, composed of dedicated hardware and a dedicated virtual machine. The technology is relatively new and specific to Web applications. Our solution aims at adding distribution to any kind of Java application.

Diaconescu *et al.* [7] propose a compiler and runtime infrastructure for transparently distributing a Java application. It performs a graph analysis based on [26] that performs on bytecode (Pangaea performs on source), and with this graph generates a distributed program. It uses the Joeq virtual machine [33] to compile the unmodified application into a distributed version. Communications between the server and surrogates are in MPI. During execution, the system is monitored to estimate the quality of repartition, but the user can not adapt the

distribution. Our prototype executes on a flexible environment, the MVM, that eases the adaptation of policies.

AIDE [15] extends the ChaiVM virtual machine, a JVM dedicated to embedded devices and analyzes interactions between classes during execution of an application. A graph partitioning algorithm uses the analysis to find the best partitioning in terms of communications between classes. Partitions can be offloaded to surrogates, dynamically and transparently for the application. Surrogates execute also an extended virtual machine. We aim at offloading on standard Java virtual machines, in order to have a large amount of external resources.

7 Conclusion

We have presented a runtime infrastructure that is able to offload computation to surrogates. It performs monitoring and analyzes the results collected in order to distribute execution. Not all applications can benefit from our offloading system: high relations between objects, or high dependencies with non-serializable objects can prevent offloading. However, we believe that a program which separates business code and platform dependent code has a good part of its entities enable to be distributed.

Future works to improve our prototype are:

- To develop the method analyzer. For the moment, the analysis is done by hand. This is not conceivable for large applications: this tool will ease usage of the offloading system.
- To offload objects and classes. Objects and classes offloading serve for more memory saving.
- To integrate different kinds of static analysis. Object graph analysis [26], and class interactions graph [10] will give additional informations on what is best to offload.
- To target different standard distribution frameworks. The current framework is RMI, but frameworks dedicated for to grids could give better performances.
- To execute different kinds of applications with different needs. We have evaluated the advantages of our system for dynamic load balancing for a Web server. Our next target is offloading under memory constraints like in embedded devices.

We have applied dynamic aspect weaving to inject transparent monitoring and distribution in applications. The result is an extensible framework for load-balancing and/or offloading. Evaluation of the system shows that a Web server is improved in responsivness and availability.

Acknowledgements

We would like to thank the anonymous referees for their suggestions to improve the content and presentation of this paper.

References

1. Azul systems. http://www.azulsystems.com.
2. Jini. http://www.jini.org.
3. Y. Aridor, M. Factor, and A. Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *Proceedings of the International Conference on Parallel Processing*, pages 4–11, Fukushima, Japan, September 1999.
4. S. Bouchenak and D. Hagimont. Zero Overhead Java Thread Migration. Technical Report 0261, INRIA, May 2002.
5. X. Chen and V. H. Allan. MultiJav: A Distributed Shared Memory System Based on Multiple Java Virtual Machines. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 91–98, Las Vegas, USA, July 1998.
6. M. Dahm. The Doorastha System. Technical Report B 00-01, Freie Universität Berlin, May 2000.
7. R. E. Diaconescu, L. Wang, Z. Mouri, and M. Chu. A Compiler and Runtime Infrastructure for Automatic Program Distribution. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 52–67, Washington, USA, 2005.
8. S. Funfrocken. Transparent Migration of Java-Based Mobile Agents. In *Proceedings of the Mobile Agents Conference*, pages 26–37, 1998.
9. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification (2nd Edition)*. 2000.
10. X. Gu. Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments. In *In Proceedings of the IEEE Pervasive Computing and Communication*, 2003.
11. D. Hagimont and D. Louvegnies. Javanaise: Distributed Shared Objects for Internet Cooperative Applications. In *Proceedings of the Middleware Conference*, The Lake District, England, 1998.
12. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, Jyväskylä, Finland, June 1997.
13. E. Lattanzi, A. Gayasen, M. Kandemir, V. Narayanan, L. Benini, and A. Bogliolo. Improving Java Performance by Dynamic Method Migration on FPGAs. In *Proceedings of the IEEE Reconfigurable Architecture Workshop*, April 2004.
14. M. Ma, C. Wang, and F. Lau. JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Journal of Parallel and Distributed Computing*, 60(10):1194–1222, 2000.
15. A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, T. Giuli, and X. Gu. Towards a Distributed Platform for Resource-Constrained Devices. Technical report, Hewlett-Packard, 2002.
16. N. Nagaratnam and A. Srinivasan. Remote Objects in Java. In *Proceedings of the IASTED International Conference on Networks*, 1996.
17. F. Ogel, B. Folliot, and G. Thomas. A Step Toward Ubiquitous Computing: An Efficient Flexible Micro-ORB. In *Proceedings of the ACM SIGOPS European Workshop*, pages 176–181, Leuven, Belgium, sept. 2004.
18. F. Ogel, S. Patarin, I. Piumarta, and B. Folliot. C/SPAN: A Self-Adapting Web Proxy Cache. In *Proceedings of the Autonomic Computing Workshop of the International Workshop on Active Middleware Services*, page 178, June 2003.

19. F. Ogel, G. Thomas, and B. Folliot. Support Efficient Dynamic Aspects Through Reflection and Dynamic Compilation. In *Proceedings of the Symposium on Applied Computing*, Santa Fe, USA, March 2005.
20. M. Philippsen and M. Zenger. JavaParty: Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.
21. I. Piumarta. The Virtual Processor: Fast, Architecture-Neutral Dynamic Code Generation. In *Proceedings of the Virtual Machine Research and Technology Symposium*, pages 97–110, San Jose, USA, 2004.
22. I. Piumarta, F. Ogel, and B. Folliot. YNVM: Dynamic Compilation in Support of Software Evolution. In *Proceedings of the OOPSLA Engineering Complex Object Oriented System for Evolution Workshop*, Tampa Bay, USA, October 2001.
23. T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *Proceedings of the Agent Systems and Applications/Mobile Agents Symposium*, pages 16–28, 2000.
24. M. Seltzer. The World Wide Web: Issues and Challenges. Presented at IBM Almaden, 7 1996.
25. K. Shudo and Y. Muraoka. Asynchronous Migration of Execution Context in Java Virtual Machines. *Future Generation Computer Systems*, 18(2):225–233, Oct. 2001.
26. A. Spiegel. Object Graph Analysis. Technical Report B-99-11, Freie Universität Berlin, July 1999.
27. A. Spiegel. PANGAEA: An Automatic Distribution Front-End for Java. In *Proceedings of the Heterogeneous Computing Workshop*, pages 93–99, 1999.
28. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. *Lecture Notes in Computer Science*, 2072, 2001.
29. G. Thomas, F. Ogel, A. Galland, B. Folliot, and I. Piumarta. Building a Flexible Java Runtime upon a Flexible Compiler. In Jean-Jacques Vandewalle David Simplot-Ryl and Gilles Grimaud, editors, *Special Issue on 'System & Networking for Smart Objects' of IASTED International Journal on Computers and Applications*, volume 27, pages 28–47. ACTA Press, 2005.
30. E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002.
31. E. Tilevich and Y. Smaragdakis. NRMI: Natural and Efficient Middleware. In *Proceedings of the International Conference on Distributed Computing Systems*, Providence, USA, May 2003.
32. E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Proceedings of the Agent Systems and Application/Mobile Agents Symposium*, pages 29–43, Zürich, Switzerland, September 2000.
33. J. Whaley. Joeq: A Virtual Machine and Compiler Infrastructure. In *Proceedings of the Interpreters, Virtual Machines and Emulators Workshop*, pages 58–67, San Diego, USA, June 2003.
34. W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, 1997.
35. W. Zhu, C. Wang, and F. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *Proceedings of the International Conference on Cluster Computing*, Chicago, USA, September 2002.