GrADSoft and its Application Manager: An Execution Mechanism for Grid Applications

Authors

Ken Kennedy, Mark Mazina, John Mellor-Crummey, Rice University

Ruth Aydt, Celso Mendes, UIUC

Holly Dail, Otto Sievert, UCSD

October 04, 2001 Revision of original dated **August 10, 2001**

Earlier source documents

- August 6, 2000 **Compiler/Resource Selector Interaction Scenario** by Dail, Sievert, & Obertelli.
- February 24, 2001 **GrADSoft — A Program-level approach to using the Grid** by Mazina, Sievert, Dail, Obertelli, & Mellor-Crummey.

Introduction

Initial efforts within the GrADS project have demonstrated the complexity of writing applications for the Grid and managing their execution. To deal with this complexity, the GrADS project has adopted a strategy for program preparation and execution that revolves around the idea that a program must be *configurable* to run on the Grid. To be configurable in the sense intended by GrADS, a program must contain more than just code—it must also include a portable strategy for mapping the program onto distributed computing resources and a mechanism to evaluate how well that mapped program will run on a given set of resources. Our current goal in selecting resources is to minimize runtime, but our design allows for other factors, such as cost of resources, to be considered as well.

Once a configurable object program, plus input data, is provided to the GrADS execution system, there must be a process that initiates the resource selection, launches the problem run, and sees it through to completion of execution. In the GrADS execution framework the *Application Manager* is the process that is responsible for these activities – either directly or through the invocation of other GrADS components or services. In this scenario, the individual GrADS components only need to know *how* to accomplish their task(s); the question of *when* and with *what* input or state becomes the Application Manager's responsibility.

The goal of this document is to define the role of the Application Manager in initiating, monitoring, and finalizing execution of a GrADS application.

Definitions

- **Application** — Code implementing one or more algorithms to solve a user's problem of interest. An application will often be composed of multiple *sub-applications*. At the highest level, the application is typically called a *Program*. In a non-Grid environment, *sub-applications* are typically *subroutine calls*.

- **Problem Run** — An application plus a set of input data and other parameters such as the problem size plus any other run-specific parameters specified or desired by the user. The problem run is the computation the user wants executed on the Grid.

- **Resource** — A physical device that can be used to perform work. This device may be reserved, but is most commonly shared. Examples: Cray T90, cluster of Pentium III workstations with a 100Mbit Ethernet interconnect.

- **Virtual Machine (VM)** — An actual collection of resources selected specifically for a particular problem run, and the topological relationship(s) among these resources. For example, the virtual machine for a master/worker problem run might be the physical machines `dralion`, `soleil`, `magie`, and `mystere`, where `soleil` is the master and the others are workers. Assuming the workers do not have to communicate among themselves, the virtual machine would describe a star topology with `soleil` at the center.

- **Application Manager** — A process that coordinates all the other pieces of GrADS and provides easy-to-use access to the Grid for scientific computing. The Application Manager starts up prior to the actual launch of the problem run on the Grid resources and exists for its' duration.

- **PPS** — Program Preparation System; refer to The GrADS Project: Software Support for High-Level Grid Application Development, July 31, 2001. The Program Preparation System can be split into two phases, the ***Building*** phase and the ***Binding*** phase. The former refers to activities done before the virtual machine is developed; the latter, those activities that occur after the virtual machine to be used for the actual computation has been chosen. Post-mortem analysis of a problem run is considered part of the (next) Building phase.

- **Configurable Object Program (COP)** — Created in the PPS Building phase, the COP contains an AART Model, IR Code, a Mapper, and a Resource Selection Evaluator, which are defined below. .

- **Application Abstract Resource and Topology (AART) Model** — An AART Model provides a structured method for encapsulation of application characteristics and requirements in an input-data-independent way. This consists of a collection of descriptive and parametric resource characteristics plus a description of the topology connecting these resources. The purpose of the AART Model is to kick-start the resource selection process and to provide part of the information needed by the Mapper and the Resource Selection Evaluator. Created in the PPS Building phase.

- **Intermediate Representation (IR) Code** — The GrADS version of a binary before the virtual machine is selected. The compiler has done as much as it can do until the virtual machine composition is known, at which point final compile and link can happen. The user should not have to start from untouched code each time he/she varies the data in a new run of an application, although the virtual machine is likely going to vary run-to-run. The term IR Code has traditionally been used in compiler work to refer to a short-lived transformation of code as it moves through the compiler's stages; GrADS IR Code has persistence. Created in the PPS Building phase.

- **Mapper** — A GrADS component that determines data and computation layout on an arbitrary virtual machine. This eventually includes the location both of specific data and of specific computational efforts. Created in the PPS Building phase.

- **Resource Selection Evaluator (RSE)** — A GrADS component that uses one or more parametric resource selection models to produce a prediction of a single, possibly composite, metric that can be used to select resources for the problem run. Created in the PPS Building phase.

- **Dynamic Optimizer** — Refer to The GrADS Project: Software Support for High-Level Grid Application Development, July 31, 2001. Part of the PPS Binding phase.

- **Performance Contract** — A statement of the expected performance of the application on the resources allocated to it. The contract may contain a contract monitoring performance model that has been fully developed prior to application launch, or it may contain a model that evolves based on runtime observations. Refer to Specifying and Monitoring GrADS Contracts, Second Draft.

- **Performance Monitoring Setup Module** — A GrADS component that inserts sensors into the final executables so that performance can be monitored. This module also assembles the performance contract, initial violation thresholds, and contract evaluation method(s) that will be used by the Contract Monitor. Part of the PPS Binding phase.

- **PES** — Program Execution System; refer to The GrADS Project: Software Support for High-Level Grid Application Development, July 31, 2001.

- **Resource Selection Seed Model (RSSM)** — An Application Manager creation: the AART Model combined with the user's problem run input.

- **Scheduler/Resource Negotiator (S/RN)** — The S/RN chooses grid resources appropriate for a particular problem run based on that run's characteristics and organizes them into a proposed virtual machine. In GrADS, the S/RN is basically an optimization procedure that searches the space of acceptable resources looking for the best fit with the application's needs as determined by an objective function contained in the Resource Selection Evaluator.

- **Rescheduler/Resource Renegotiator (R/RR) —** The R/RR chooses grid resources to maintain application performance/progress, after execution has begun. It performs the

same tasks as the S/RN, but additionally accounts for the specifics of the existing execution environment, i.e., the VM currently in use. Grid resource selection is guided by heuristics indicating the cause(s) of poor performance. The Rescheduler is most typically invoked when application performance falters.

- **Resource Reselection Evaluator (RRE) —** A GrADS component that uses one or more parametric resource reselection models to produce a prediction of a single, possibly composite, metric that can be used to select resources for the *remaining portion* of the problem run. Created in the PPS Building phase.

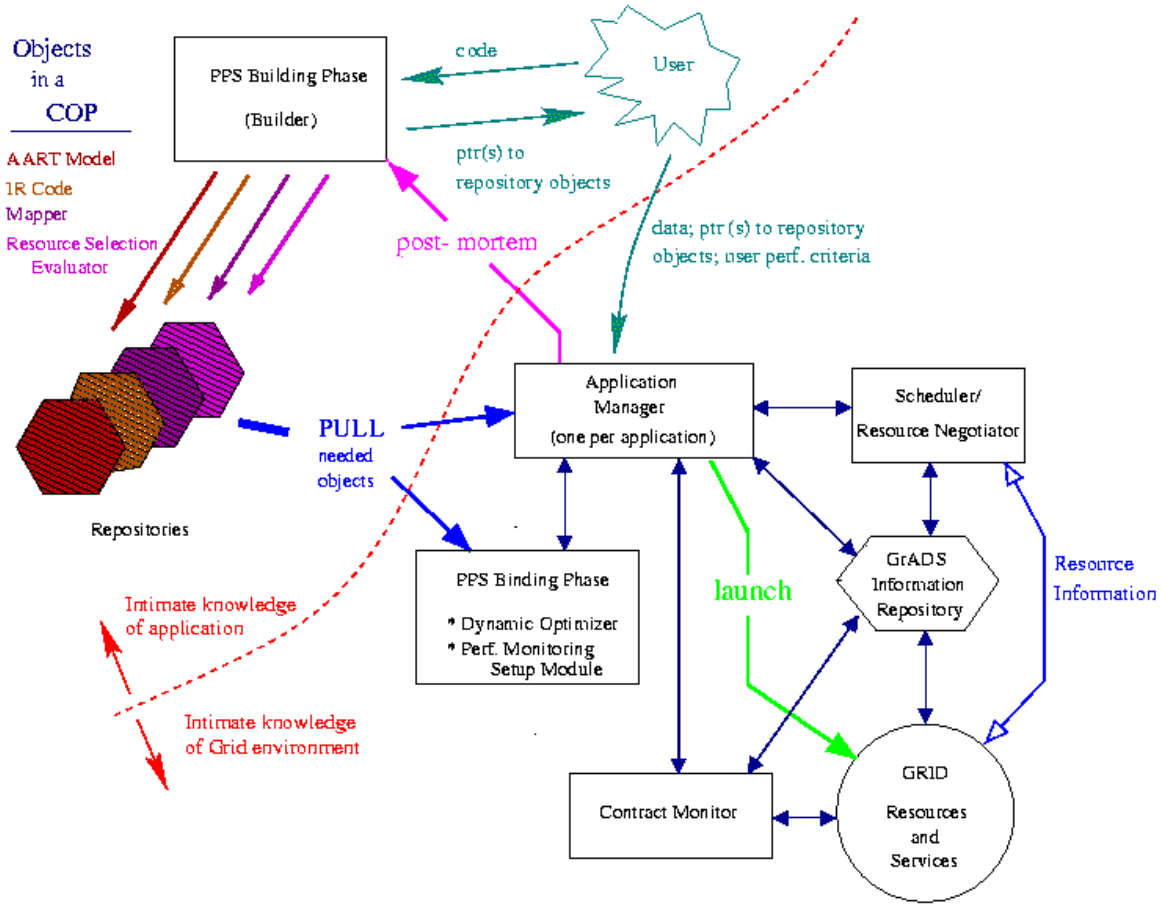
- **Contract Monitor** —A GrADS component consisting of one or more processes, which are possibly distributed across many Grid resources, that receives information from sensors embedded in a running Grid application. It may also receive information from execution environment sensors.

  The Contract Monitor uses the performance contract, violation thresholds, and contract evaluation method(s) from the Performance Monitoring Setup Module to evaluate the reported sensor values and determine if the application is delivering an acceptable level of performance. Optionally, the Contract Monitor Component may try to identify the cause of poor performance. The Contract Monitor makes its findings available to the Application Manager, which may make further decisions based on the information it receives.

  In addition to the real-time processing of the sensor and Contract Monitor information during application execution, the information can be collected and archived for later use by the Program Preparation System or other GrADS components.

- **GrADS PES Repository** — This repository holds information on executing GrADS problem runs. This state includes what resources are in the virtual machine plus other information necessary to restart a failed Application Manager.

- **Launcher** — A specialized GrADS interface to the Globus Toolkit's GRAM protocol that is responsible for starting the problem run on the selected virtual machine. Globus provides the actual Grid run-time tools at the virtual machine level, rather like a virtual OS. The launcher functionality may be encapsulated in the Application Manager.



Functional Relationship Diagram

General Scenario

Our User, or a Problem Solving Environment (PSE) on behalf of the user, provides the Builder with source code (which may be annotated with resource selection or run-time behavior information) or a handle to an existing IR Code object previously created for the user. The remainder of this scenario will not try to distinguish between the human user and any PSE working on behalf of that human user.

The Builder constructs any required objects and returns a handle to a Configurable Object Program. The COP includes the IR Code, AART Model, Mapper, and Resource Selection Evaluator. Note that at this point, the Mapper and Resource Selection Evaluator have too little information do anything useful. The IR Code will include library stubs needed by statements in the user's source code.

The User starts the Application Manager. This may be a standard GrADS Application Manager or a user designed one. The Application Manager needs the handle to the COP, I/O location information, the problem run size information (specifically, information to allow calculation of memory requirements), plus any desired resource selection criteria and other run-specific parameters desired or required.

The Application Manager retrieves the pieces of the COP. The AART Model is combined with the problem run information, resulting in the Resource Selection Seed Model. This produces the preliminary state necessary for the Mapper and the Resource Selection Evaluator to start being useful.

The Scheduler/Resource Negotiator uses the Resource Selection Seed Model, information about the state of Grid resources from the GIS, and the services of the Mapper and the Resource Selection Evaluator to develop a proposed virtual machine. (See **Resource Selection Scenario** below for more details on this step; jump-starting the resource selection process efficiently is a significant research topic.)

The Application Manager stores state, basically a checkpoint, on the impending problem run (i.e. application + data) in the GrADS PES Repository. The Application Manager then calls the PPS Binding Phase, passing it the COP handle, selected virtual machine, and the user's run-time information.

The PPS Binding Phase uses the Mapper for actual data layout, and creates optimized binaries using the Dynamic Optimizer. It also inserts monitoring sensors based on information from the Performance Monitoring Setup Module. For some Grid-aware libraries, the PPS Binding Phase may need to arrange for dynamic linking to pre-built libraries for specific platforms. Handles to the optimized problem run binaries are passed back to the Application Manager, which again checkpoints its state to the GrADS PES Repository.

The Application Manager starts the Contract Monitor and then launches the binaries by invoking

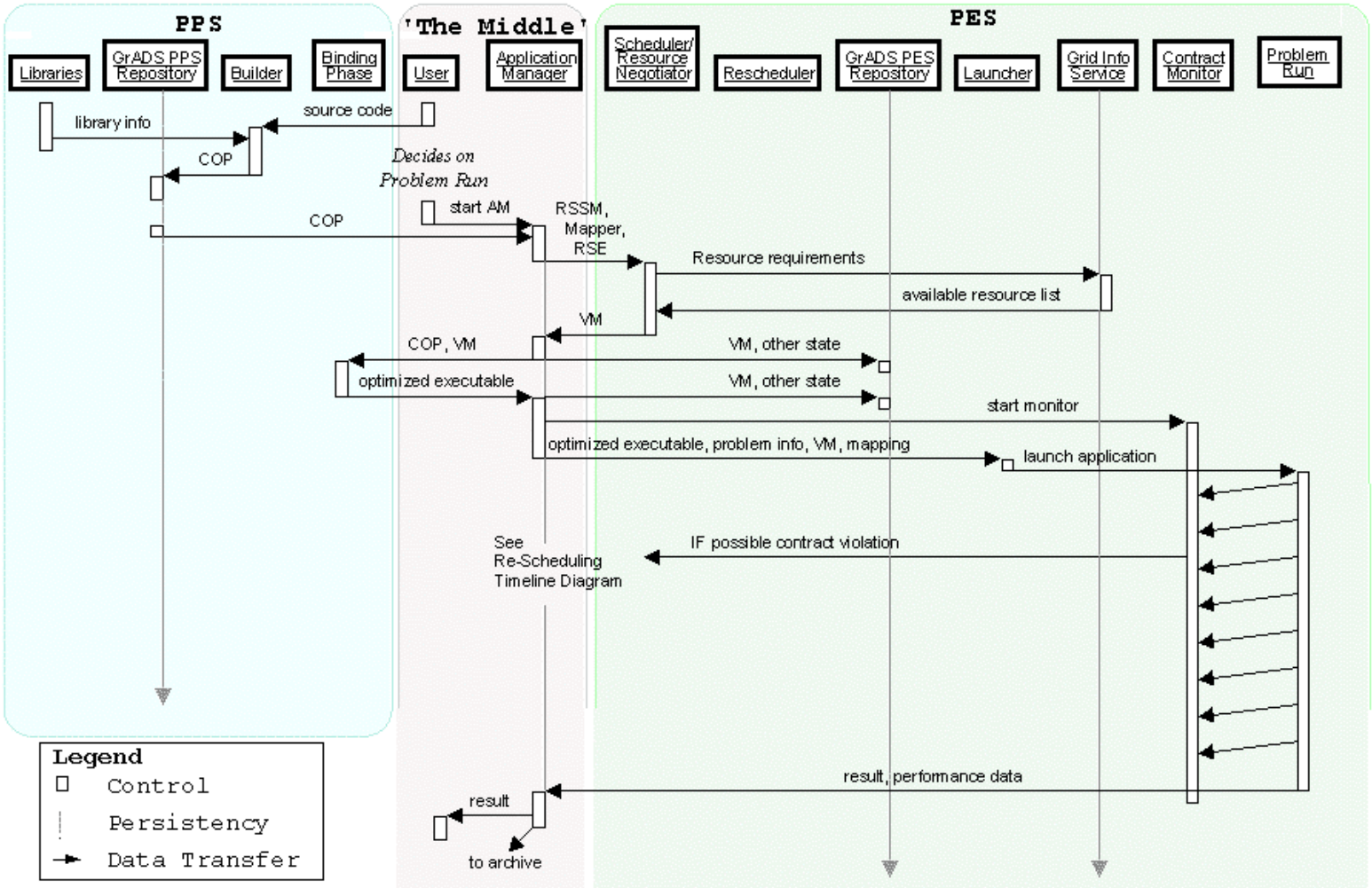
the GrADS Launcher. While the Contract Monitor is initializing, PPS inserted code in the binaries may be positioning data on the resources making up the virtual machine.

As the code runs, the Contract Monitor gathers sensor data and uses the performance contract, violation thresholds, and contract evaluation method(s) from the Performance Monitoring Setup Module to determine if the application is delivering an acceptable level of performance based on the models and thresholds provided. In addition, the Contract Monitor may try to make some determination of the cause of the poor performance. It reports its findings, together with summary monitoring information, to the Application Manager.

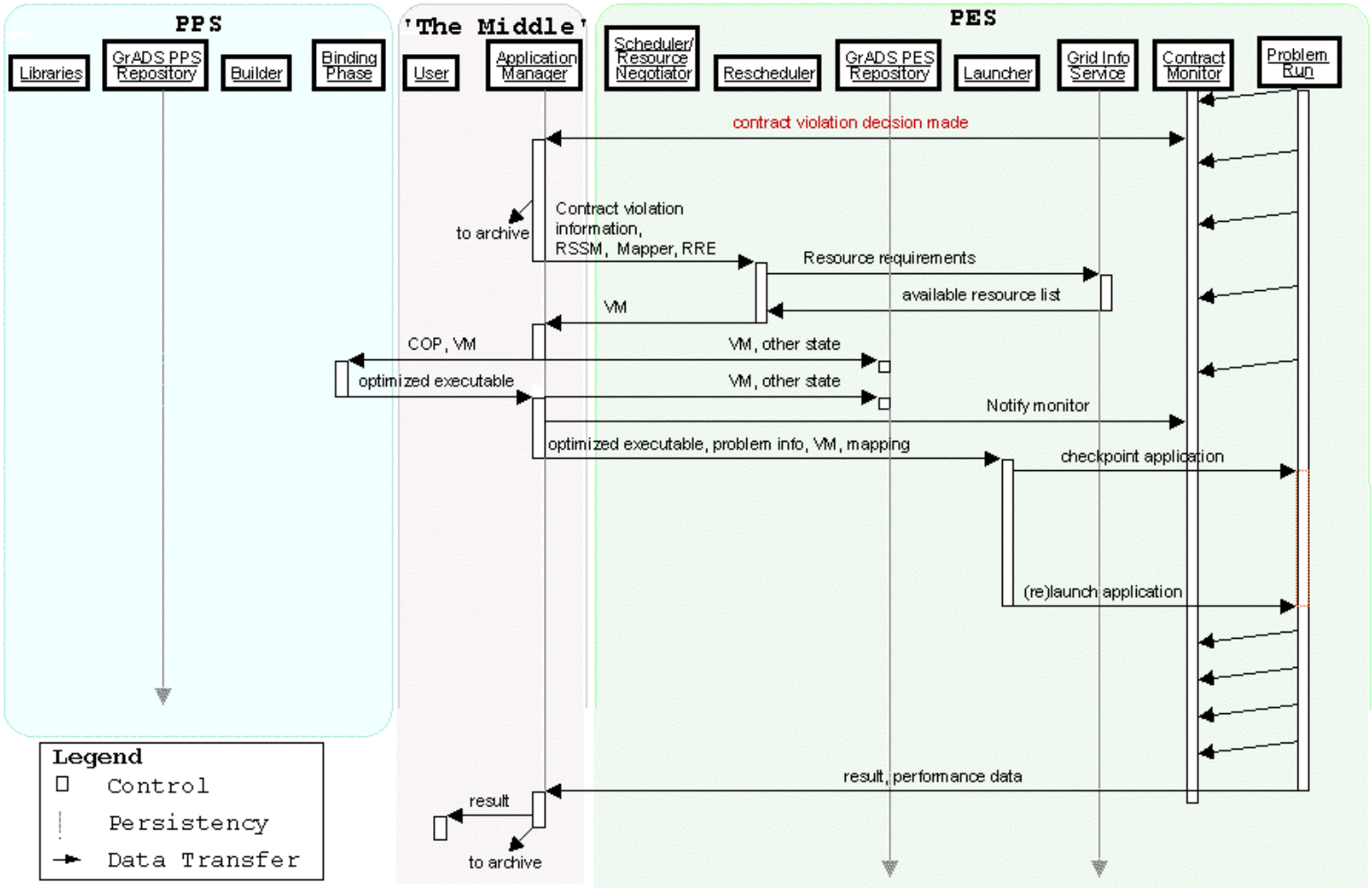
Concurrently, the Contract Monitor output, as well as the original sensor output, can be archived for later use to refine models, adjust thresholds, or guide future executions. In addition, the application, Contract Monitor, and Application Manager may adjust the performance contract, violation thresholds, and contract evaluation method(s) throughout the application lifetime in response to evolving application patterns and resource volatility. If the Application Manager determines that the application is not making reasonable progress (or alternately, if the system becomes aware of more suitable execution resources), the Rescheduler is invoked. Using knowledge of the current execution, the Rescheduler determines the best course of action in order to improve progress. Examples of rescheduling actions are replacing particular resources, redistributing the application workload/tasks on the current resources, and adding or removing resources; or doing nothing (continuing execution with the current VM).

If the Rescheduler constructs a revised VM, the Application Manager builds new optimized executables, checkpoints the application, and reconfigures and re-launches the application. The application reads in the checkpoint information and continues program execution. See **Resource Reselection Scenario** below for more details.

Once the application finishes, the Application Manager makes certain that the relevant collected performance data is fed back (i.e. archived) into the Program Preparation System and shuts down the Contract Monitor.



General Scenario Timeline Diagram



Re-Scheduling Timeline Diagram

A Resource Selection Scenario

The S/RN receives the Resource Selection Seed Model, the Mapper, and the Resource Selection Evaluator from the Application Manager.

The S/RN uses the RSSM to determine what is required of resources (rather than desired).

The S/RN uses the Grid Information Service to get a list of resources matching the requirements. For example, the resulting 'available machine list' may be resources that the user has accounts on and that have access to a specialized device.

The Grid Information Service is then queried to obtain resource characteristics on this list of available machines (processor speed, memory available, bandwidth, etc), resulting in a ‘bag’ of resources that may be usable for this problem run.

The S/RN now creates a 'bunch' of possible VM(s), that is, structured sets of resources with a basic mapping, and compares/evaluates them using the RSE. In general, this involves the following steps:

- The Resource Selection Seed Model is used in conjunction with heuristics to create this 'bunch' of possible virtual machines that are likely to be desirable.

- The Mapper is used to provide a rough mapping of the problem run onto the resources. This mapping just gives an idea of how much work (computation, communication, etc) each processor will do -- it is not the same as a detailed mapping as done in the Binding Phase.

- The Resource Selection Evaluator is used to compare possible virtual machines, either by returning a comparable value (execution time prediction) or by providing a comparison operation for ranking possible VM(s).

This process results in one or more proposed virtual machines that are returned to the Application Manager. For simplicity, this document assumes one, but its easy to return the top ***n*** solutions if that turns out to be a useful approach.

A Resource Reselection (Re-scheduling) Scenario

The R/RR receives the current VM (including the grid resource information assumed when this VM was created), the Mapper, and the Resource Reselection Evaluator information from the Application Manager. It also receives contract violation indications and other contract information from the Application Manager and, possibly, directly from the Contract Monitor. In addition to raw or processed Contract violation information, the Contract Monitor information may contain hints indicating possible performance limiters. The R/RR uses this information, in addition to dynamic grid resource information, to produce better-performing VMs.

The Resource Reselection process performs the same general actions as the Resource Selection process, with the following differences:

- Resource Reselection is Resource Selection with initial conditions: the problem run is already executing on a particular VM, thus problem data and executables have an initial distribution and resource affinity.
- Resource Reselection is concerned only with the performance of the remaining portion of the problem run, and not the entire problem run.
- Under most circumstances, Resource Reselection must be a lightweight process to be effective; if Resource Reselection demands significant overhead, it is typically better to (a) simply "ride out" the rest of the problem run with the existing resources, or (b) completely restart the application from scratch, invoking the standard Resource Selection algorithm.
- Resource Reselection may have access to additional information (from the contract monitor) that assists in selecting appropriate resource sets.
- In the query of dynamic grid resource information from the Grid Information Service, Resource Reselection must be able to discount resource usage by the current problem run; we don't want to discard a good resource simply because it is loaded by the current problem run!

First, the Resource Reselection procedure determines the symptoms of poor performance. Using heuristics, it matches these symptoms with specific root causes, and determines one or more possible reactions that will promote performance. For example, if the flop rate of one processor is low and this processor's cpu utilization is high, then we may guess that this processor is heavily loaded, and needs to be exchanged. [Refer to Contract Renegotiation, January 13, 2001.]

Additional possible reactions are determined from the supplied Contract information: if resource fault hints are reported by the Contract Monitor, they are compared with current Grid Information Service readings and the historical GIS values used when initially creating the current VM. If the GIS comparison concurs with the Contract Monitor hints, these resources are selected for removal/replacement.

Once a set of possible reactions has been determined, the R/RN, like the S/RN, uses the Grid Information Service to get a list of resources matching the application's requirements. This information, properly accounting for the current problem run resource load, is used to create multiple VMs. The performance of each of these VMs for the remainder of the problem run,

combined with the overhead required to instantiate each VM (e.g., process migration, data migration, data redistribution), is ranked. A heuristic selection algorithm chooses the VMs that result in improved performance.

This process results in zero or more new virtual machines that are returned to the Application Manager.

Appendix A:

Functionality and Services Provided by Components (incomplete outline form)

- Application Manager
- PPS Building Phase
    - Configurable Object Program
    - AART Model --- Three fundamental features of an AART Model are
        - The AART Model characteristics describe application desired resources and topological organization independent of any given run-time data. For example, the AART Model contains the desired kind of resource topology (e.g., machines are arranged in a two-dimensional mesh) without mention of the exact size of such a mesh (which is highly data and resource dependent). Note that topology type is just a characteristic of the application.
        - The AART Model characteristics may change based on the problem size. So the model may be a continuum of models or a discrete number of very different models based on that size. Below some application-specific problem size, any AART Model will likely specify using a single compute resource at one location.
        - The AART Model attempts to describe resources and topologies necessary for *efficient computation*, although this version of this document does not attempt to define *efficient*.

      The topology-type characteristic will typically have a dimensionality (e.g. 3-D mesh) or a number of levels (e.g. trees). Then one may describe characteristics at both the total-program level and characteristics that apply to specific dimensions or levels of the topology. To support more general application topologies, the AART Model actually supports resource sets, with the dimension and level abstractions being specialized views of sets. The model class can, in general, address three types of constraints:

        - application-wide (e.g. all sets) constraints
        - constraints specific to resources in set *i*
        - pair-wise, directional constraints from set *i* to set *j*

      The program-level AART Model must be able to compose multiple AART Model objects while respecting constraints in the individual models. (A major open research question is "how?").

    - IR Code --- The program-level IR Code object will consist of IR Code objects describing the various pieces of the application: the user code, PSE/numeric libraries, performance sensor/monitoring libraries and any other code necessary for the application to run. For PSE/numeric libraries that have already been GrADized and the GrADS performance sensor/monitoring libraries, these Intermediate Code objects should already exist either as self-contained objects or as stubs that provide the Binding Phase information about which library binaries to include in the final executable. Any high-level optimization efforts during the Binding Phase will use this intermediate representation.
    - Mapper
    - Resource Selection Evaluator
    - GrADS PPS Repositories
- PPS Binding Phase
    - Dynamic Optimizer
    - Performance Monitoring Setup Module
- PES
    - Scheduler/Resource Negotiator
    - Contract Monitor
    - Rescheduler/Resource Renegotiator
    - GrADS PES Repository
    - Launcher