Distributed Application Management using Jini™ Connection Technology

The Jini™ Technology Enabled Applications Manager provides a framework, utilizing Jini™ and JavaSpaces™ technologies, for selecting and submitting native (i.e. not Java™ technology based) applications onto one of a set of compute resources, and for monitoring and controlling the resultant running jobs. The applications, compute resources, and running jobs are abstracted as Jini™ services.

The architecture illustrates the effectiveness of the Jini™ technology service model for abstracting both hardware (compute resources) and software (applications) as services on a network.

Introduction

The Applications Manager (hereafter called "JAM", for Jobs & Applications Manager) is a tool for selecting and submitting applications onto one of a set of compute resources and monitoring and controlling the resultant running jobs. The applications, compute resources, and running jobs are reflected as Jini™ connection technology services. JAM targets High Performance Computing (HPC) applications, which generally consist of native (not Java™ technology based[1]), compute−intensive codes. These codes generally run on one machine or a cluster of machines with each such compute resource abstracted at the Jini™ service level as one or more *queue services* that interact with an underlying native resource management system (RMS) such as Sun™ Grid Engine or Platform Computing's LSF. These queue services reflect the queue structure of the underlying resource management systems.

The JAM architecture has been developed as part of a collaborative effort between Sun Microsystems and the Swiss Center for Scientific Computing.

This paper presents an overview of the requirements for JAM along with a description of the JAM architecture.

Goals and Objectives

The JAM architecture provides a framework within which applications can be selected, launched, monitored and controlled in a unified manner, independent of the application or the properties of the compute resources and their respective resource management systems.

JAM can also be thought of as implementing aspects of a *distributed resource management system*. It provides a mechanism whereby a user can select an application from among a well−known set, and based on resource requirements both intrinsic to the application as well as supplied externally by the user, be presented with a list of candidate queues to which the user can choose to submit the application for execution. These JAM queues are associated with underlying computational resources that are themselves managed by native resource management systems. These computational resources are likely to be distributed both geographically and organizationally. Once the application has been queued to the underlying resource management system and it has launched their job, the JAM architecture allows that job to be monitored and controlled. This is all accomplished within JAM's unified framework.

This unifying framework provides a computing environment within which users are provided a simple, yet effective method for selecting from a set of available compute resources to run their applications. In addition, once a compute resources has been selected users are not exposed to the vagaries of the potentially differing resource

1 There is nothing in the architecture of JAM that specifically rules out its use for Java−based jobs, though some optimizations would be logical to consider for remote submission of such jobs (e.g. utilizing Java's dynamic classloading to make the code available at the target machine). We have not specified or implemented any such optimizations.

management systems controlling those resources. In essence, JAM allows users to utilize a set of potentially heterogeneous compute resources by presenting a resource independent, application level interface to the user.

Base Technologies

The two key technologies exploited in the JAM architecture are Jini™ technology and JavaSpaces™ technology. We discuss here only those salient features of these technologies that are helpful to understanding the architecture of JAM.

Jini™ Connection Technology

Given the distributed nature of JAM, it seems natural to take advantage of the distributed service model provided by Jini™ technology as the key abstraction for applications, jobs, and compute resources to allow these objects to register and interact with each other. The use of the Jini™ service model for interacting with application, queues and jobs provides the following benefits:

- The Jini™ distributed model meshes well with the distributed nature of JAM's applications and compute resources
- The Jini™ discovery model allows applications, compute resources, and jobs to be dynamic; changes in their state are spontaneously reflected in the lookup service
- The Jini™ infrastructure provides the necessary robustness and error models needed in a distributed system like JAM

An important aspect of Jini™ technology is the *service proxy model*. When a service back end registers itself with a Jini™ lookup service, part of what is registered is a *service proxy* object. This service proxy is downloaded to the client performing a lookup for the service. The proxy contains the code which implements the interface between the client and the service: the client's sole interaction with the service is via this service proxy. Another way to think of this is that the service provides its own driver, which is downloaded on demand and instantiated in the client.

Depending on the design of the particular service, the service proxy may or may not communicate with its service's back end, or other remote objects. The client doesn't know or care about this; it represents implementation details of the service.

JavaSpaces™ Technology

JAM also makes extensive use of JavaSpaces™ technology for handling and keeping track of submitted jobs as they progress through their life cycle. JavaSpaces™ technology provides a shared object store, along with notification capabilities, to provide a framework supporting loosely cooperating, distributed programming. This maps well to the distributed way in which JAM jobs are launched, monitored and controlled.

JavaSpaces™ services are Jini™ services, and implementations are provided as part of the Jini™ technology SDK from Sun. We utilize a private instantiation of a persistent JavaSpaces™ service implementation which we term a *Job Repository*.

JAM Architecture

We now describe the architectural components of JAM. First, we describe the services which make up the core of JAM: applications, queues, jobs and the Job Repository (JavaSpaces™ service). We then discuss the supporting components of the architecture, including the client−side code that integrates it all. Figure 1 shows a high−level overview of JAM's various components.

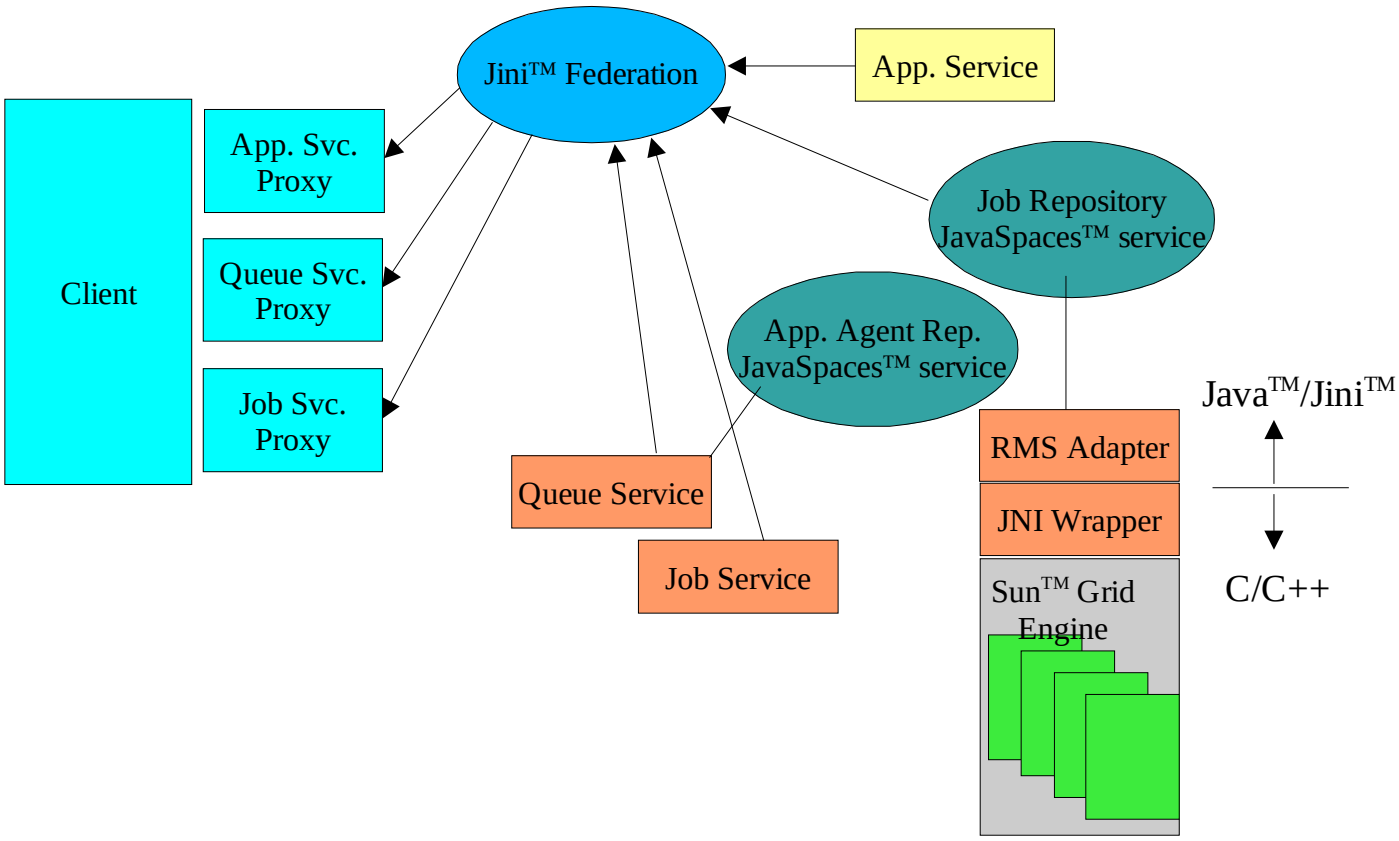

Figure 1: Overall JAM Architecture

Applications

There are two classes of applications supported by JAM.

Pre−registered applications: These are specific applications that have had Jini™ service wrappers generated for them, and that have been made known to JAM. For example, a commercial HPC code (NASTRAN, ANSYS, etc.) could have a Jini™ service wrapper written for it, making it available to submit via JAM.

User−defined application: This class represents applications which are specified by the user via a template that is registered with JAM.

There is a Jini™ service associated with the user−defined application, as well as one for each pre−registered application. The entirety of the application service is implemented within its service proxy. There is only a skeletal service back end for handling registration, leasing and administrative duties.

Registered applications are made available for selection by the user, as shown in Figure 2.

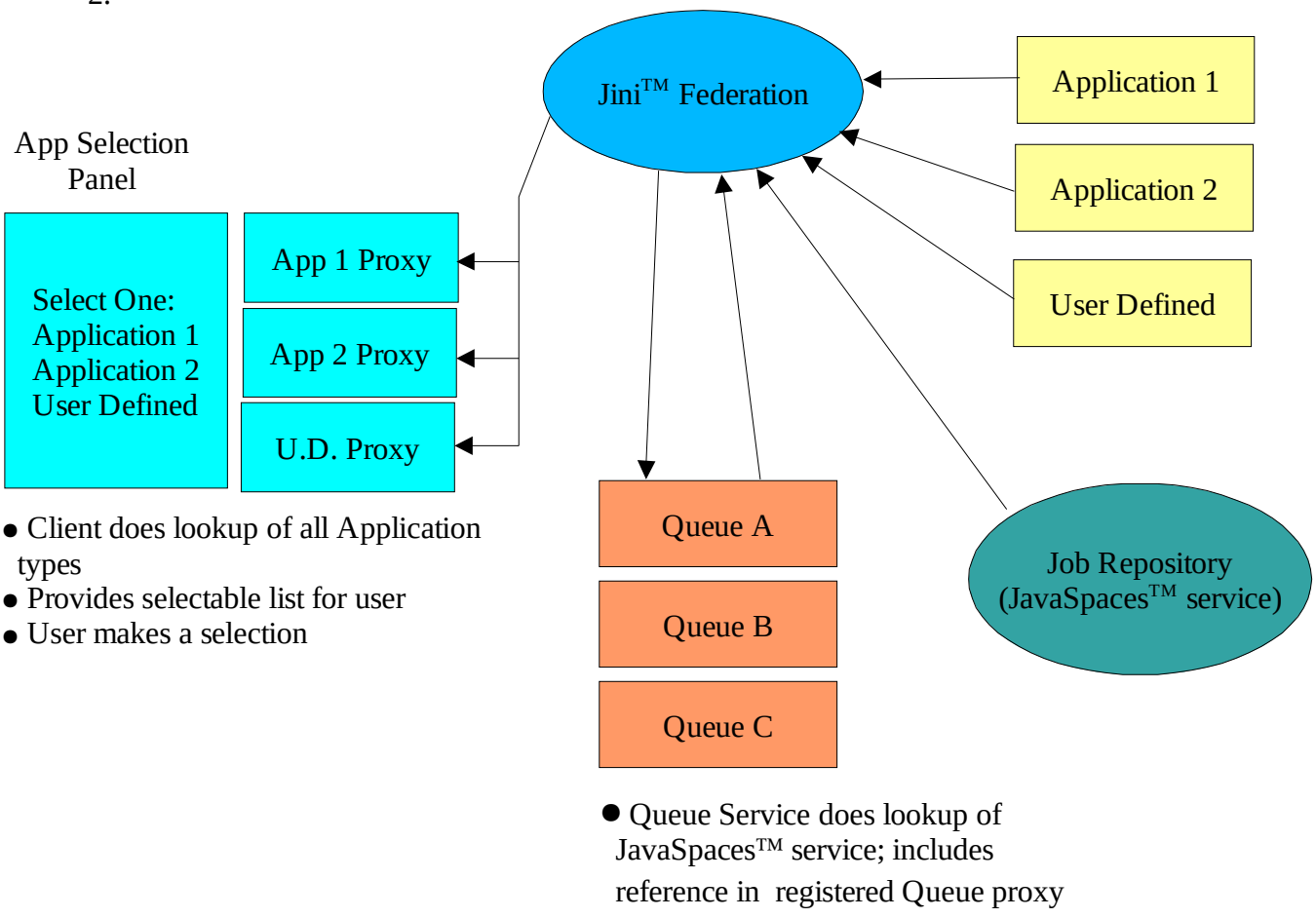

Figure 2: Queue service registration; Application Selection

Each application service includes a user interface pane for entering application and resource parameters. The application parameters include those which are application−specific, and therefore different for each registered application, and those which are application−independent and common to all applications in JAM. Applications can provide their own hard−coded values for any of the application−independent parameters, in which case, the user is not able to set or change the value.

The resource parameters are independent of both applications and resource managers; they represent attributes that are abstracted by all registered queue services, including both queue− and machine−level attributes.

Included within each application's proxy is an Application Agent. Its function is to perform filtering and optional ranking of the available queue services. There are two levels of filtering supported by the agent. The first utilizes Jini™ technology's attribute−matching lookup semantics to filter available queues based on the resource parameters entered by the user into the application service's user interface pane. The second, optional method involves application−specific evaluation code for further refining and ranking candidate queue services. This evaluation code is written to an Application Agent repository and run within the back−end of each queue service, providing an application−specific method for intelligent queue filtering and ranking. Since this code is run within the context of each queue service's back−end, it enables different filtering and ranking semantics (e.g. range checking) than is possible with standard Jini™ attribute matching.

This queue filtering occurs dynamically. As the user enters resource parameters, the list of candidate queue services being displayed is updated to match. Conversely, if the status of a queue service changes (either due to a change in value of one of its service attributes, or due to the service itself being removed or added) the queue service list is adjusted as appropriate to reflect this new information. This functionality is provided by the Jini™ infrastructure.

Job Repository JavaSpaces™ Service

The Job Repository is central to the architecture of both queue services and job services. All aspects of job submission, monitoring and control are handled through interactions with this private JavaSpaces™ service instantiation. Every job submitted to JAM is represented by a job descriptor within the Job Repository. Modifications to the job descriptors stored in the Job Repository trigger events in other JAM components, as described in the following sections.

RMS Adapter

The RMS Adapter is used to communicate between the Job Repository and the native resource management system. It is divided into two logical levels. The upper half of the Adapter is independent of the underlying native resource management system. It deals with making necessary modifications to the job descriptions in the Job Repository and

reacts to modifications made from elsewhere, forwarding the requests to the native resource management system.

The lower half of the Adapter is specific to the underlying resource management system and provides the actual link to the resource management system. In general, JNI wrappers are utilized to communicate between the Java™ technology−based upper half of the RMS Adapter and the native resource management system itself.

Examples of standard resource management systems include Sun™ Grid Engine and Platform Computing's LSF.

Application Agent Repository

The Application Agent repository is a JavaSpaces™ service instance which holds the optional, application−specific evaluation code for execution by the queue services as well as the results of the evaluation code run for each queue service.

Queues

Each registered queue service represents an underlying queue defined by a resource management system on a specific compute resource. The compute resource being managed by the resource management system may be either a single machine or a cluster of machines. It is important to note that the queue service is intended simply as an abstraction of the queue(s) which are generally part of the native resource management system; there are no queue semantics implemented at the JAM level.

Before a queue service first registers itself with the Jini™ technology federation, it performs a lookup to retrieve a reference to the Job Repository JavaSpaces™ service. This reference is included in the queue service's proxy object. This is noted in Figure 2.

Attributes of the underlying resource management system are reflected in the queue service attributes in a resource manager−independent manner. It is these service attributes which are matched against the resource parameters specified by the user to determine the list of matching candidate queue services for a specific job, as shown in Figure 3.

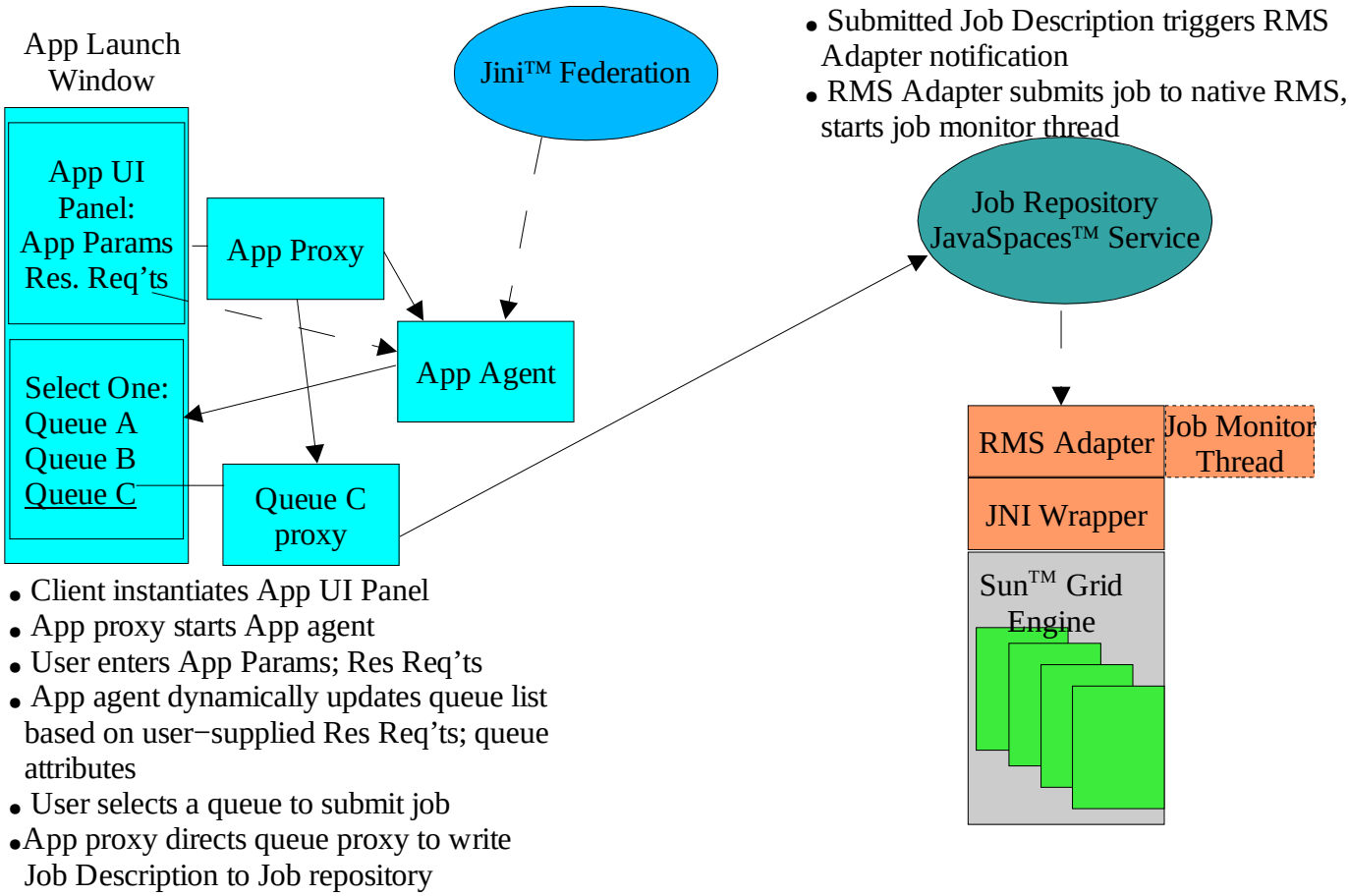

Figure 3: Queue filtering, Application submission

When submitting jobs for execution, each queue service proxy interacts with its associated compute resource through the Job Repository. The queue service proxy writes a new job description into the Job Repository to submit a job for execution. The RMS Adapter has registered for notification of new job description entries being written into the Job Repository and responds when so notified by submitting the job to the underlying resource management system. It then updates the job description in the Job Repository as the job's state changes during its life cycle.

Each queue service back end is registered for notification of Application Agent evaluation request arrival, at which point it copies the entry out of the Application Agent repository, executes the code therein, and writes back a new entry, containing the results for that queue service.

Jobs

Users can monitor and control their jobs once they have been submitted. Monitoring includes the ability to browse the current state of the job's execution (queued, running, finished, etc.) and resource consumption in addition to being able to interact with the job's I/O. Control includes the ability to suspend or signal a running job and to restart a

running or completed job. In addition, optional steering hooks can be provided by the application's service wrapper to enable application−specific steering of the running job as it executes. Application steering might, for example, involve dynamically changing one or more application input parameters.

Each job submitted by JAM is represented by its own job service. How and when this job service gets registered is now described. Reference Figure 4 in the following discussion.

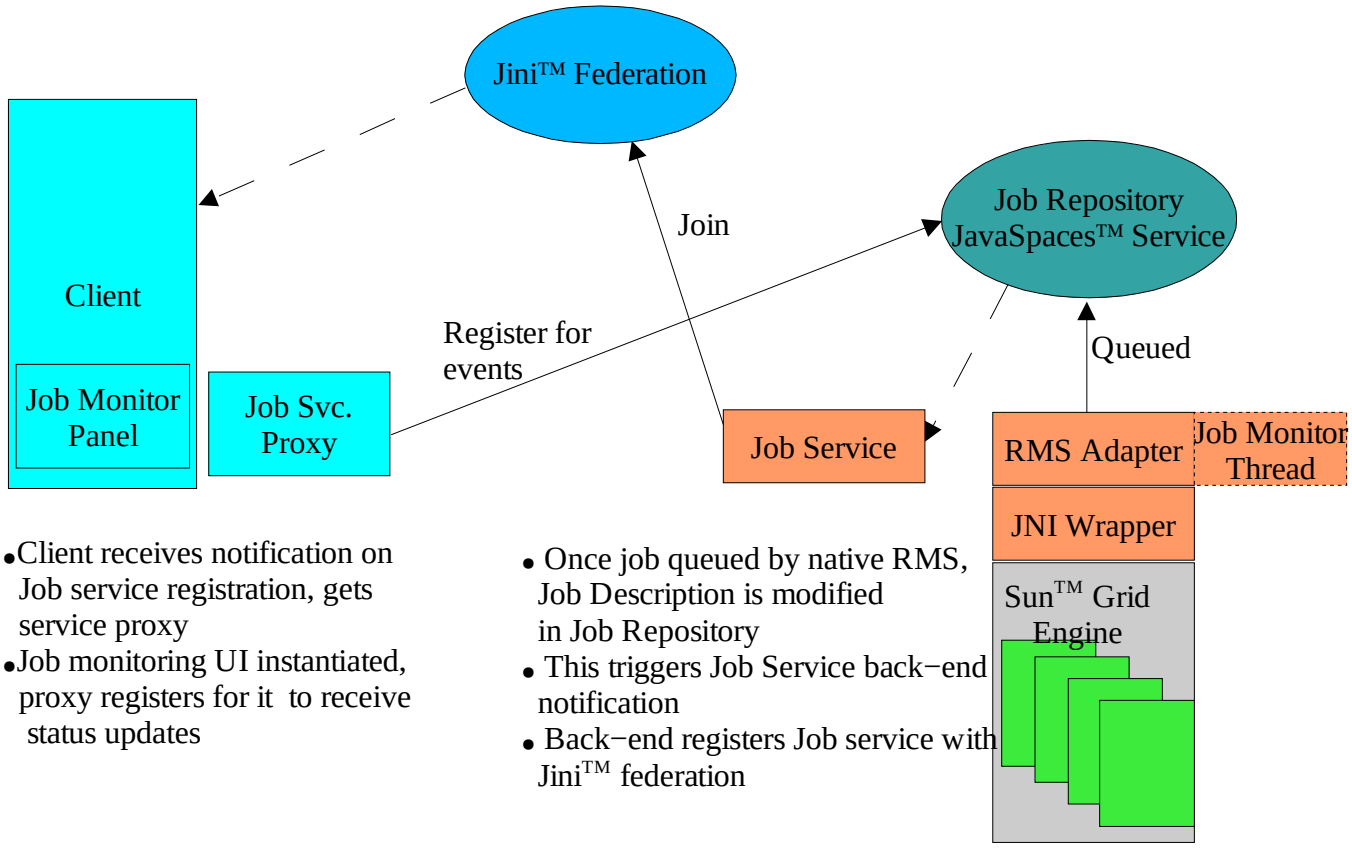

Figure 4: Job Service Creation

As mentioned in the discussion on queues, a job is submitted for execution by the queue proxy writing a new job description to the Job Repository. The RMS Adapter receives a notification when a new job description is written. The RMS Adapter submits the job to the native resource management system and changes the state of the job description to note that it has been queued. At this point, the job service back−end code, which had previously registered for notification of queued jobs appearing in the Job Repository, receives a notification and registers the job service, including a handle to the Job Repository, as a Jini™ service.

Job monitoring and controlling functions also utilize the Job Repository. Once the job service is registered, its service proxy is downloaded to the client (the client is discussed

further below) which had registered for notification of such job service registration when the application was submitted. The job service proxy reads the job description from the Job Repository in order to get job status information as well as a handle on the job's stdio (generally, a URL). The proxy modifies the job description in order to notify the queue service back end of requests to change the job's execution state (e.g. signal, restart). The RMS Adapter has registered to receive notifications of these job description updates, and responds by forwarding the request to the native resource management system. See Figure 5.

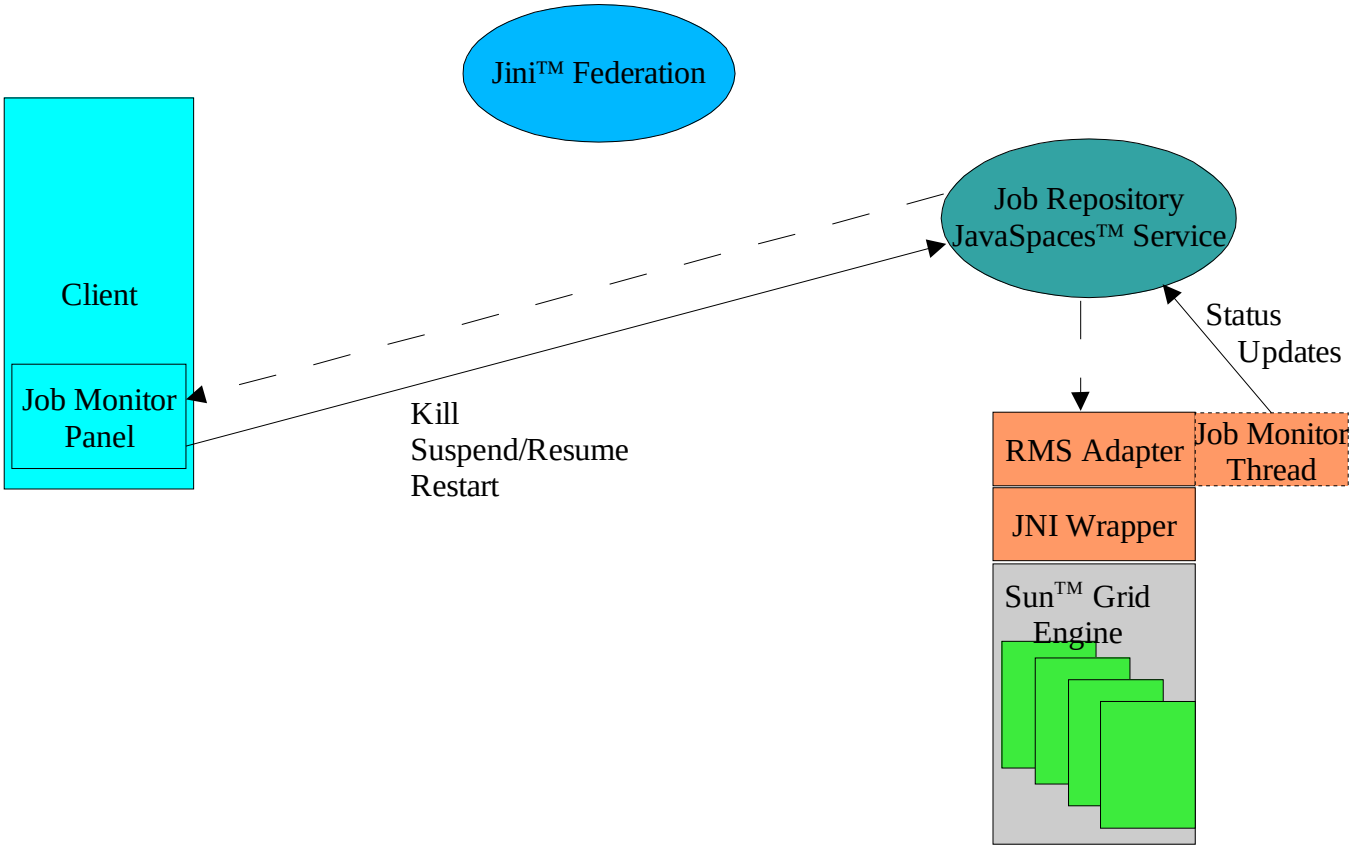

Figure 5: JAM Job Monitoring and Control

Application Registry

A list of registered applications needs to be constructed and then updated as new applications are added to JAM. The Application Registry serves as the persistent store for this information. The persistence mechanism can be implemented using simple files, or a database back−end, depending on the size of the registry. While it is tempting to consider utilizing a persistent JavaSpaces™ service, this is not a recommended usage pattern.

Administration

The JAM architecture supports two levels of administration: user and service. At the

user level, access control lists (ACLs) are used for controlling the following:

- Launching specific applications
- Submitting to specific queues
- Modifying the Application Registry (adding/changing/deleting applications)

At the service level, an administrator can monitor and control the overall operation of JAM, including bringing services up or down, and monitoring the state of various compute resources, running jobs, etc. Jini™ technology provides a set of interfaces to support the administration of services. The Jini™ services used by JAM all implement the appropriate administration interfaces, which provides the underpinnings of a service−level administration implementation.

Client Code

This section describes the different front−end user interface panels with which a JAM user interacts in the general order they will be encountered during a typical session. The Jini™ technology provides the ability to run the client code from anywhere a unicast or multicast connection to a Jini™ lookup service hosting JAM's services can be established.

Login

This is the first screen a user sees when invoking the JAM client. It is where the user is authenticated to JAM through a standard user name and password dialog. The goal is to provide a single sign−on model, where these initial credentials are securely saved and forwarded as needed throughout the user's session with JAM.

Application Selection

The client displays a list of all applications registered with the Jini lookup service. The user selects one for submission.

Parameter Input, Queue Selection, Job Submission

Once the user selects an application, both application and resource parameters are entered. The application proxy itself provides a panel for entering application−specific parameters, while the client provides both a container for the application−supplied panel, and its own panels for entering application−independent parameters and resource parameters. Figure 6 shows a frame with the application proxy supplied panel at the top, the resource parameter input panel in the middle, and the queue selection panel (along with a panel for status messages) at the bottom.

As resource parameters are entered or modified and as the state of various queue services change, a list of matching candidate queue services is displayed. This list is dynamically updated by the Application Agent.

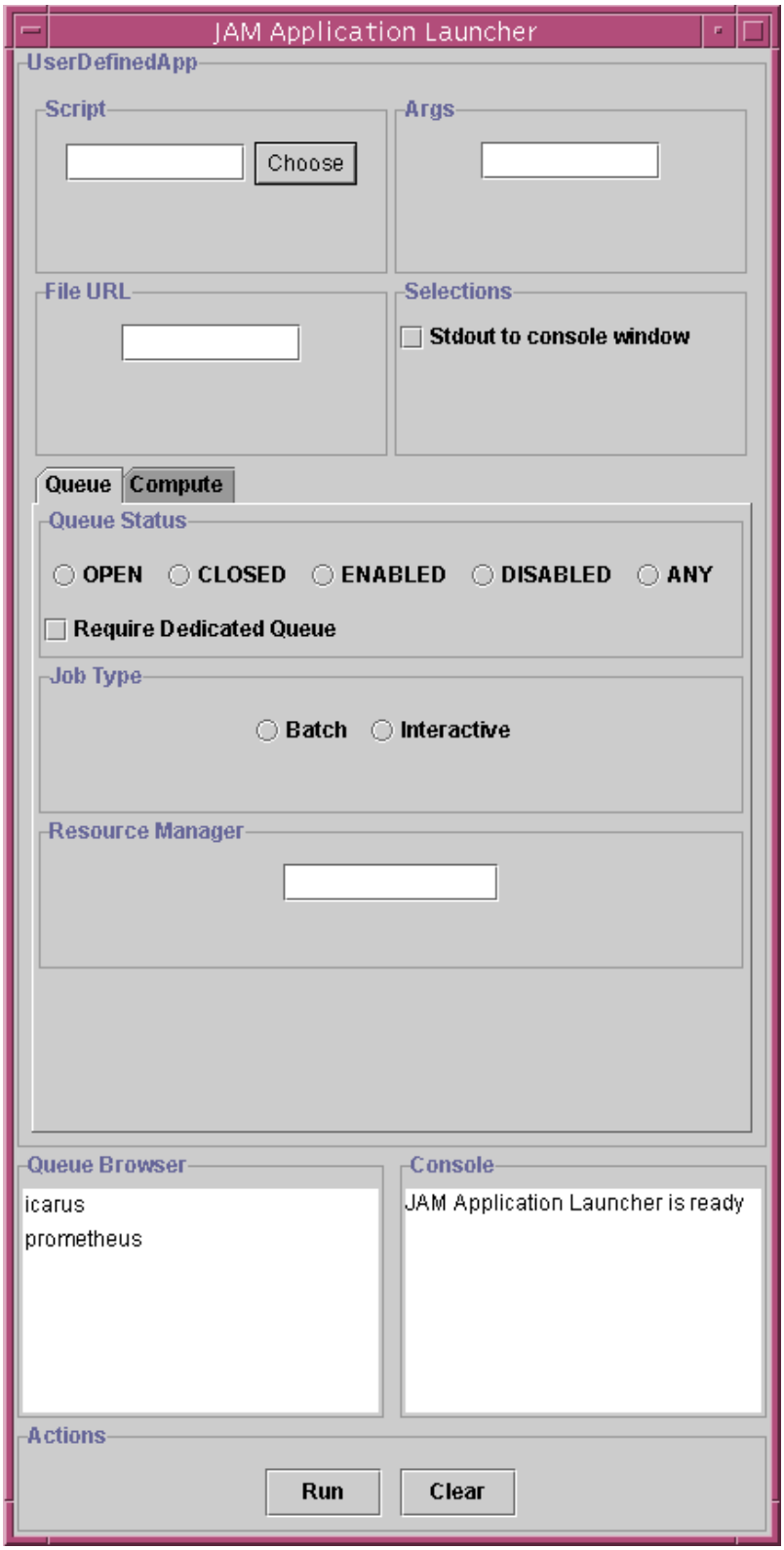

Figure 6: Parameter input, Queue selection window

Job Monitoring, Controlling, Steering

The application service proxy can optionally register for job monitoring to be performed by the client. If this is the case, the client will be notified when the job service is registered (which occurs after the job has been submitted to the native resource management system). The client then displays the current execution state of the job along with its stdout and stderr streams[2] with both continuously updating as the job runs. It also provides controls for signaling the job. This is shown in Figure 7.

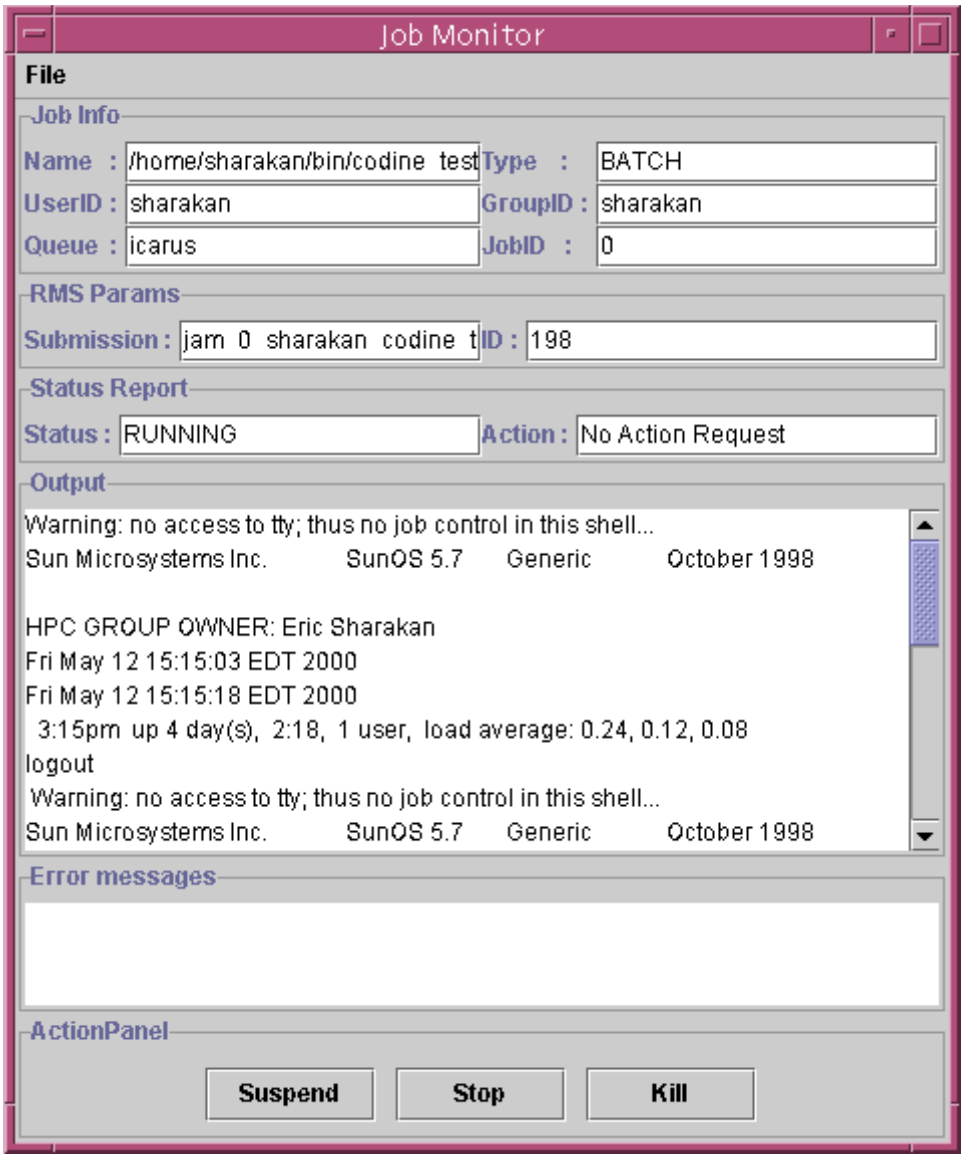

Figure 7: Job Monitor Frame

2 Not all resource management systems support streaming stdout and stderr back to the job submitter. In this case, JAM endeavors to get this information from files that such a resource manager might use to store this information. In this case, the output might not be streamed to the window; in fact, the output might not appear until the job has finished execution.

In addition, the user is optionally able to access application−specific steering hooks which are provided as part of the application service. The application steering architecture is still an area of active research.

Resource Monitoring

Resource monitoring is also available during queue selection, providing useful, continually updated information about the current state of the selected resource. For example, double−clicking on one of the queue service items as shown in Figure 6 will result in a window as shown below in Figure 8.

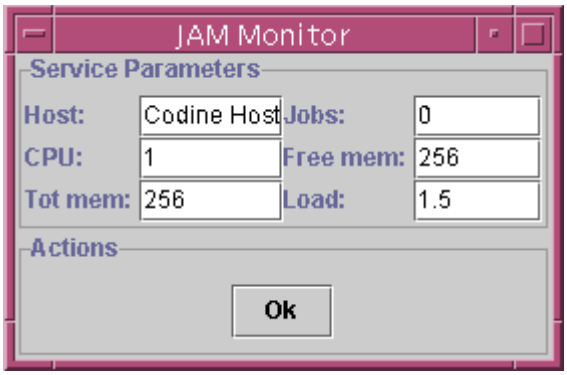

Figure 8: Resource Monitoring

Summary

JAM provides a unifying framework for selecting and launching jobs into one of a set of available resources. JAM's architecture leverages the distributed nature of Jini™ and JavaSpaces™ technologies to build a dynamic, distributed system. This allows us to concentrate on JAM's high−level architectural issues, as all the underlying distributed infrastructure is provided by Jini™ and JavaSpaces™ technologies.