Screen Saver Science: Realizing Distributed Parallel Computing with Jini and JavaSpaces *

William L. George and Jacob Scott
National Institute of Standards and Technology
Information Technology Laboratory
Mathematical and Computational Sciences Division

September 4, 2002

1 Introduction

Screen Saver Science™ (SSS) is a distributed computing environment in which useful computations are performed on a set of participating computers whenever their screen savers are activated [4]. In contrast to other distributed computing projects, such as SETI@Home (http://setiathome.ssl.berkeley.edu), the compute servers of this system, that is, the part that runs within the screen saver, will not consist of a dedicated scientific application. The SSS server will have no particular calculation embedded in it at all, but instead will be capable of performing any computation, subject to local resource constraints such as the amount of memory available. This is made possible through the use of applications compiled to portable Java bytecode along with the Jini and JavaSpaces technologies that have been enabled by the Java environment. Another fundamental difference between SSS and other distributed computing projects is that SSS servers can communicate with each other during the computation in order to coordinate the computation, rather that simply exchanging data and results with a central job manager, thus presenting a distributed parallel computing model to the SSS application programmer. Also, a calculation running in an SSS server can submit one or more new calculations back into the SSS system.

This project will explore the issues involved in building a production quality SSS computing environment for routine use by computational scientists. Parallel algorithms suitable for this environment will also be developed and tested. We expect to show that this is possible with a minimum of extra software needed above the basic Java/Jini/JavaSpaces software that is currently available. We intend to develop a small set of Java packages to be used to develop applications

*To be presented at the 2002 Parallel Architectures and Compilation Techniques Conference.

for submission to our SSS system, mostly for handling file I/O and interprocess communication [7].

2 Background: Java/Jini/JavaSpaces

Java has a number of qualities that promote it as a suitable language in which to write distributed applications. The portability of Java bytecode greatly simplifies code development for heterogeneous environments. Java's garbage collection system removes the need for explicit memory allocation and deallocation. And with its many high level packages, including RMI, Jini, and JavaSpaces, Java gives the distributed application programmer a major head start over other languages.

In the SSS environment we expect most tasks to consist primarily of scientific numerical codes, an application area that Java was not originally designed for. However, the performance of Java on numerical computations has been continuously improving as the result of more aggressive optimizations in the Java Virtual Machines (JVM) as well as language improvements targeted at numerical computations. Such improvements in Java have been the focus of the JavaNumerics group (http://math.nist.gov/javanumerics/) of the JavaGrande Forum (http://www.javagrande.org/).

Jini is a network technology extension to the Java programming language. Jini provides for the automatic discovery of services on the network using a set of simple lookup and discovery protocols. These protocols are designed to enable the building of scalable and robust networked systems. Jini leverages the dynamic remote class loading ability of Java to deliver services, or proxies for these services, to clients.

One of the fundamentals of Jini is the Lookup service, which is at the core of how clients and services interact. The Lookup services act as dynamic yellow pages. Services register a proxy object with the Lookup services, as well a set of attributes that gives more detail about the services. Proxies can be self contained or communicate with a remote service using any wire protocol it requires. These service objects support well known interfaces, so a client finds a particular service that it needs by sending a *template* to the Lookup service, specifying in the template which classes, interfaces, or attributes the required service must match.

A JavaSpace is a Jini service that provides a persistent shared memory for other processes and Jini services on the network. This service was modeled after the tuple-space programming paradigm developed by Gelernter [2, 3]. A JavaSpace will register with Lookup services, just like any other Jini service, so that processes may find it. Items in a JavaSpace are Java objects (in serialized form) that can be found via an associative lookup similar to that used by the Lookup service. The basic API provided by JavaSpaces consists of three methods: `read`, `write`, and `take` (a destructive read). These methods, in combination with the Jini facilities for leases, transactions, and remote events, provide a simple yet powerful distributed programming environment.

The combination of portable Java byte code with the coordination software of Jini and JavaSpaces provide all of the infrastructure needed for constructing a basic *compute server environment*. A compute server is a process that is capable of accepting a unit of work called a *task*, and returning results upon completion. A compute server runs an infinite loop that: gets a task, performs the computation in the task, and returns the result. Other processes are responsible for providing the tasks and for collecting the results. For example, a compute server environment using Jini and JavaSpaces would consist of four main components:

1) `Tasks`: `Task`s are Java objects that contain the basic unit of computation. They have an `execute()` method that returns a `Result` object. The code in the `execute()` method can be different for each `Task`. `Task`s are stored in one or more JavaSpaces until taken by a compute server. A `Task` can also act as compute client in the system by submitting `Task`s from within its `execute()` method.

2) `Results`: `Result` objects hold the output of a `Task`. `Result`s are stored into a JavaSpace by the compute server.

3) Compute Server: Compute servers are processes that monitor one or more JavaSpaces for `Task`s to run. They repeatedly take a `Task` object from the space, run its `execute()` method, and write the `Result` to the space.

4) Compute Client: Compute clients generate the `Task`s and write them into a JavaSpace. These clients, or other processes, gather the `Result` objects and combine them, if needed, to form the final output of the distributed computation.

Using JavaSpaces, each iteration of the compute server loop is wrapped in a Jini transaction so that if a failure occurs, at any time, the current task is automatically returned to the JavaSpace as if it was never taken.

Most texts on JavaSpaces contain a description of a similar master-worker system since this is one natural use of JavaSpace technology [1, 5, 6].

3 Unresolved issues

Remote access to files is crucial to the operation of SSS. It is expected that the tasks in SSS will require the reading and writing of large files containing data arrays and that these files will not be stored on the files systems of the SSS servers. We are currently developing software to handle this requirement.

The biggest unresolved issue for SSS is network security, including mutual authentication, authorization, and communications integrity for both SSS clients and servers. The latest release of the Jini protocols (version 1.2), as well as all previous releases, provide no security beyond what is provided by the JVM in which it runs. Fortunately, the next major release of Jini is focused on adding a security framework to Jini. We expect security issues to be sufficiently addressed by the time we attempt to deploy the initial SSS system.

4 Current State of SSS

The SSS system is being developed using Java 1.4 and Jini 1.2. The SSS compute server is currently capable of running any task that does not require remote file I/O. The server runs as a stand-alone Java process and has not yet been integrated into a screen saver, although it does act like a screen saver in that it runs a separate thread that displays a simple animation on the local display. To keep track of the tasks in the system, each task submitted includes unique identifying information such as the PI (primary investigator), the project name, computation name within that project, and task number within that computation. Each SSS Server can request that preference be given to tasks from a particular PI and/or project. Basic statistics are maintained for each SSS Server, each project, and each computation, such as the number of tasks submitted and completed and the total CPU time used. All of this information is kept in the SSS JavaSpace for access by any interested process.

An important piece of SSS, remote file I/0, is being tested externally and will be integrated into SSS as it becomes functional. Our SSS file I/O package is designed as a Jini service (modeled after Sun's experimental ByteStore service, but using RMI), and currently has basic functionality. Clients can read and write remote files, and can also create and destroy remote files. This service is built using Keith Edwards' Service Writer's Toolkit (http://www.kedwards.com/jini/swt.html) which provides management for persistence, administration, and leasing. Correctness testing and performance tuning are not yet complete.

Our initial major SSS application will be a Monte Carlo computation in *ab initio* quantum chemistry. This application was chosen for its highly parallel structure and the ability to easily adjust the size of the individual tasks. The original Fortran version of this application is currently being translated into Java.

References

[1] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[2] David Gelernter. Linda in context. *Comm. of ACM*, 32(4):444–458, 1984.

[3] David Gelernter. Generative communication in Linda. *ACM Trans. Prog. Lang. and Sys.*, 7(1):80–112, 1985.

[4] William L. George. Screen Saver Science™. NIST. `http://math.nist.gov/mcsd/savg/parallel/screen` Accessed 9 Aug. 2002.

[5] Steven L. Halter. *JavaSpaces Example by Example.* Java Series. Prentice Hall, 2002.

[6] Sing Li. *Profession Jini.* Wrox Press, Chicago, Il, August 2000.

[7] James S. Sims et al. Accelerating scientific discovery through computation and visualization II. *Journal of Research of the National Institute of Standards and Technology*, 107(3):223–245, 2002. May-June issue. `http://nvl.nist.gov/pub/nistpubs/jres/107/3/cnt107-3.htm`.