

Execution of Monolithic Java Programs on Large Non-Dedicated Collections of Commodity Workstations

Michael Factor
IBM Research Lab in Haifa
Haifa University Campus
Haifa 31905, Israel
factor@il.ibm.com

Assaf Schuster
Computer Science Department
Israel Institute of Technology
Technion City
Haifa 32000, Israel
assaf@cs.technion.ac.il

Konstantin Shagin
Computer Science Department
Israel Institute of Technology
Technion City
Haifa 32000, Israel
konst@cs.technion.ac.il

ABSTRACT

Today's typical computational environment consists of a large numbers of commodity workstations interconnected by high bandwidth networks. However, only a small portion of workstation's capacity is utilized since a large share of workstations may be idle at any given moment. Moreover, the communication subsystem dedicated to the idle workstations may be also unused. This results in immense aggregate waste of computational and network resources. We propose the general design of a runtime that executes a standard Java application on available idle workstations. This runtime transparently distributes the application's threads and objects among the currently idle nodes.

Our target environment contains a very large but fluctuating number of non-dedicated workstations. The biggest obstacle in such a *non-dedicated* environment is a participating workstation becoming unable to continue its part of the execution. This can be caused by a failure, *e.g.*, a power outage, *etc.*, or by user intervention, *e.g.*, shutdown or restart. These events result in the immediate loss of the data and threads located on the node, which violates the integrity of the distributed execution. Another, at first seemingly less severe problem is minimizing the response time impact when a node is reclaimed by its owner. This is important, because if the response time impact is not negligible the user will forbid participation of his/her workstation in the distributed execution.

The described system possesses the following features. First, it is *fault tolerant*, preserving the correctness of the distributed execution even if a large portion of nodes fails. By use of the system's fault tolerance we make it *non-intrusive* with respect to the owners of the participating nodes. We reduce impact on the node's response time to essentially zero by modeling the node's reclamation as a failure, killing all runtime threads and freeing runtime memory as soon as presence of node's owner is detected.

Second, our runtime is completely *transparent* to the programmer and can execute a standard, pre-existing, multithreaded Java program written for a single machine. This eliminates the need for a Java programmer to study a new programming paradigm or to use unfamiliar libraries in order to deal with the true face of a non-dedicated distributed environment.

Third, the proposed runtime is *orthogonal* to the implementation of the JVM. It does not require any changes to a node's JVM. This is achieved by rewriting the application's bytecodes, tying it to the bytecodes of the runtime. The instrumented program becomes a distributed application, ready to be executed on multiple nodes using only the node's standard JVM. The ability to use standard JVMs allows portability across *heterogeneous* collections of commodity workstations. Another significant benefit from independence on JVM implementation is the ability of each node to locally optimize the performance of its JVM, *e.g.*, via a JIT.

Finally, the described runtime is highly *scalable*, using an efficient and scalable multithreaded software distributed shared memory (DSM) and a fault tolerance scheme that does not require global cooperation. Scalability allows our runtime to incorporate a large number of workstations increasing its computational force.

We view this work as a first step in providing a convenient computing infrastructure using wide-scale non-dedicated environments. The underlying question is whether (mostly idle) Internet and enterprise interconnects are sufficiently wide to efficiently support high-level programming paradigms, such as distributed shared memory. If this is the case, then, as opposed to current community wisdom, it is not mandatory to restrict the communication pattern of a program that is utilizing idle resources. Java, as a popular multithreaded programming language, is best suited for this experiment.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*run-time environments*. D.1.3 [Programming Techniques]: Concurrent Programming – *distributed programming*.

General Terms

Design, Reliability, Languages.

Keywords

Fault Tolerance, Distributed Shared Memory, Non-dedicated Environment, High-level programming paradigm, Utilization of Idle Resources.