# Build a Compute Grid with Jini™ Technology

DECEMBER 2004

# Table of Contents

SECTION 1

# Introduction

Although the power of computers to process data has progressed at an astounding rate over the past decades, the problems to which people wish to apply this compute power have exploded in magnitude as well. A common approach to solving a demanding compute problem is to employ an algorithm that allows multiple processors to work on the problem in parallel, and to run this algorithm on a either a symmetric multiprocessing (SMP) computer, or more recently, on a compute grid that harnesses the collective processing power of multiple computers distributed over a network. The ready, low-cost availability of high network bandwidth and networkable CPUs, coupled with improvements in software networking technologies, has enabled the creation of capable, lower cost compute grids today. The ability of such compute grids to easily and affordably scale to meet the needs of large scale computation problems makes this approach to parallel processing appealing to a wide range of users today.

### ABOUT JINI™ TECHNOLOGY

Jini technology is an open software architecture that enables Java dynamic networking for building distributed systems that are highly adaptive to change. It can be used to create technology systems that are scalable, evolvable, and flexible, as typically required in dynamic runtime environments.

Jini technology was originally created by Sun Microsystems, and was contributed by Sun to the Jini Community℠ in 1999. It is freely available and is advanced by members of the Jini Community through the open Jini Community Decision Process.

Many patterns and system architectures have been developed for performing parallel processing on a compute grid. One popular approach centers around a master-worker arrangement of networked grid nodes. In such a system, one or more of the nodes are designated as masters while the other nodes act as workers. When a client submits a computation job to a master node, that master node divides the computation into smaller chunks, or tasks, and distributes those smaller jobs to workers. The workers then compute those tasks and return their results to the master. When all such jobs are completed, the master assembles the results and returns them to the client.

This paper describes how Jini™ technology and JavaSpaces™ technology are used to implement the master/worker pattern to perform parallel computation on a compute grid. Jini technology, because it is designed to help people build distributed systems that are highly adaptive to change, can simplify and reduce the costs of building and running a compute grid. The JavaSpaces service, a component of Jini technology, provides a powerful yet simple way to coordinate parallel processing jobs.

SECTION 2

# A Jini Technology-Based Compute Grid Architecture

The Jini technology-based approach to parallel computation described by this paper involves three kinds of participants: *masters*, JavaSpaces services (*spaces*), and *workers*. In its most basic form, a master decomposes a job into discrete *tasks*. Each task represents one unit of work that may be performed in parallel with other units of work. Tasks are associated with objects written in the Java™ programming language ("Java objects") that can encapsulate both data and executable code required to complete the task. The master writes the tasks into a space, and asks to be notified when the task results are ready. Workers query the space to locate tasks that need to be worked on. Each worker takes one task at a time from the space and performs the task computation. When a worker completes a task, it writes a *result* back into the space, and attempts to take another task. The master takes the results from the space, and reassembles them, if necessary, to complete the job. This process is shown in Figure 1.

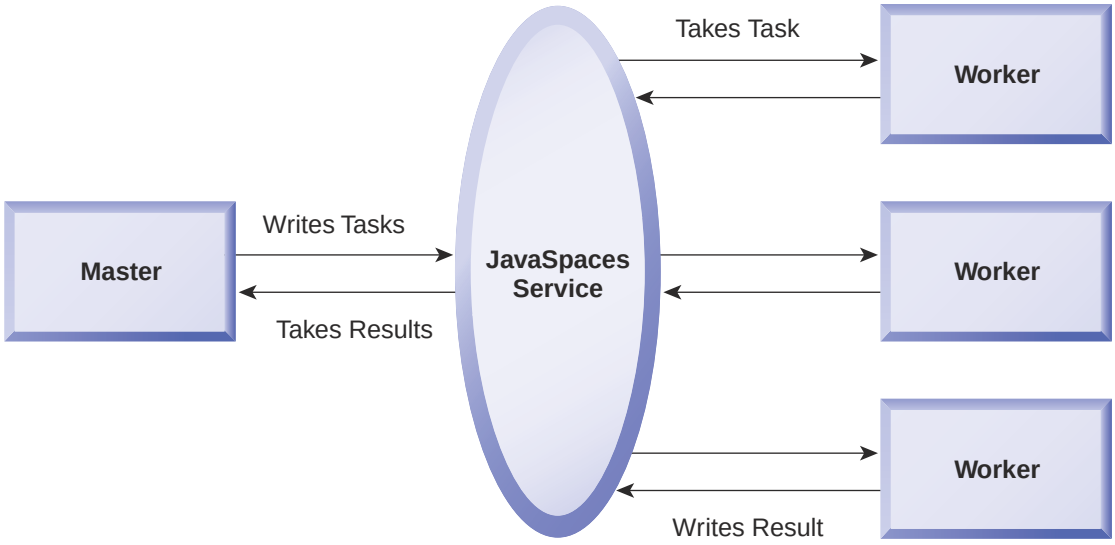


Figure 1. The Basic Master/Worker Pattern Implemented using Jini Technology and the JavaSpaces Service

## FLOW CONTROL

### Communications

One fundamental challenge is simply coordinating all the activities of such a system. How do you assign tasks to the workers? How do workers communicate their results back to masters? If a job succeeds, how are the results reassembled? Beyond the coordination challenges presented by a single job are the challenges of running multiple jobs. To obtain maximum use of the compute resources, you want to minimize worker idle time. Can multiple jobs be run in parallel? If so, how do you keep the tasks from one job separate from the tasks of other jobs?

The centerpiece of this compute grid architecture, the JavaSpaces service acts as the switchboard through which all of the grid's distributed processing is coordinated. The space is the primary communication channel between masters and workers. The master sends tasks to the workers, and the workers send results back to the master, through the space. More generally, the space is also capable of providing distributed shared memory capabilities to all participants in the compute grid. As will be described in a case study example later in this paper, entries may be used to maintain information about the state of the system, information that masters and workers can access to coordinate a wide range of complex interactions. But simplicity is what makes the power of this Jini service most appealing: four basic methods (read, take, write, and notify) provide developers with all the capabilities necessary to coordinate distributed processing across a compute grid.

The question of how to assign tasks to workers is easily resolved by the JavaSpaces service interaction paradigm. Workers look for task entries in a space using *templates*, which are entries that have some or all of their fields set to specified values that must be matched exactly. Remaining fields are left as wildcards—they are not used in the lookup. Each worker looks for and takes entries from the space that match the task template that it is capable of executing. In the most flexible model, generic workers each match on a template that features an "execute" method, take a matching entry, then simply call the execute method on the taken task to perform the work required. In this *worker pull* model, tasks need not be assigned to workers from any centralized coordination point; rather, the workers themselves, subject to their availability and capabilities, determine which tasks they will work on when.

The JavaSpaces service notify method is often used by masters to help them track the return of results associated with tasks that they put into the system. The master provides a template that can be used to identify results of the tasks that it put into the space, then registers with the JavaSpaces service to be notified[1] when a matching result entry is written into the space.

To distinguish between tasks, implementations of the basic compute grid architecture generally place a unique identifier into each task and result entry they write to the space. This enables a master to match each result to the task that produced it. Most implementations further partition the unique identifier into a job ID and a task ID. This makes it easy for workers and masters to distinguish between tasks and results associated with different jobs, and hence serves as a simple technique for allowing multiple jobs to run on the compute grid at the same time.

1. JavaSpaces service notifications are a specific example of the more general concept of distributed events. For more information about how Jini technology supports event notifications in a distributed system, refer to *Appendix A—A Quick Introduction to Jini Technology.*

The best way to manage work through a compute grid often depends on the sort of work that is being processed. For example, some computations may require that a particular task be performed before others. How do you prevent a task from being run before the tasks it depends on have been completed? How do you control the number of jobs that are running on the grid at one time? To keep the system busy, you may wish to queue jobs up in advance so they run as soon as computation resources become available. If so, how do you accomplish this? The case study examples that follow demonstrate how particular compute grid implementations have leveraged the features of the JavaSpaces service to address questions like these.

### Task Distribution

The most flexible compute grids are able to run different computations on different nodes at the same time, and to run different computations on a single node over time. To allow this flexibility, a compute grid needs to employ generic workers that can be equipped dynamically to handle whatever work needs to be processed at any given time. The challenge presented is fundamental to sound compute grid design: how do you distribute tasks to workers, including the data and code needed to execute them?

Using a JavaSpaces service-based grid model, this is accomplished fairly simply: because JavaSpaces service entries represent Java objects, entries offer a natural medium for delivering both the code and data required to perform a task. It is common for the serialized form of JavaSpaces service entries to be annotated with a codebase URL. Leveraging this capability, a master places both the data and an associated codebase annotation into a task entry which it writes to the space. When a worker takes a task from the space, it deserializes the task and dynamically downloads the code needed to perform the task work.

## ROBUSTNESS

### Partial Failure and Transactional Integrity

A fundamental challenge of distributed systems, including compute grids, is dealing with partial failure—a failure of a part of the system. Although today's computers are highly reliable, the more computers you bring together in a compute grid, the more likely it is that at least one of them will be in a failed state at any point in time. What are the implications of a partial failure in a compute grid? What happens to a task if the worker that is processing it crashes? What if the network or other communication channel between components goes down? How do you keep track of which task the worker was working on when that worker failed? How do you ensure that the failure does not corrupt any transactions that depend on the successful completion of the interrupted task? How do you ensure that the interrupted task is assigned to a different worker? Equally important, how can a crashed worker be re-introduced into the running system? To function effectively, compute grids must expect, tolerate, and be able to recover from partial failure.

The secret to surviving the failure of a worker midstream in a computation is to perform worker tasks within the context of a transaction. A worker can, before it takes a task, start a new transaction on a Jini transaction manager service. The worker will then receive a transaction service proxy for the transaction, and can obtain a transaction lease[2]. The worker must renew that transaction lease periodically to keep the transaction live.

2. For more information on Jini technology distributed leasing, see *Appendix A—A Quick Introduction to Jini Technology.*

The worker then passes the transaction proxy to the JavaSpaces service when it performs the take operation that returns a task. In this manner, the task is taken under a Jini distributed transaction. When the computation is complete, the worker writes the result to the space, passing along the same transaction proxy. The result is written under the same transaction. Finally, the worker commits the transaction via the transaction proxy, and the result becomes available for the master to take.

If a worker takes a task under a transaction, but crashes before completing the computation, writing the result back to the space, and committing the transaction, the worker's lease on the transaction will eventually expire. Once the lease expires, the transaction manager will automatically roll back the transaction, and the task that had been taken by the failed worker will reappear in the space. That task will then be available to be taken by a healthy worker.

Reintroducing a recovered worker back into the system is virtually effortless. A restarted worker does not attempt to complete the task it had been performing when it failed; that task will have been made available to other workers as part of the failed transaction rollback. Instead, like any other running worker, the restarted worker simply asks the JavaSpaces service for a new task to work on, and continues to process tasks as before.

Several strategies are used by different JavaSpaces service implementations to protect the compute grid's system state in the event of a JavaSpaces service failure. Using one popular approach, the JavaSpaces service itself is backed by a persistent store that protects the compute grid's system state in the event of a failure. On restart following a failure, the space automatically repopulates itself with all the entries it held at the time of the failure, and masters and workers resume their interactions with the space. Several commercially and freely available JavaSpaces service implementations feature support for space clustering, offering another approach to safeguarding a system against the threat of a JavaSpaces service failure.

### Security

In many environments, grid security can be vital. How do you ensure only approved jobs can be run on the grid? How do you ensure that only valid workers can take tasks out of the space and put results back into it? How can workers be sure that the code they download can be trusted before it executes? How do you prevent unapproved parties from viewing the results of jobs? How can you ensure that tasks and results are not tampered with while in transit over the network between workers, masters, and spaces?

The Jini technology security model extends the Java platform security model to enable secure interactions between Jini clients and services. The Jini technology security model delivers authentication, authorization, integrity, and confidentiality to interactions between software components that communicate using remote calls over a network. In addition, the Jini technology security model ensures that downloaded code, around which Jini technology is fundamentally designed[3], can be trusted.

In a Jini technology-based master/worker compute grid, masters and spaces can authenticate one another, as can workers and spaces, before engaging in any interactions. Based on what principal a master authenticates as, the JavaSpaces service can determine whether or not to allow that master to write tasks into the space and/or to take results out of the space. Similarly, the JavaSpaces service can authorize workers (or not authorize them)

3. For more information about how Jini technology uses downloaded code, see *Appendix A—A Quick Introduction to Jini Technology.*

to take and write task entries based on the principal that the worker authenticates as. Workers can determine whether or not they trust the JavaSpaces service, based on the credentials that the space presents, before taking and executing tasks from the space.

Network communications between clients and services can also be secured to ensure that confidentiality and integrity are maintained. The Jini technology security model allows a wide range of security protocols to be used, including those provided by Kerberos or Secure Sockets Layer (SSL).

## PERFORMANCE

### Throughput

The master/worker pattern provides the best throughput for problems that have a high computation to communication ratio: the time the workers spend doing actual computation far exceeds the time workers spend communicating over the network with the rest of the system. This ratio is influenced most directly by the grain size of individual job tasks relative to the size of a job as a whole. If you divide the job into a large number of very small tasks that can each be computed quickly, the handshaking between the parts may consume more computing resources than the actual task computations. If you divide the job into a small number of very large tasks, you could end up with fewer tasks than there are workers, thereby underutilizing the computation resources. Somewhere in between is a range of grain sizes that will yield an acceptable communication to computation ratio.

Once a job has been broken down into appropriately sized tasks, the computation to communication ratio can be further tuned by designing tasks to use network resources efficiently. For example, in cases where all task computations require the same data or code to execute, network communications can be improved dramatically by caching or storing the required code or data locally on worker machines.

Another way to improve throughput of the system is to minimize worker idle time. One approach to keeping workers busy is to ensure that they are capable of performing any task that needs to be executed. Jini and Java technologies, as applied in the master/worker pattern described here, enable task specific data *and code* to be delivered to generic workers when the workers are ready to perform a task. As long as there is work to be done, generic workers like these will never be idle.

### Dynamic Scaling

Most compute grids will need to adapt to changes in load over time. You may want to use more workers to accommodate increased demand, or to reduce the number of workers participating in the grid if they are being underutilized. How can you add and subtract workers from the grid dynamically, without stopping, reconfiguring, and restarting the system? If the load demand on your system is irregular, you may want to automate a dynamic scaling response. How do you incorporate more or fewer workers into your system depending upon actual need, without requiring human intervention?

Again enabled by the JavaSpaces service, loose coupling between masters and workers greatly simplifies the design of applications that scale dynamically to utilize resources currently available on a changing compute grid infrastructure. Because masters do not communicate directly with workers, they need not necessarily know how many workers there are, or where on the network workers are located. Likewise, workers need not know where the master is, or even whether there are one or multiple masters. As mentioned previously,

removal of a worker, even one that is actively processing a task, can be managed elegantly using distributed transactions. All of this means that there is no need to restart or reconfigure tasks that are moving through the system in order to accommodate the dynamic, runtime addition or removal of workers.

Similarly, the JavaSpaces service, because it does not assign tasks to workers, does not need to know anything about the number or location of workers in the system at any time. The JavaSpaces service will serve any number of workers that make requests of it, and so it too can accommodate the dynamic, runtime addition or removal of grid workers.

Finally, Jini technology supports dynamic service discovery[4], which enables Jini clients to locate and use Jini services on the network, regardless of where either the client or the service is located. In the case of the compute grid architecture, workers are Jini clients and spaces are Jini services. As such, any new worker added to the system will automatically find the JavaSpaces service, regardless of where the worker or the space is located.

In combination, these Jini technology features enable dynamic scaling of an operational compute grid. To remove a worker, simply unplug it. To add a worker, simply connect it up and switch it on. Jini technology facilitates making these changes while the system is running, without needing to restart or reconfigure operating masters, workers, spaces, or tasks moving through the system.

The simplicity of this approach to dynamic scaling lends itself well to automation: a system monitor can be set to start and stop workers automatically, in response to observed changes to some measure of load on the space.

Another aspect of scalability to be considered is that of the JavaSpaces service itself. Entries are generally not large resource consumers, and so a single JavaSpaces service can often support the scalability needs of most enterprise-class compute grids. JavaSpaces service entries are intended to point to, rather than embed, large data sets or executables. This means that entries that represent even very large task executables or result data sets can be passed via a JavaSpaces service with minimal impact on the space's resources. All of this notwithstanding, should the load of tasks and results passing through a JavaSpaces service tax the space too heavily, additional JavaSpaces services can be added to the system. Because JavaSpaces services are Jini services, workers and masters, being Jini clients, can dynamically find any number of them on the network, and can process tasks and results through multiple available spaces.

### Load Balancing

To achieve highest performance and efficiency, processing must be balanced across a compute grid so that some workers are not unnecessarily overburdened with work while others remain idle. Generally, load balancing techniques are used to distribute the computation load across workers in a way that aims to accomplish this goal. Unfortunately for efficiency, most compute grids are composed of a heterogeneous mix of more and less intensive tasks that are performed on a range of faster and slower worker machines, making the job of load balancing difficult to do well using traditional approaches. How can you prevent a situation in which some workers are overwhelmed with work while others are underutilized? How can you ensure that a task is distributed to a worker that will be free to do the work most quickly?

4. For more information about dynamic service discovery, see *Appendix A—A Quick Introduction to Jini Technology.*

In the Jini technology-based compute grid architecture described here, the JavaSpaces service takes care of load balancing among the workers. Workers obtain task entries by performing a take operation on the space. The *take* semantic is what facilitates load balancing. The flow of tasks to workers is regulated by the workers themselves, since they take tasks out of the space only when they are ready to work on them, and as a result load is balanced naturally across the system. The JavaSpaces service guarantees that each entry is only taken once. Faster workers will take more tasks. Slower workers will take fewer. Faster workers with difficult, time-consuming tasks will also take fewer tasks. Crashed workers will take none.

## ADMINISTRATION

Many typical systems administration tasks are eased by the use of Jini technology as the underlying architecture of a compute grid. For example and as discussed, system scaling, failure recovery, and distribution of executable task code are accomplished with little to no human intervention. Dedicated load balancing systems are unnecessary, eliminating the need for an administrator to manage them. An administrator can add, remove, start, stop, update, or relocate any master or worker at any time and without impacting the ongoing operation of any other components in the compute grid.

### Provisioning Many Workers

One issue to consider when building a compute grid is how to most effectively administer the worker computers. The cost burden of administering a worker is multiplied by the number of workers you have in your compute grid. The most pressing question is how do you get the worker's code to the worker's computer? How do you install and upgrade the code of the worker itself—the code that takes a task, executes the task, and writes the result back to the space?

You could, of course, have a human administrator manually install the worker code on each machine, and repeat the process for each update. But that approach does not scale well as the number of workers in your compute grid increases. An alternative approach that is quite common automates the process. On each worker, a small worker bootstrap program is installed. When this program starts, it downloads the latest generic worker code from a code server and runs the worker code. To update all the workers, one need only update the code server and restart the workers.

SECTION 3

# Compute Grid Case Studies

Many commercial and scientific organizations are today using compute grids that they have developed using Jini technology and the JavaSpaces service. The master/worker pattern described in the previous section has been applied, with many subtle variations on the theme, to reduce the cost and speed the completion of large computation tasks in a wide variety of applications. This section introduces four of these compute grids, and explores some of the variations of the general master/worker pattern that they exemplify.

## BUILDING MODELS IN THE FACTORY OF THE FUTURE

### Tuning a Uniform Compute Grid

Valen Technologies helps financial services companies, primarily commercial insurance underwriters, to better predict risk and more accurately score, screen, and price policies than traditional underwriting methods allow. For an insurance company, often a mere 8% of policies generate 80% to 90% of claims. Thus, companies that can improve their ability to predict the risk of a policy based on the data supplied on the policy application can improve their profitability, lower their overall risk, be more competitive, and charge their customers prices for insurance that are commensurate with the actual risk. Valen Technologies can help enable insurance carriers to do that, and as a result can help lower losses and improve profitability.

Here's how it works: An insurance company supplies Valen Technologies with a set of samples, which consist of data for actual policies (e.g., policy data, claims data, billing data, etc.) and a set of risk factors (e.g., weight of car, driver's experience, and zip code for auto insurance). Each sample combines all of the policy information and risk factor data associated with a single policy. A sample set includes samples that are of the same policy type and share the same set of risk factors. The risk factors for a set of samples, typically numbering in the thousands, describe a multi-dimensional space in which each sample occupies one point. Associated with each sample (each point in the hyperspace) is a *loss ratio,* a measure of insurance risk that is calculated by dividing the total claims against the sample policy by the total premiums collected for it.

The solution that Valen Technologies provides back to the insurance company is, at the end of the day, a mathematical decision support model that is based on the sample data. Similar to the way cartographers take a number of data points in three dimensional space and draw a contour map, Valen Technologies analyzes sample data and generates multi-dimensional insurance risk maps. Because they are multi-dimensional, however, risk models cannot be presented as simple contour maps; instead, they are described as complex mathematical expressions that correlate insurance risk to thousands of risk factors in multi-dimensional hyperspace. The mathematical models produced are in turn used by Valen's Risk Manager application, given data from a policy application, to provide an underwriter with a *risk score* that predicts the risk represented by that particular policy.

To produce a risk model, Valen must discover a mathematical expression that best characterizes the sample data. Each of the thousands of risk factors included in the sample set are variables that could influence the model alone or in interaction with others, making the space of all possible models so vast that it cannot be searched by brute force alone. The key to producing risk models successfully lies in determining which of the risk factors are the most predictive (typically, only 5% of risk factors are predictive), and focusing on developing the model around them. Valen Technologies uses artificial intelligence techniques and computational learning technology to cycle through different models iteratively, observe the results, learn from those results, and use that learning to decide which model to iterate next. This process occurs hundreds of thousands of times in the process of finding the most accurate model.

Evaluating hundreds of thousands of candidate models demands a significant amount of computation. To enable this processing to take place in an acceptable time frame without incurring the costs of SMP servers, Valen Technologies built a parallel processing system on a compute grid using Jini technology and the JavaSpaces service. Using the Jini architecture enables Valen's system to build risk models, which previously took months of labor-intensive work to develop, in a matter of weeks. By building risk models rapidly, Valen enables its customers to have access to up-to-date decision support data that can help them retain a competitive edge, avoid adverse selection, and stay aligned with shifting market conditions.

Valen has built a conceptual "factory" that generates and tests many model ideas in search of one that will best match a sample data set. In Valen's vernacular, one *job* is one attempt at modeling a given set of samples. A job is composed of multiple *iterations.* An iteration is a set of *tasks.* First, an *optimizer* determines what combinations of task parameters to try and creates an iteration, typically a set of between 2,000 and 20,000 tasks, to run through the compute grid. Those tasks are stored in a database. A master, responsible for getting those tasks completed, places them into the space and then monitors the space, waiting for completed results to return. Workers take tasks from the space, along with any data needed to compute those tasks, and calculate the results. Since the same task execution code is always used, it is pre-loaded onto all workers. Tasks are sized so that it typically takes a worker a few minutes to compute the result. Workers then place the results back into the space as a result entry, which contains a statistics object that shows the fitness of that task's approach. The result entry also contains the entire compute task entry, including a task ID that allows the master to match the result with its task. To complete the computation of all tasks in an iteration typically takes on the order of hours, and when all task results have been returned to the space the master takes them from the space and stores them in a database. Based on its analysis of results of the completed iteration, the optimizer is then able to create a new generation of tasks and initiate a new model iteration. This process continues until a satisfactory model is calculated, typically involving computation of tens of thousands of tasks in total and completing in a few weeks.

In Valen's compute grid application, each task is a candidate model, and each task is trying to achieve the same goal: prove that it is the best model. The optimizer applies different algorithms to the sample data, inspects the results, and creates a new generation of tasks—a new iteration. Through this process, the factory attempts to weed out non-predictive risk factors, to select the best algorithm (or combination of algorithms), and to optimize the performance of the chosen algorithm by tuning its parameters. The process stops once the model has ceased improving for 10 iterations. As a last step, some kerning is performed to make sure the simplest model is chosen of those that are equally good.

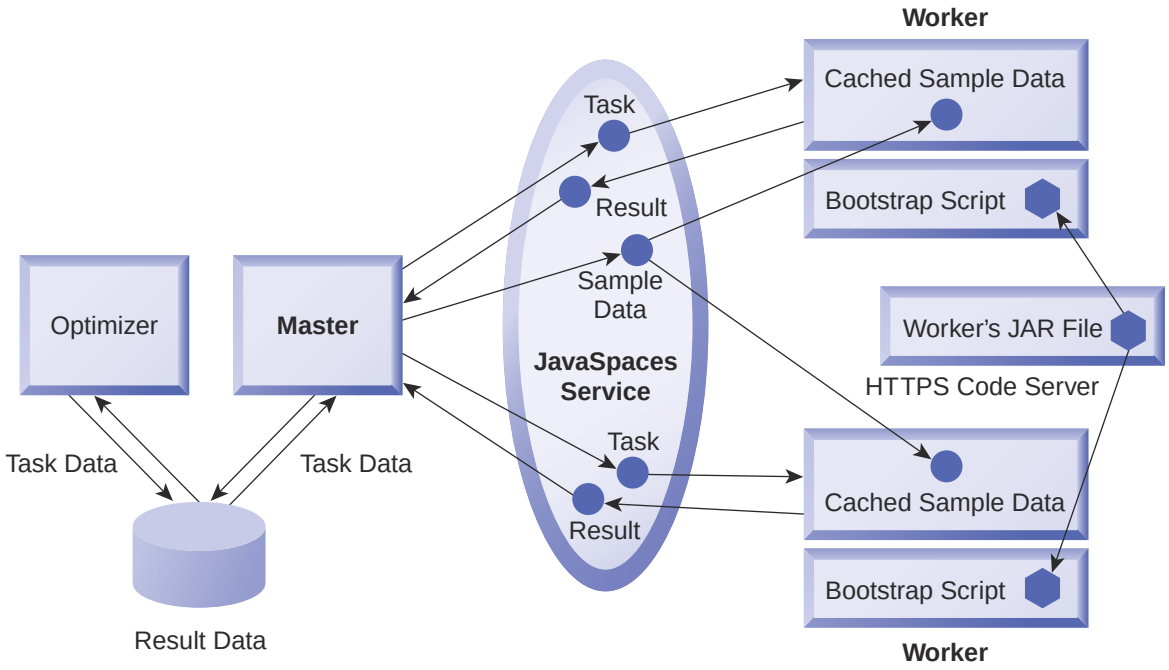


Figure 2. The Factory of the Future

The Valen Technologies implementation is an example of a common variation of the master/worker compute grid architecture in which the computation code used by workers is uniform over the grid. Because Valen's grid is dedicated to one kind of computation, calculating the performance of candidate models, the task code needed by all workers is the same. One of the basic issues raised by the compute grid architecture is how to administer the workers, in particular, how to upgrade the worker code. In Valen's case, because the task code is specific to calculating the performance of candidate models, worker and task code are updated together. In other words, when a task arrives at a worker, all the code needed to execute the task is already available at the worker. Rather than install upgrades by hand at each worker, Valen places a JAR file of the worker and task code on an HTTPS server. Whenever a worker server is restarted, a UNIX bootstrap script runs and downloads that JAR file and makes a local copy.

One special consideration of this particular compute grid application is that the worker needs a lot of data—the sample data plus the algorithm parameters—to calculate a result. Since the same sample data is used by all tasks for a particular job and this data does not change, Valen opted to place the sample data into its own entry, rather than into each individual task entry. Workers download the sample data from the space once per job. They cache the sample data and use it again and again as they compute results. This approach is highly efficient, saves time and memory, and allows Valen to maintain a high computation to communication ratio of more than 100 to 1.

## SEARCHING AND ALIGNMENT OF UNKNOWN SEQUENCES WITH KNOWN DNA AND AMINO-ACID SEQUENCES

### Processing Grid Tasks with Non-Java Code

At the Livestock Issues Research Unit of the US Department of Agriculture–Agricultural Research Service (USDA–ARS)[5] in Lubbock, Texas, Dr. Scot E. Dowd and his team of scientists use an algorithm called Basic Local Alignment Tool (BLAST) to compare newly discovered, unknown DNA and protein sequences against a large database, more than 3 gigabytes, of known sequences. BLAST searches the database for sequences that are identical or similar to the unknown sequence. This process enables the scientists to make inferences about the function of the unknown sequence based on what is understood about the similar sequences found in the database.

Advances in sequencing technology have resulted in a large influx of unknown sequences that need to be analyzed. Many projects at the USDA–ARS's Livestock Issues Research Unit, for example, involve as many as 10,000 unknown sequences, each of which must be analyzed via the BLAST algorithm. Fortunately, the implementation of the BLAST algorithm used by the USDA–ARS scientists, which is written in C++, has been optimized for performance. But despite this optimization, given the multi-gigabyte size of the database of known sequences, a single search on a modern desktop computer takes several minutes. A project involving 10,000 unknown sequences requires about three weeks to complete on that same desktop computer.

To perform BLAST searches more quickly, scientists at the Livestock Issues Research Unit built a compute grid based on Jini technology and the JavaSpaces service[6]. On top of that grid, they created S-BLAST, a distributed form of the BLAST algorithm that reduces the amount of time required to perform searches for large sets of unknown sequences. Armed with S-BLAST's compute grid architecture and 17 commodity computers, projects that previously took three weeks to complete can now be finished in less than one day.

The entry point into S-BLAST is a *blast provider* Jini service that allows users to create jobs and display the results. The blast provider service passes the job to a *jobber* service, which creates tasks and distributes them by placing them in a space. The workers are called *taskers*. A tasker takes tasks from the space and makes a system call, passing the task data representing the unknown sequence to the BLAST program. If a copy of the known sequence database is installed on the worker node, the BLAST program will search through sequences in that local copy. If not, the worker will be dynamically assigned to a file server that contains the sequence database. The result of the BLAST invocation is an XML document, which is collected by the tasker and written back to the space. Results from all tasks associated with a job are collected from the space by the jobber service. The jobber waits for all tasks to come back for one job, then sends the cumulative job results back to the blast provider. Alternatively, the user can request that the jobber return results to the blast provider in chunks, such as 100 results at a time. The blast provider presents the results to the user and/or saves the results to the local disk. This architecture is shown in Figure 3.

5. Mention of trade names or commercial products in this article is solely for the purpose of providing specific information and does not imply recommendation or endorsement by the U.S. Department of Agriculture.

6. S-BLAST is built on top of SORCER (Service Oriented Computing EnviRonment), a federated grid infrastructure that represents an evolution of the concepts developed in the FIPER project, a program founded by the National Institute of Standards and Technology (NIST). For more information on the SORCER project, visit http://sorcer.cs.ttu.edu.

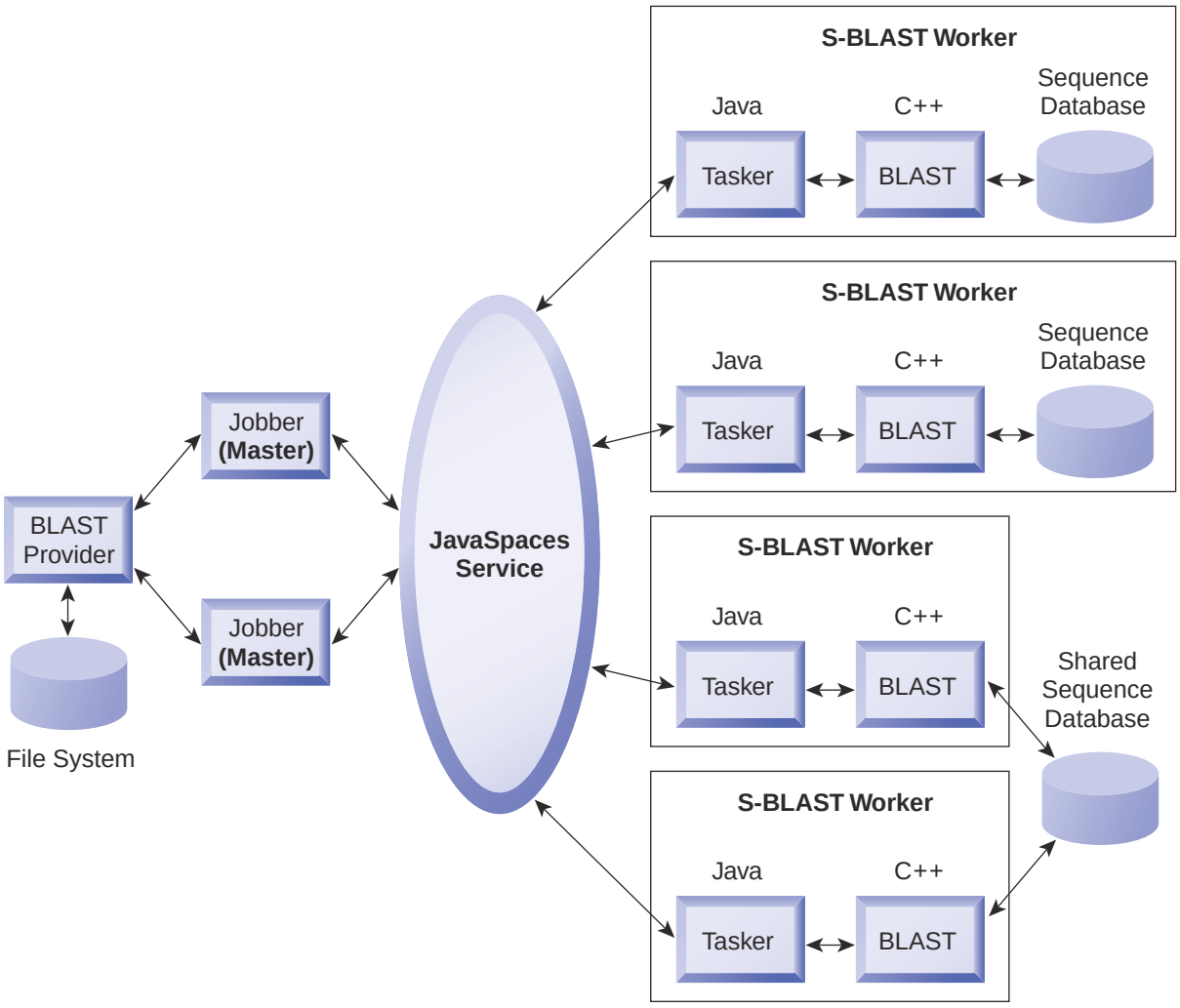


Figure 3. The S-BLAST Architecture

The USDA–ARS implementation is another example of the common variation of the compute grid architecture in which the computation code used by workers is uniform over the grid. In the S-BLAST application, the code being executed is the same across all tasks, because all tasks are doing the same thing: running the BLAST algorithm on an unknown sequence. Dedicated to performing a single sort of computation, the USDA–ARS grid has been tuned to perform faster by maintaining a local copy of the BLAST code on each worker.

This application further highlights that, although Jini technology and the JavaSpaces service are built on top of Java technology, this compute grid architecture can be used for distributing the processing of non-Java code. The BLAST algorithm used by S-BLAST is a native executable compiled from C++ source code. The system uses taskers to represent the BLAST code to the Jini system. Taskers are Jini clients, and as such they can dynamically find and interact with Jini services, including the JavaSpaces service from which they take

tasks. The taskers simply make a system call to invoke the BLAST algorithm and collect the XML document that results. The tasker then places that XML document as a string in a result entry, and writes that back to the space.

## COMPUTING THE TRAVELING SALESMAN PROBLEM

### Distributing Database Updates Over a Grid

In Argentina, 400 sales reps of Nobleza Piccardo, part of British American Tobacco, drive vans filled with cartons of cigarettes to do direct sales with customers. Each sales rep carries a handheld computer that tells him which customers to visit, in which order, what to deliver to each customer, what to try to sell to them, and what promotions to offer. As they visit each customer, reps enter information about deliveries and sales into their handhelds. Each night the reps bring their trucks back to one of ten distribution centers, place their handheld computers in synchronization pods, and push synchronization buttons. Data move from the handhelds up to an enterprise system, and, done for the day, the reps go home. The next morning at 6:00 a.m., the sales reps arrive back at work and push the buttons on their handhelds to synchronize them again. But this time, data move from the enterprise system down to each handheld. The handhelds now contain instructions on where to go and what to do for the new day.

Between the times the 400 handheld computers are synchronized, at night and again in the morning, Nobleza Piccardo's Field Order Management application must do a lot of processing. First, each handheld includes new information about deliveries and sales that must be incorporated into the master enterprise database. The number of total customer visits by the 400 sales reps averages around 17,000 a day, with each visit generating 30 to 40 database transactions. Then, based on current inventory, current orders, and current marketing strategies, the system must create a route for each of the sales reps for the next day—which of 47,000 direct sales customers to visit, in which order, what to deliver, what marketing promotions to offer—all before the sales reps return at 6:00 a.m.

To accomplish this significant amount of processing each night and with minimum cost, Nobleza Piccardo built a compute grid based on Jini technology and the JavaSpaces service. The compute grid was built to replace a legacy system of mid-range servers. This system was peaking out under the load of the nightly batch processing, which took about seven hours to complete and consumed 97% of available processing resources. At $150,000 per server, and in the face of a 300% Argentinean currency devaluation, an incremental expansion of the same architecture was not economically attractive. As a result, Nobleza Piccardo opted to create a lower cost compute grid architecture to distribute the batch processing across a farm of 33 entry level, dual processor $3000 servers. Today, running on this Jini technology-based compute grid, the nightly batch processing is accomplished in 30 minutes to two hours.

In addition to processing the nightly Field Order Management operations, Nobleza Piccardo's compute grid is being used today by the Central Office portion of their system. Central Office applications are designed around the notion of a business object, called a *unit of processing* (UoP). UoPs can be distributed, which makes it possible for them to run on the compute grid. If an application is asked to perform a long-running computation that will strain the server with a heavy load, the application can offload some of its work to the compute grid: the UoP can be sent to the compute grid by writing it into the space as a compute grid task. For example, Nobleza Piccardo has a UoP that is responsible for generating invoices for as many as 20,000 customers. This UoP first divides the work into smaller UoPs and then sends those UoPs into the space as

tasks to be processed by the compute grid. By offloading long-running business processes like this one, other applications running on the same application server remain highly available to other users. As a result, overall availability of many applications is improved, including customer and account management, order taking, order processing, stock taking, route management, delivery management, pricing and invoicing, collection and bank reconciliation, promotions management, and more.

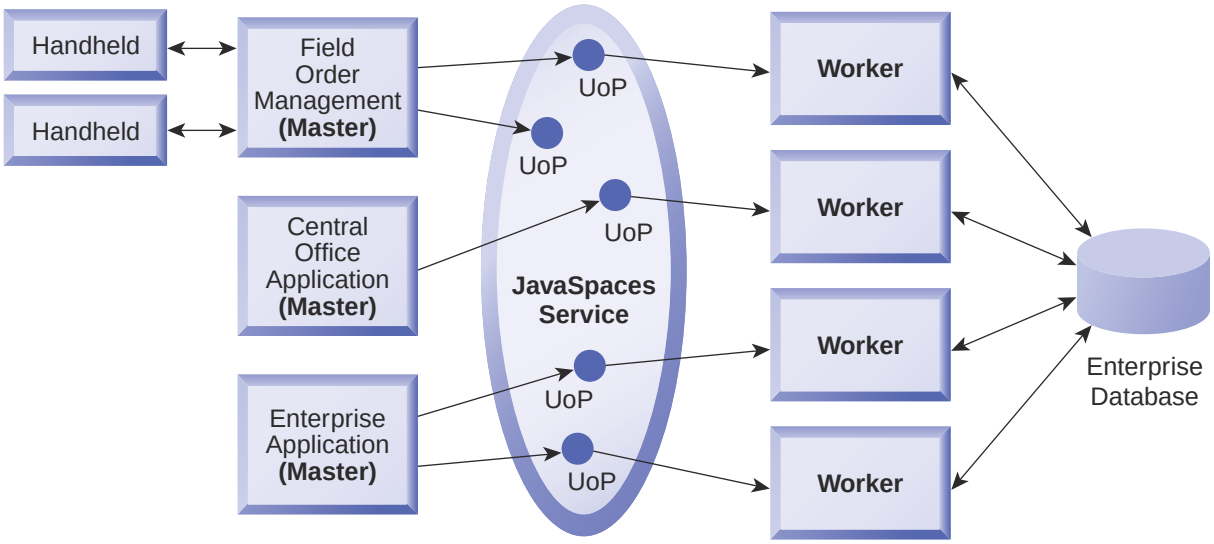


Figure 4. Central Office and Field Order Management Applications Run Tasks on the Nobleza Piccardo Compute Grid

Field Order Management tasks are also designed around UoPs. When handhelds are synchronized in the evening, a UoP for each handheld is written to the space. These UoPs take care of incorporating the sales and delivery data uploaded from the handheld database into the enterprise database. Each handheld contains a small relational database. This entire database is uploaded to a handheld manager, which creates the UoP and writes it to the space. Based on the uploaded data, which usually represents new orders, the following processes need to be executed:

- Recalculate customer state (credit limit, sales history, next suggested order, and other statistics).
- Recalculate sales rep objectives for the next day (for example if Monday will be a holiday, the objectives for Friday will be increased to reflect the situation so that the customer will not run out of stock on Monday or Tuesday).
- Define a sales route for the next day (sales routes are static, but they are affected by sales rep replacements, vacations, illness, and so on).
- Determine product availability, and based on a rule system, define how to fulfill the order (for example, if the customer requested 10 units of Lucky Strike Promotion Pack and there is no availability of that product in stock, then deliver 10 units of Lucky Strike standard edition instead).
- Determine next day routes and van load based on the processed orders.
- Print invoices and other commercial documents, such as credit/debit notes.

No reassembly is necessary after all the smaller tasks complete, because each task writes its results directly into the enterprise database. This highlights one of the key features of this particular application of the compute grid architecture—the workers talk directly to the enterprise database. In the generic master/worker compute grid pattern, workers report their results back to the master by writing result entries to the space, and the master collects the results and updates a database. In the Nobleza Piccardo architecture, however, the main goal is to distribute enterprise processing that is itself quite database intensive, so in general the results of the workers' processing is stored directly in the enterprise database. Each UoP task represents one or more atomic database transactions.

## ASTRONOMICAL DATA MINING

### Ordered Task Execution on a Generic Compute Grid

At the Canadian Astronomy Data Centre in Victoria, Canada, scientists work with a large repository of images from telescopes. Over 350,000 telescope image datasets, more than 25 terabytes of data, are archived in their database, and the number grows daily as new images are added. To these images the scientists apply various suites of calibration and analysis processing. The calibration processing takes raw image data from the archive, removes telescope instrument signatures, and places the corrected images back into the archive. The analysis processing is part of a scientific data mining project called the *Canadian Virtual Observatory* (CVO). Through this processing, scientists extract information from the archived image data and populate a CVO data warehouse. Because the desired amount of processing is too large to be accomplished economically by a single computer, and because the desired processing is generally parallelizable, the Canadian Astronomy Data Centre performs these operations on a compute grid using Jini technology and the JavaSpaces service.

Although the Canadian Astronomy Data Centre's image processing problems are usually parallelizable, they often do not lend themselves to division into tasks that can be computed in arbitrary order. In many cases the output of one computation is required as input for another. To accomplish this they model each job as a directed acyclic graph of nodes. In fact, they call each compute job a *graph.* For example, Figure 5 shows a graph that represents one compute job. A task is associated with each node in the graph. The output of a *root* task, positioned at node one in the graph, is needed as input for the computations positioned at nodes two, three, four, and five in the graph. The root node task must therefore be completed before tasks positioned at nodes two, three, four, and five can begin. Tasks two, three, four, and five can execute independently and in parallel, but must all be completed before the task positioned at node six can be executed.

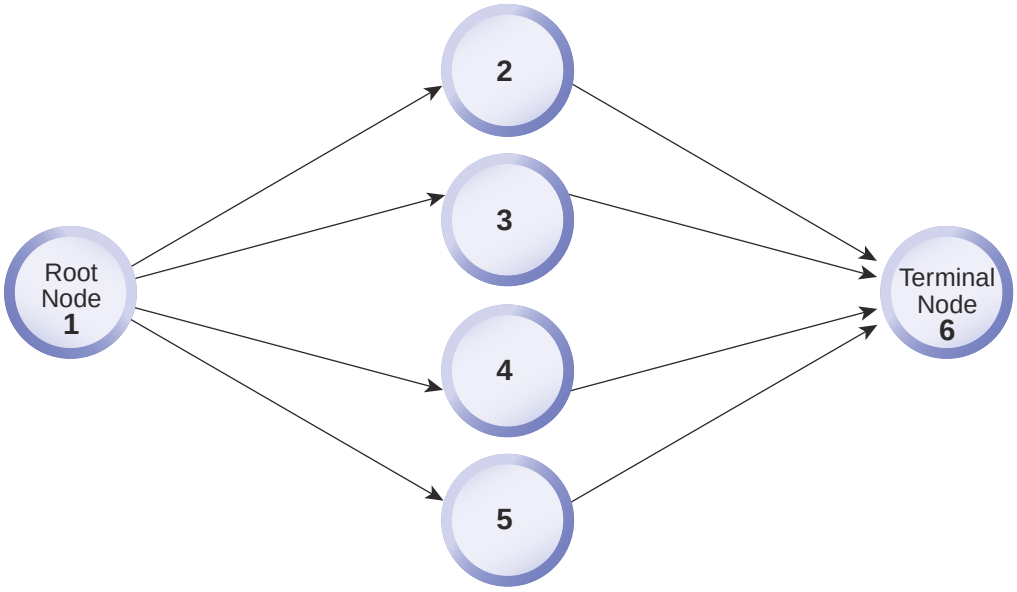


Figure 5. An Acyclic Directed Graph of Nodes

Complex compute jobs like these, consisting of ordered and interdependent tasks, cannot be processed in the same way that simpler jobs can be, without regard to the order that tasks are processed. Workers cannot take and process tasks arbitrarily; instead, they must be able to identify tasks that are ready to be worked on at a point in time. The system must therefore track the progress of work through the graph, and make clear to workers which tasks can be taken at any time. The Canadian Astronomy Data Centre compute grid does this very well, by using the space as a place to track and maintain the state of graphs and of the work flowing through them, and by using smart workers to update and maintain these entries.

A graph is represented to the compute grid using a combination of entries in the space. Each node in the graph is represented by one node entry in the space, and each edge (or arrow) in the graph by one edge entry in the space. Each node entry contains either a task object that holds work yet to be done, or a result object that holds results of completed work associated with that node in the graph. To keep track of which tasks are ready to execute, the space also contains a priority queue data structure.

The priority queue consists of a set of *simple queues,* one for each priority level. Each simple queue is a linked list of queue item entries. Each queue item contains an item ID, a next item ID, and a payload. The simple queue is a linked list because each item's next item ID effectively points to (provides the item ID of) the next item in the simple queue. The number of simple queues in the priority queue is equal to the number of priority levels, which need be at least as great as the maximum depth of any graph in the space. For example, the graph shown in Figure 5 has a graph depth of three, and would therefore require at least three simple queues in the priority queue, one for each of priority levels one, two, and three. Node one, the lone root node in the graph shown in Figure 5, is at depth one in the graph. Nodes two, three, four, and five are at depth two, and node six is at depth three.

When a task is ready to be executed, the task entry will be placed in the simple queue representing the priority equal to the task's graph depth. For example, when node one in Figure 5 is ready to execute, the task for node one will be placed as the payload in a queue item entry, and that queue item entry will be placed at the end of the simple queue representing priority one. When the tasks for nodes two, three, four, and five are ready to execute, they will be placed into the simple queue representing priority two (this is illustrated in Figure 6). When the task for node six is ready to execute, it will be placed in the simple queue representing priority three.

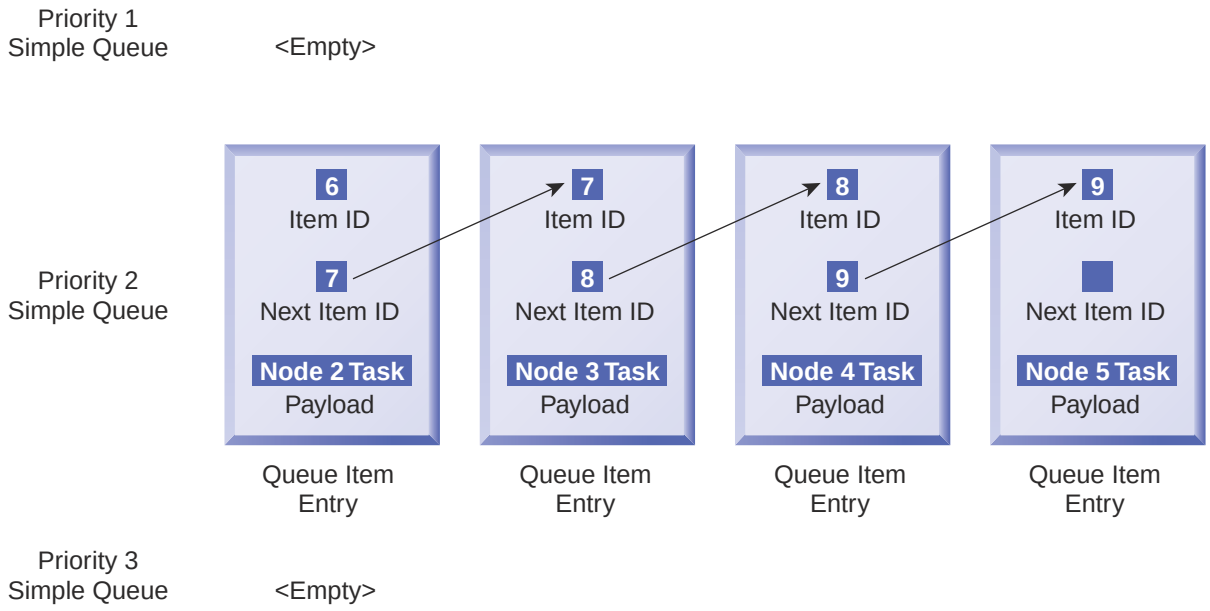


Figure 6. A Priority Queue with Three Simple Queues

Workers have been implemented with the intelligence needed to read and update the graph and queue entries. Rather than taking a task from the space arbitrarily, a worker first examines the priority queue and takes the next task from the highest priority non-empty queue. After a worker performs the task computation, it places the result back into the same node from which it took the task. For example, once a worker has finished computing the task from node one in Figure 6, the worker will put the node one result entry into the space. In addition, the worker will update any nodes that depend upon the just completed task, indicating that the just completed task is available as input. The worker that provides the final input for a particular task will place the task, which is now available to run, directly into the priority queue. For example, the worker that completes node one's task will update nodes two, three, four, and five, indicating that node one is complete. Since nodes two, three, four, and five each depend only on node one, the worker will also place the tasks for nodes two, three, four, and five into the priority queue.

The graphs that describe the compute jobs that run on the Canadian Astronomy Data Centre compute grid can become quite complex; the graphs may even transform themselves during execution of the compute job. For example, galaxy morphology measurement, the process of finding candidate galaxies in an image and classifying them based on their shape, can require a very large graph with hundreds of nodes. A *source*

*detection* task scans a large image for likely galaxies, yielding a list of *sources:* coordinates and sizes for candidate galaxies. The source list and original image are then used as inputs by an *image chopper* task, which chops the image into pieces of various sizes, one piece per source. This is shown in Figure 7.


Figure 7. Source Detection Feeds Data into the Image Chopper Task

The image chopper creates a brand new graph that directs the individual source images it produces into an equal number of *galaxy morphology* tasks, which classify each candidate galaxy based on its shape. The outputs of the galaxy morphology tasks, which can be run in parallel, are directed into a single *merge* task. In effect, the image chopper task creates a new graph with many root nodes, the galaxy morphology tasks, and one terminal node, the merge task. The terminal node is mapped into the original graph at the image chopper's original position. This is shown in Figure 8.

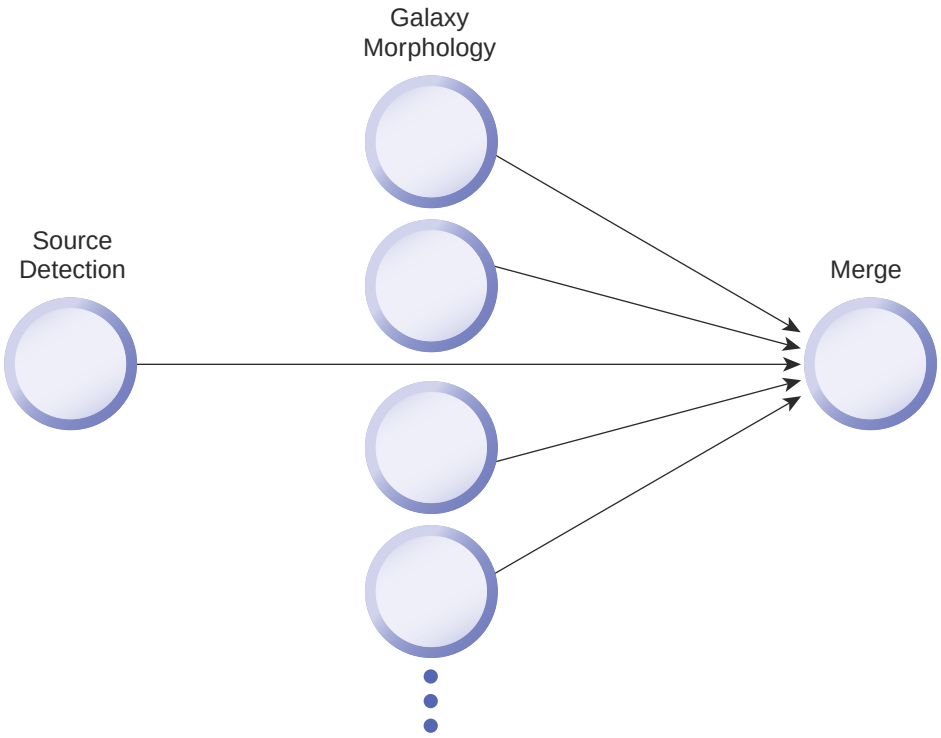


Figure 8. The Image Chopper Task Replaces its Node with a New Graph

The previous example highlights that in the Canadian Astronomy Data Centre's system, any task can dynamically alter the graph by adding nodes. This means the entire graph structure need not be known up front before launching a job. The graph can morph during the course of the job. This dynamic graph modification capability enables check pointing, loops, branching, and many other kinds of standard computing concepts in a distributed computing environment. It highlights that the master/worker pattern can be applied to problems that, although parallelizable, still require some flow control.

As the graphs describing the Canadian Astronomy Data Centre's compute jobs are complex, so are the implications of a failure. A failure of any task should propagate to all tasks that directly or indirectly depend on the failed task. To accomplish this error propagation, a worker, upon catching an exception thrown by the task, places a special error result back into the node. This error result, which contains information about the thrown exception, is also placed into the result of every subsequent node in the graph immediately, without processing their tasks.

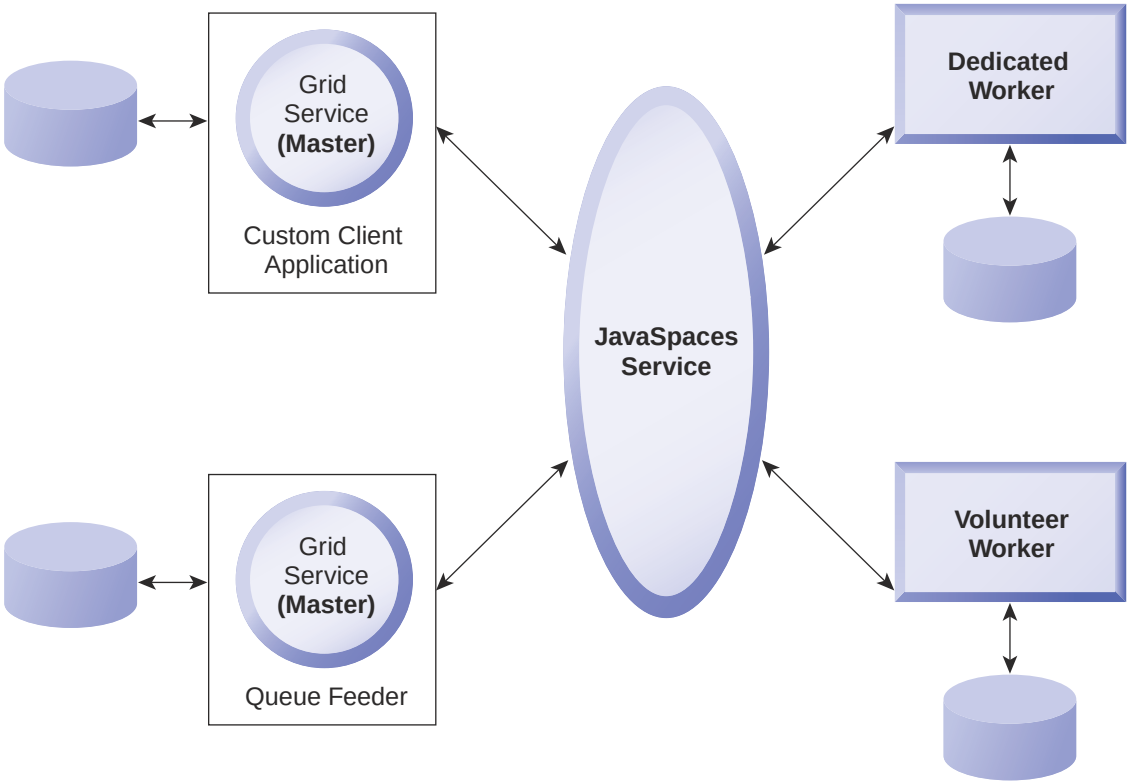


Figure 9. The Canadian Astronomy Data Centre Compute Grid

The architecture of the Canadian Astronomy Data Centre's compute grid is shown in Figure 9. Users of the grid access it through a *grid service*, a Jini technology service that provides an API for the compute grid to client programs. Most commonly, the client application pulls information from a database and, based on that information, creates tasks that it passes to the compute grid via the grid service. The application then inspects the results that come back, and, generally, updates the database.

The Canadian Astronomy Data Centre uses two kinds of workers. A set of about 30 computers are dedicated to hosting two workers each for the compute grid. The Centre also provides software to enable desktop computers to volunteer to be compute grid workers during off hours. Because failure is managed through Jini transactions, employees can kill volunteer worker processes when they arrive back at work. The task the killed process had been working on will reappear in the space and be taken by another worker.

Given that their compute grid consists of an average of 60 dedicated workers plus a varying number of volunteer workers, having a few hundred graphs in the space at any one time is sufficient to maximize the compute grid's throughput. Sometimes the scientists at the Canadian Astronomy Data Centre may have as many as 100,000 to 200,000 graphs they want to run. In such situations they use a *queue feeder* client to manage the placement of graphs into the space. The queue feeder client coordinates between the database, which contains the graphs to run, and the grid service. The queue feeder wakes up once per hour, counts the number of graphs currently running in the space, and puts more graphs into the space until the number of graphs in the space has risen to a few hundred.

SECTION 4

# Conclusion

Jini technology enables the creation of distributed systems that are highly adaptive to change, and is well suited for use as the underlying architecture of compute grid applications. Jini technology enables compute grid masters and workers to find and connect to JavaSpaces services in dynamic operating environments, which simplifies the runtime scaling and failure recovery of compute grid applications. Extending the Java platform programming model to recognize and accommodate partial failure, Jini technology enables the creation of compute grid applications that remain highly available, even if some of the grid's component parts are not. Robustness is further enhanced with Jini technology's support for distributed systems security. And finally, the JavaSpaces service contributes a simple yet powerful coordination point that facilitates task distribution, load balancing, scaling, and failure recovery of compute grid applications.

Successful Jini technology-based compute grids are today serving the needs of a wide range of users, including many in financial services, defense, logistics planning, life sciences, image processing, scientific research, and manufacturing. The many problems to which Jini technology-based compute grids are currently being applied include modeling, simulations, data processing, rendering, and pattern matching.

Jini technology, including the JavaSpaces service, is readily available from both commercial and non-commercial sources. For more information about Jini technology, visit www.jini.org.

APPENDIX A

# A Quick Introduction to Jini Technology

Jini technology is designed to help people build systems that deal gracefully with a fundamental characteristic of any distributed system: change. This appendix presents a quick introduction to Jini technology, and shows how Jini technology helps manage change within an operational distributed system.

## SERVICE PROXIES

Jini technology is used to build systems that exhibit a service-oriented architecture. Functionality on the network is divided into, and provided by, discrete services. In Jini technology, each service is represented by an object called a Jini service *proxy.* If a client wants to use a service, the client obtains a proxy for the service. The client then invokes methods (or functions) on the service proxy, and the proxy takes care of fulfilling the promised service, including possibly talking across the network to a server or other entity.

## DISCOVERY AND LOOKUP

A fundamental kind of change in a distributed system happens whenever new participants join the system or others move around the network. If a client wants to use a service that just arrived or was recently relocated on the network, how does that client find the service? Jini technology addresses these issues by providing a *lookup service* where clients can find services they want to use. In addition, Jini technology provides a set of *discovery protocols* that enable Jini clients to find lookup services with no prior knowledge of their location. If a client desires to use a service for the first time, the client can find that service's proxy in the Jini lookup service. If that service later moves, the client can still find the service via the Jini lookup service. If the lookup service moves, the client can locate that lookup service via the discovery protocols. Once a client has obtained a proxy to a desired service via a Jini lookup service, the client uses the proxy to interact with the service directly, without further involvement from the lookup service.

## PARTIAL FAILURE

One of the main problems that must be dealt with in a distributed system is partial failure—the failure of a part, but not all, of the system. That's because, in a distributed system, some of the computers may be working while others are not. Alternatively, all the computers could be operational, but their interconnection network may fail. From the perspective of one computer, such network partitioning may appear as a failure to other computers.

### Distributed Leases

Perhaps the most powerful tool Jini technology offers to address partial failure is the *distributed lease.* When a client requests that a service maintain state on behalf of the client, the service provides a lease to the client. The lease is a period of time during which the service agrees to maintain the state, to the best of its abilities. The time period of the lease may be determined solely by the service or negotiated by the client and service. During the period of the lease, a client may cancel the lease, enabling the service to free resources associated with maintaining the state. Alternatively, the client may request that the lease be renewed. The service may renew the lease for the requested period or a shorter period, or may refuse to renew the lease at all. If the lease period lapses with no renewal request received from the client, the lease expires and the service is free to release the resources associated with the maintained state.

How do leases address partial failure? If a client requests a service to maintain state on its behalf, and then crashes, that client will be unable to renew its lease. Eventually the lease will expire, and the service will clean up, freeing any resources it maintained on behalf of the crashed client. If the client and service are both healthy but the network between them goes down, the client will be unable to renew its lease. Both the service and the client will then know that the client's lease expired and their interaction is no longer in effect.

For example, when a service provider registers with a Jini lookup service, that service provider receives a lease on that registration. The service will remain registered in the lookup service only as long as the service provider continues to renew the lease. If the service provider crashes, the lease will eventually expire and the lookup service will deregister the service proxy.

### Distributed Events

Another example of leasing involves *distributed events*, which are a Jini technology mechanism that allow parties to interact asynchronously. A client can register with a service to receive notification of events from that service. The ability to register a remote listener with a service provides the distributed computing equivalent of the observer-observable object-oriented design pattern. When a client employs a remote listener in a service, the client obtains a lease on its event registration from the service. As in any Jini service-client interaction, the client must renew its event registration with the service before that lease expires. As long as a client maintains a valid event registration with a service, that service notifies the client of events asynchronously, in the form of a remote method invocation on the registered listener. Should the client become removed from the network, that client's event listener registration would expire, releasing the service from any further obligation to maintain the listener or send event notifications to that client.

### Distributed Transactions

Another Jini technology weapon against partial failure is *distributed transactions.* A transaction allows a set of operations to be grouped in such a way that they either all succeed or all fail, and the operations in the set appear from outside the transaction to occur as a single operation. Transactional behaviors are especially important in distributed computing, where they provide a means of enforcing consistency over a set of operations on one or more remote participants. If all the participants are members of a transaction, a failure of one of the participants—a partial failure—will generally cause the transaction to abort, thereby ensuring that no partial results are written.

## PROTOCOL INDEPENDENCE AND MOBILE CODE

Another way implementations of Jini technology facilitate change is with mobile code. Most distributed computing systems require that code needed to perform computations on each task be installed on every worker node prior to starting the computation. Some systems meet that requirement by making such code available to each worker via a shared file system. As new code becomes available, an administrator in such systems must move that code to a directory mounted on each worker. Jini relieves the administrator of that duty, and takes advantage of the classloader architecture of the Java Virtual Machine (JVM) to move code around the network at runtime.

When a service registers its proxy in lookup services, that proxy registration contains no code required to use the proxy; it contains only the serialized proxy instance and a reference to the list of network URLs from which the proxy's code can be obtained so the proxy instance can be reconstructed (deserialized) in a client JVM. Thus, when a client obtains a service's proxy during the Jini service discovery process, the client dynamically loads the proxy's code from those network locations. Such dynamic code mobility is possible due to the ability of the JVM to load classes from any network location. The Jini technology model allows implementations to fully exploit this JVM network class loading capability, and thereby support dynamic distributed computing with Java mobile code. Should a service's codebase location change, that service registers a new service proxy in Jini lookup services with the new code base location on the network.

Mobile code, in conjunction with the polymorphism that Jini technology brings to network services, allows a system developer to insert updated service code into a distributed system at runtime. For instance, a service developer may decide to change the protocol a network service uses from the Java Remote Method Protocol (JRMP) to the Simple Object Access Protocol (SOAP), popular in XML-based Web services. The service proxy's Java language interface remains the same, and the developer can develop a new implementation of that interface—instead of dispatching JRMP remote invocations, the new service proxy might send SOAP messages to a remote server. Once she completes the new service proxy code, the developer can register the new service proxy in lookup services. At the same time, the developer can cause the old, JRMP-based proxy's lease to expire in lookup services, perhaps by canceling the old proxy's lease. With Java technology's mobile code mechanism and Jini technology's network polymorphism, a client obtaining a reference to the new service proxy from Jini lookup services will dynamically download the new proxy's SOAP-aware code.

Alternatively, a developer may choose to leave both the JRMP and SOAP-based service proxies in Jini lookup services (by causing the service to renew both proxies' registrations). In that case, a client could choose the appropriate protocol to use for a service based on that client's capabilities, the available bandwidth, or any other criteria. As this example illustrates, mobile code allows Jini services to be agnostic with respect to communication protocols. It also illustrates the main way Jini technology facilitates updating a part of the distributed system while the system as a whole continues to run.

## SECURITY

One of the starting assumptions of Jini technology is that the network is not necessarily secure. To address this, Jini technology extends the Java platform's security model to define a new model that clients and services can employ to interact in a secure manner on an unsecure network. The Jini technology security model provides basic network security (authentication, authorization, integrity, confidentiality) for remote method invocations. The model also provides mechanisms for addressing security in the face of mobile code. For example, the model defines mechanisms for verifying object integrity and proxy trust.

The primary mechanism defined by the Jini technology security model that clients and services employ to express their network security needs is a system of *constraints*. The Jini technology contraint model, together with the security policy mechanism defined by the Java platform, is used to specify what a Subject can do, cannot do, and/or must do. The constraint model is extensible, and can be employed to express other types of needs, such as quality of service, in addition to needs that are related to security. Through constraints, a client or service specifies "what" sort of security they wish to employ, not "how" that desired security is provided; "how" is left as a detail of the implementation, to be configured at deployment time.

In addition to the constraint model, Jini technology's security architecture includes an enhanced implementation of the Java Remote Method Invocation (Java RMI) programming model. Java RMI is the standard mechanism provided by the Java platform for performing remote communication. Currently, the Java platform provides two implementations of the Java RMI programming model. One implementation employs the Java Remote Method Protocol (JRMP) for communication. The other implementation employs the Internet Inter-Orb Protocol (IIOP). Because neither of these implementations currently support Jini technology security or the constraint model, a third implementation of the Java RMI programming was created. This new implementation is referred to as Jini Extensible Remote Invocation (Jini ERI). Jini ERI, a complete implementation of Java RMI, is designed to be layered and pluggable, so that any layer of a remote communication can be configured at deployment time.

Jini technology offers a configuration model that supports deployment time configuration of clients and services—in particular, for deployment time security configuration. The configuration model supports the deployment time configuration of complex Java objects, not just strings and primitive types. The Jini technology configuration model is defined to be a subset of the Java programming language expression syntax, and provides for named entries and Java type checking. Using this configuration model, a service provider needs to implement a service only once. The provider can then change how the service behaves when the service is deployed. For example, it may be desirable to run a number of deployments of the service; one that communicates over an insecure channel, one that communicates using Secure Socket Layer (SSL), one that communicates over secure HTTP (HTTPS), and one that employs Kerberos. Using Jini technology's configuration model, the service provider does not have to change the service's code and then recompile. The service provider merely has to start the same service implementation four times, each with the appropriate configuration.

## THE JAVASPACES SERVICE

The JavaSpaces service is an implementation of the tuple space concept first proposed by David Gelernter at Yale University. A JavaSpaces service holds *entries*, typed groups of objects. An entry holds objects via fields. For example, a "task" entry could hold three objects: a job ID, a task ID, and a command object. Another entry, a "result," could hold a job ID, a task ID, and a result object.

A JavaSpaces service, often referred to as a *space*, offers four basic functions: write, read, take, and notify. You can write an entry into a JavaSpaces service, creating a copy of that entry in the space to be used in future read or take operations. Entries written to a space need not be unique; two or more entries in a space may contain exactly the same values. You can search for entries in a space using *templates*, which are entries that have some or all of their fields set to specified values that must be matched exactly. Empty fields are left as wildcards—they are not used in the search.

There are two kinds of search operations, read and take. A read request to a space returns either an entry that matches the template on which the read is done, or an indication that no match was found. A take request operates like a read, but if a match is found, the matching entry is removed from the space. You can also request a JavaSpaces service to notify you via a Jini technology distributed event when an entry that matches a specified template is written.

All operations that modify a JavaSpaces service are performed in a transactionally secure manner with respect to that space. If a write operation returns successfully, that entry was indeed written into the space. If a take operation returns an entry, that entry has been removed from the space and no future operation will read or take the same entry. In other words, each entry in the space can be taken at most once.

In addition, the JavaSpaces service supports a simple transaction mechanism that allows multi-operation or multi-space updates to complete atomically. This is done using the two-phase commit model under the default transaction semantics of Jini technology distributed transactions. The default transaction model in Jini technology is a transaction where all participants follow a two-phase locking protocol that ensures serializable transactions.

Entries written under a transaction may be read or taken under the transaction, but will not be visible outside the transaction unless and until that transaction commits. Entries taken under the transaction will no longer be available for taking both under the transaction and outside it—the entry will be taken from the space. However, if the transaction rolls back instead of committing, entries taken under that transaction will reappear in the space. Entries written into a JavaSpaces service are governed by a Jini technology distributed lease. If the lease expires, the entry will disappear from the space.

# Resources

For more information about their Jini technology-based analytic decision making software, visit the **VALEN TECHNOLOGIES** web site at www.valentech.com.

Visit the **USDA LIVESTOCK ISSUES RESEARCH UNIT** web site at liru.ars.usda.gov to learn more about or to download the S-BLAST application.

More information about **SORCER** (Service ORiented Computing EnviRonment), a federated grid infrastructure that underpins the S-BLAST architecture, can be found at sorcer.cs.ttu.edu.

Visit www.noblezapiccardo.com to learn more about **NOBLEZA PICCARDO** and the dynamic business that their Jini technology-based compute grid is helping to drive.

For more information about the **CANADIAN ASTRONOMY DATA CENTRE**, visit the Centre's web page at cadcwww.dao.nrc.ca.

More information about the **CANADIAN VIRTUAL OBSERVATORY** can be found at www1.cadc-ccda .hia-iha.nrc-cnrc.gc.ca/cvo.

Useful links to additional information about **JINI TECHNOLOGY**, including the JavaSpaces service, can be found at the Jini Community℠ Resource web page at www.jini.org/resources. Here you will find links to books, tutorials, Jini Community projects, email archives, Frequently Asked Questions, forums, and more that can help introduce Jini technology, its applications, and the Jini Community.

**LEARN MORE AT www.jini.org**

Visit the Jini Community℠ web site to learn about Jini technology or become involved with the Jini Community. The Jini Community comprises a wide range of members, including individuals, companies, organizations, and schools, who share and collaborate through Jini technology-focused development projects, discussion lists, and events.


This Jini technology white paper was produced by Sun Microsystems. The inventor and original contributor of Jini technology, Sun is an active member of the Jini Community and supports the growth of new markets enabled by Jini technology. Sun participates in the Jini Community's ongoing advancement of the technology and offers the Jini Technology Starter Kit free of charge to members of the Jini Community.