Update to "The Virtual Grid Description Language: vgDL"

Version 0.96

Andrew A. Chien, Henri Casanova, Yang-suk Kee, Richard Huang, Dionysis Logothetis, and Ken Yocum
vgrads@cs.ucsd.edu

March 16, 2005

"The Virtual Grid Description Language: vgDL" document, dated August 9, 2004, described the design rationale and initial design for the vgDL language. In the intervening period, with extensive construction of use cases and the building of the first vgES implementation, there have been numerous minor changes and refinements to the language. So, while "The Virtual Grid Description Language: vgDL" remains an excellent exposition of the motivations, ideas, and goals behind vgDL, this update serves as a reference for the current language definition and vgDL selection attributes.

1. vgDL EBNF Grammar

In this section, we define vgDL syntax by showing its grammar in EBNF.

- Terminals are shown "**quoted, bold**", and

We assume that the non-terminals *String, Integer, Float, and Time* denote respectively a string of alphanumerical characters, an integer number, a floating-point number, and Time/date in traditional formats. Please refer to examples and the release notes for the supporte formats.

```
Vgrid := VgDefineExpr ["at" Time ]
VgDefineExpr := Identifier "=" VgExpr
Identifier := String
VgExpr := VgSubExpr | VgDefineExpr ("close" | "far" | "highBW" | "lowBW") VgDefineExpr
VgSubExpr := VgAssociatorExpr | VgNodeExpr | "{" VgExpr "}"
VgAssociatorExpr := VgBagExpr | VgClusterExpr
VgBagExpr := ("LooseBagof" | "TightBagof")   "(" Identifier ")" "[" MinNode ":" MaxNode "]"
                    [ "[" RedlineExpr "]" ] [" [" Rank "=" ArithmeticExpr "]" ] "{" VgDefineExpr"}"
MinNode := Integer
MaxNode := Integer
Number := Integer
VgClusterExpr := "Clusterof" "(" Identifier ")" "[" MinNode ":" MaxNode "]" [ "[" RedlineExpr "]" ]
                    [" [" Rank "=" ArithmeticExpr "]" ] "{" VgDefineExpr "}"
MinTime := Integer
MaxTime := Integer
VgNodeExpr := "[" RedlineExpr "]" [" [" Rank "=" ArithmeticExpr "]" ]

RedlineExpr := CondAndExpr [ "||" CondAndExpr ]* [ "," Predicate ]
CondAndExpr := EqualExpr [ "&&" EqualExpr ]*
EqualExpr := RelationalExpr [ ("==" | "!=") RelationalExpr ]*
RelationalExpr := AddExpr [ (">=" | "<=" | ">" | "<") AddExpr ]*
AddExpr := MultExpr [ ("+"|"-") MultExpr ]*
```

```
MultExpr := UnaryExpr [ ("*" | "/")    UnaryExpr ]*
UnaryExpr := Integer | Float | Attribute | "(" RedlineExpr ")" |
                    ((“Cluster” | “TightBag” | LooseBag”) "." Attribute)
Predicate := "Required" "(" Attribute ["," Attribute ]* ")"
Attribute := String
ArithmeticExpr := ArithMultExpr [ ("+"|"-") ArithMultExpr ]*
ArithMultExpr := ArithUnaryExpr [ ("*" | "/")    ArithUnaryExpr ]*
ArithUnaryExpr := Integer | Float | Attribute | "(" ArithmeticExpr ")" |
                    ((“Cluster” | “TightBag” | LooseBag”) "." Attribute)
```

The interested reader is directed to numerous examples of vgDL in the other papers as well as the appendix of this document for vgDL examples.

2. Resource Attributes for Selection: vgFAB and vgDL

This section lists the common, standardized set of resource attributes one can use to describe a virtual grid in vgDL. These resource attributes, such as CPU or OS type, allow the vgFAB to select resources. Note that vgDL is extensible and supports the addition of arbitrary attributes. However, to incorporate new attributes for selection, they must be produced by the underlying information provider (e.g., MDS, NWS, or Ganglia), stored in the vgFAB database, and incorporated into the vgFAB selection algorithm. A future design goal for vgFAB is to transparently support arbitrary selection attributes.

Here we describe the set of resource attributes currently used by the vgFAB 0.7 implementation (3/15/2005 release) for selection. For each attribute we give its name, its type, its meaning in the current vgES implementation, and an example value. Note that while there is some overlap, these vgDL and vgFAB resource selection attributes are distinct from the VG resource attributes described in the “Virtual Grid Resource Attributes” document.

Individual Resource Attributes:

Attribute Name	Type	Description	Example
Processor	String	CPU model name.	Pentium 4
OS	String	OS or kernel version name in vendor-specific convention.	Linux
Clock	Integer	Clock speed of a CPU. (MHz)	2000
Cache	Integer	Second-level unified cache size of a CPU. (KB)	256
CPUs	Integer	Total number of CPUs.	2
Memory	Integer	Total Memory Size. (MB)	512
Disk	Integer	Total Disk Size. (MB)	80000

Cluster Attributes:

Attribute Name	Type	Description	Example value

Processor	String	Processor Model	Pentium
CPUs	Integer	Number of CPUs	64
Nodes	Integer	Number of Hosts	32
Memory	Integer	Total Memory Size MB	32768
Disk	Integer	Total Disk Size MB	5120000

TightBag and Loose Bag attributes:

Attribute Name	Type	Description	Example value
CPUs	Integer	Number of CPUs in a Host	2
Nodes	Integer	Number of Hosts	128
Memory	Integer	Total Memory Size MB	40000
Avail_Memory	Integer	Available Memory Size MB	20000
Disk	Integer	Total Disk Size MB	10000000
Avail_Disk	Integer	Available Disk Size MB	3000000

3. vgDL Examples

```
Node = [(Memory > 128)&&(Clock > 1.0)][Rank = Memory+Clock]

VeryEasyOddRank1 = [(Memory > 128)&&(Clock>1.0)] [Rank = Memory-CPU]

VeryEasyOddRank2 = [(Memory>128)&&(Clock > 1.0)] [Rank = Memory-Clock]

VeryEasyOddRank3 = [(Memory > 128)&&(Clock > 1)] [Rank = Memory*CPU]

VeryEasyOddRank4 = [(Memory > 128)&&(Clock > 1)] [Rank = Memory/Clock]

VeryEasyOddRank5 = [(Memory > 128)&&(Clock > 1)] [Rank = Memory/(Clock-Clock)]


BigBag = LooseBagOf(n)[1:8193]
{
        n = [(Memory > 128)&&(Clock > 1)]
}

BiggerBag = LooseBagOf(n)[1:65537]
{
        n = [(Memory > 128)&&(Clock > 1)]
}

BigTightBag = TightBagOf(N)[1:1000000]
{
        N = [(Memory > 128)&&(Clock > 1)]
}

BigTightBagMemory = TightBagOf(N)[1:1000000] [Rank = TightBag.Memory]
{
        N = [(Memory >= 1024)&&(this != mybeautifulhouse)]
}

BigTightBagCPU = TightBagOf(N)[1:1000000] [Rank = TightBag.Nodes]
{
        N = [Clock >= 1]
}

BigTightBagOdd = TightBagOf(N)[1:1000000]
{
```

```
        N = [mynextdoorneighbor != cheezwhiz] [Rank = Clock-Memory]
}

BigClusterMemory = ClusterOf(N)[1:1000000] [Rank = Cluster.Memory]
{
        N = [(Memory >= 1024)&&(vgrid > gridalone)]
}

BigClusterCPU = ClusterOf(N)[1:1000000] [Rank = Cluster.Nodes]
{
        N = [Clock >= 1]
}

BigClusterOdd = ClusterOf(N)[1:1000000] [Rank = Nodes*Memory]
{
        N = [myfather != fredmcmurtz]
}

BigClusterOfCluster = ClusterOf(N)[1:1000000] [Rank = Cluster.Nodes]
{
        N = ClusterOf(M)[8:32]
        {
                M = [(Memory >= 1024)&&(vgrid > gridalone)] [Rank = Clock]
        }
}

BigClusterOfClusters = ClusterOf(N)[1:1000000] [Rank = Cluster.Nodes]
{
        N = ClusterOf(M)[8:32]
        {
                M = [(Memory >= 1024)&&(vgrid > gridalone)] [Rank = Clock]
        }
}

BigBagOfClusters = LooseBagOf(N)[1:1000000] [Rank = LooseBag.Nodes]
{
        N = ClusterOf(M)[8:32]
        {
                M = [(Memory >= 1024)&&(vgrid > gridalone)][Rank = Clock]
        }
}

BigBagOfClusters2 = LooseBagOf(N)[1:1000000] [Rank = LooseBag.Memory]
{
        N = ClusterOf(M)[8:32] [Rank = Cluster.Memory]
        {
                M = [(Memory >= 1024)&&(vgrid > gridalone)]
        }
}

BigBagOfClusters3 = LooseBagOf(N)[1:1000000] [Rank = LooseBag.Memory + LooseBag.Nodes]
{
        N = ClusterOf(M)[8:32]
        {
                M = [(Memory >= 1024)&&(vgrid > gridalone)] [Rank = Clock]
        }
}

BigBagOfTBags = LooseBagOf(N)[1:1000000] [Rank = LooseBag.Nodes]
{
        N = TightBagOf(M)[8:32]
        {
                M = [(Memory >= 1024)&&(vgrid > gridalone)]       [Rank = Clock]
        }
}

BigBagOfTBags2 = LooseBagOf(N)[1:1000000] [Rank = LooseBag.Memory]
```

```
{
        N = TightBagOf(M)[8:32] [Rank = TightBag.Memory]
        {
                M = [(Memory >= 1024)&&(vgrid > gridalone)]
        }
}

BigBagOfTBags3 = LooseBagOf(N)[1:1000000][Rank = LooseBag.Memory + LooseBag.Nodes]
{
        N = TightBagOf(M)[8:32]
        {
                M = [(Memory >= 1024)&&(vgrid > gridalone)] [Rank = Clock]
        }
}

LBofTBofLB = LooseBagOf(N)[2:16][Rank = LooseBag.Memory]
{
        N = TightBagOf(M)[8:32][Rank = TightBag.Nodes]
        {
                M = LooseBagOf(P)[4:16]
                {
                        P = [(Memory >= 1024)&&(vgrid > gridalone)][Rank = Clock]
                }
        }
}

LBofTBofLB = LooseBagOf(N)[2:16][Rank = LooseBag.Memory]
{
        N =  TightBagOf(M)[8:32][Rank = TightBag.Nodes]
        {
                M = LooseBagOf(P)[4:16][Rank = LooseBag.Memory]
                {
                        P = TightBagOf(Q)[2:8]
                        {
                                Q = [(Memory >= 1024)&&(vgrid > gridalone)][Rank = Clock]
                        }
                }
        }

}
```

The Virtual Grid Description Language: vgDL

Version 0.95

Andrew Chien, Henri Casanova, Yang-suk Kee, and Richard Huang
vgrads@cs.ucsd.edu

August 9, 2004

I. Motivation and Objectives

While the success and acceptance of Grids continues in the scientific and commercial computing communities, the technology required to achieve the grid vision -- flexible, adaptive computing on demand for a wide range of applications -- is still in its infancy. In fact, the applications that are easily portable into the grid environment while numerous, are mostly limited to a few paradigms (loosely-coupled parallel, asynchronous workflow, and multi-tier web-appserver-database applications). Applications that require more tightly-coupled coordination, high performance data movement coordinated with computation, and real-time coupling of instruments remain difficult to design, implement, and manage to run well. Broadening the class of application types viable on grids is the focus of the VGrADS effort, and the motivation behind our discussion here on virtual grids.

We are interested in the following questions and metrics and how their consideration affects the approachability and utility of the grid for a broad class of applications. An important issue is resource scale -- grid infrastructures are only in early stages today and will grow in the future to consist of millions, even billions of devices. Such growth will stretch our capabilities to scale grid mechanisms, services, and properties to the limit.

- What application development effort is required to run on the grid in a way that is functional, robust, adaptive, and efficient?
- What application semantics and structure must be exploited to make the above possible?
- What assumptions or knowledge of the resource environment should be embedded in the design of distributed grid application?
- What resource models are sufficient to enable expression by the programmer or automated optimization of the critical performance factors?
- What technologies and techniques are needed for initial resource selection and subsequent resource adaptation?
- What information is needed to support these techniques (static resource attributes, long-term resource characterization, and short term dynamic resource information)?
- How can we scale the virtual grid abstraction to future resource environments of billions of devices with radical heterogeneity?
- How can the desire of applications to choose optimal resource sets for performance be reconciled with shared resource environments (unavailability) and vast future grid environments (scalability)?
- What are the error and failure models that applications can expect to be universally implemented?

To explore these questions, we are developing a set of abstractions that enable expression of critical application structure, yet describe the resource needs simply, enabling a multitude of possible solutions, scale to millions of resources, and achieve quantifiably good solutions. We term these abstractions *Virtual Grids*. Our methodology is to use application case studies to both motivate and evaluate our designs, ensuring that the resulting simple abstractions meet the real needs of real applications. This document includes the following elements summarizing our current design status:

- an overview of the virtual grid abstraction and design,
- a design of the virtual grid resource specification language, vgDL, and
- a set of application examples using progressively more complex descriptions in the virtual grid resource specification language, complete with a rationale for why the application uses particular features in each case, and

II. Design Assumptions

Our fundamental assumptions are that Grid platforms are dynamic, shared resource environments of massive scale. This implies a range of limitations.

First, applications cannot individually scan or select from all of the resources, as the number is too vast. As a foundation, we presume a scalable resource information service. Some mechanism for focusing the search, selection, and binding activity is required; initial concepts we are using to address this are virtual grids and resource classes.

Second, resources are shared, widely distributed, and heterogeneous, so they may be unavailable for a wealth of reasons (other use, network disconnection, machine failure, network failure, network attack, etc.).

Third, scaling and robustness for high load factors in the grid are desirable; both for good resource behavior and for reasonable turnaround times for applications.

Fourth, we believe a separation of the application and resource management system is desirable, allowing the sharing of effort in the resource management system across multiple applications. Further, this allows applications to be designed, optimized, and evolved independent of idiosyncrasies of the current resource environment, current resource management policies, or other details.

III. Virtual Grids

Virtual grids are described by a *virtual grid resource specification* that is presented by the application to acquire resources for execution. A virtual grid resource specification captures the desired resources for an application, and its explicit resource structure can be used by the application designer to express parallelism, communication, and other forms of optimization. Virtual grid resource specifications are designed to be simple and flexible, maximizing the opportunity to match resources. To enable applications to guide detailed choice amongst resources that can make large factors performance difference,

applications provide a "ranking function" which is used by the execution system to select amongst identified candidates.

III.A. Obtaining a Virtual Grid

Applications present *virtual grid specifications* to the *Virtual Grid Execution System* (vgES). The execution system returns a resource collection that satisfies the request (or a partial solution with the unmatched part indicated) [1]

After the virtual grid specification is matched and instantiated with available physical resources, it is presented to the application as a resource description request, annotated with the list of resources, along with information sufficient to access the associated resource access mechanisms.

There are two ways in which the application can submit a virtual grid specification to the virtual grid system: *search and binding*, or *search only*. Binding of resources into a virtual grid means they are committed to the application (at the level they are specified). For example, a 100Mbit slice of a faster network pipe, a 50% fraction of a 1Ghz machine, or a 1Ghz desktop machine with 50% availability. In search and binding, the virtual grid system returns an instantiated virtual grid specification in which the resources have been discovered and bound in an atomic fashion from the application's perspective. With search only, the resources have not been bound and it is the responsibility of the application to acquire the resources. In this case it may be that resources returned by the search are no longer available by the time resource acquisition is attempted.

Once resources have been acquired, applications can configure, instantiate, and execute their constituent processes across the resources as they see fit. Typical operations include primitive communication, computation, and storage access. Also, the application may decide to obtain additional information about the resources via active probing and detailed queries to Grid information services.

III.B. Evolving a Virtual Grid

Virtual grids can be evolved by reconfiguring (requesting additional or releasing resources) and this activity can be triggered through notification hooks for resource failure, revocation, etc. Examples of reconfiguration scenarios include:

- the virtual grid system sends a notification to the application to signal that the resource environment (i.e. resources used by the application) has changed,
- application notifies the virtual grid implementation that there is an opportunity or need to exploit new resources, and includes with that some type of incremental resource specification (new or incremental to the current one), and
- entirely new application requests for resources are initiated based on a change in application workload.

[1] Because there is a value in returning more information in the case of failure, we include the second option, but returning the most useful information is a difficult problem.

Virtual grid reconfiguration can be performed relative to the current virtual grid configuration by passing in a description of the current resources explicitly, or by decorating a future resource request with hostnames or other attributes derived from the current virtual grid configuration.

III.C. Implementing Virtual Grids

Each class of virtual grid (e.g. a bag, a cluster, etc.) may in fact have a different specialized implementation, but these implementations will share a set of technologies which include scheduling, performance monitoring, information services, resource selection, checkpointing, etc. It's unclear at present if these implementations are separate (and composable) or if there are separate implementations for common combinations of resource structures (e.g. LooseBagof (Clusters)).

III.D. Virtual Grid Execution System Design

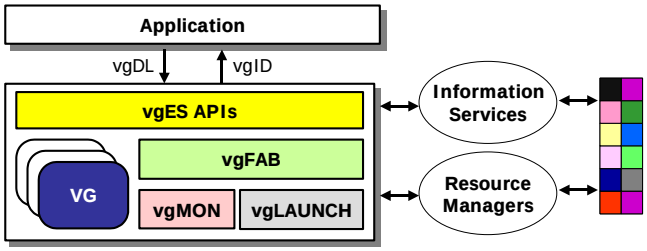

Figure 1. vgES overall architecture

The virtual grid vision is realized as part of the Virtual Grid Execution System (vgES). This work builds on and is informed by a four-year effort to build development tools for adaptive grid applications, the Grid Application Development Software Project (GrADS). Our current vgES prototype architecture builds on a key insight from GrADS that application participation (knowledge, expectations) is needed to effectively manage performance in a dynamic grid resource environment. A major innovation here is the attempt to couple applications and underlying grid resource management together through an application-oriented resource specification and an active entity (the virtual grid) that is the reification or instantiation of the application's resource environment. The application-oriented language insulates the application from the full complexity of the resource environment, and the lifecycle coupling of the application and underlying resource environment (resources and managers) enables flexible application management of resources at a high level. Note that a virtual grid does not define how the application uses resource with the virtual grid, nor does it provide a functional virtualization as in a virtual machine.

The interaction of an application with the vgES is illustrated in Figure 1. The Virtual Grid execution system (vgES) is realized in the following key elements:

- **vgDL –** a structured hierarchical language for application resource abstractions that is used to identify appropriate resources.
- **vgFAB** – the "finder and binder" that performs integrated resource selection and binding, which enables optimized resource choices in a high load resource

environment. VgFAB obtains resource information from extent information services and acquires resources by interacting with autonomous resource managers. This returns a virtual grid that communicates resource information to application in terms of application-level resource abstractions.

- **vgLAUNCH** and **vgMON** – scalable application launcher and a distributed monitoring service based on application expectations.

Virtual grids allow users to configure their own resources and change configurations according to evolving application requirements and/or resource conditions. Note that a virtual grid does not define how the application uses the resources but what should be provided and how they are managed for the application. As shown in Figure 1, users create a virtual grid by specifying application requirements in virtual grid description language (vgDL) and passing to the vgFAB. Then, vgFAB instantiates a response to the request, returns a handle (vgID) to a virtual grid instance that consists of an explicit representation (annotated tree of resources that matches the vgDL structure), and provides an interface to dynamic information about those resources. This realized virtual grid goes beyond traditional resource selection. Indeed, it consists of bound resources on which the application has been launched by vgLAUNCH and on which vgMON monitors application execution, triggering possible adaptation to changing resource/application conditions. In this sense, a VG is a "living entity" that continuously matches a (possibly evolving) vgDL specification. Part of this design has already been prototyped (see other documents on the project's Web page), and in this document we solely focus on the design and usage of the vgDL language.

IV. Specifying Virtual Grids in vgDL

A vgDL virtual grid specification consists of a core resource description and a ranking function. In this section, we describe each of these, followed by a discussion of the philosophy of our design choices and a number of the remaining open questions.

IV.A. Core Resource Description in vgDL

We provide a BNF description of the Virtual Grid Description Language (vgDL) grammar in Figures 2.1 and 2.2, which we describe hereafter.

Redline expression ::= Identifier'='
Arithmatic_expr | Logic_expr | Predicate
*Arithmatic_expr ::= A_operand [A_op A_operand]**
A_opearnd ::= Integer | Real
A_op ::= "+" | "-" | "" | "/" | "^"*
*Logic_expr::= L_operand [L_op L_operand]**
L_operand ::= Integer | Real | Boolean |

Figure 2-1. BNF grammar for Redline

*Vgrid ::= Identifier = Rdl-expression [**at** time/event]*
*Rdl-expression ::= Rdl-subexpression | ["(" Rdl-expression ")" op "(" Rdl-expression ")"]**
Rdl-subexpression ::= Associator-expression | Node-expression
Associator-expression ::= Bag-of-expression | Cluster-of-expression
*Bag-of-expression ::= **LooseBagof** "<" Identifier ">" "**[**" MinNode "**:**" MaxNode "**]**" ["**[**" Number [**"su"** | **"sec"**] "**]**"] "**;**" Node-expression |*
***TightBagof** "<"Identifier ">" "**[**" MinNode "**:**" MaxNode "**]**" ["**[**" Number [**"su"** | **"sec"**] "**]**"] "**;**" Node-expression*
Identifier ::= String
Min ::= Integer
Max ::= Integer
Node-expression ::= Identifier "=" Node-constraint
*Node-constraint ::= "**{**" Attribute-constraint | Rdl-expression "**}**" | Rdl-expression*
Attribute-constraint ::= Redline expression for attribute and constraint [see Figure 3-2]
*Cluster-of-expression ::= **Clusterof** "<" identifier ">" "**[**" MinNode "**:**" MaxNode [**","** MaxTime **":"** "MinTime"] "**]**" "**;**" Node-expression*
*op := **close** | **far** | **highBW** | **lowBW***

Figure 2-2. BNF for Virtual Grid Description Language (vgDL)

A vgDL specification (Vgrid) consists of a resource description and of an optional specification of when the Virtual Grid is needed (via the at keyword). If no at value is specified, then the Virtual Grid is needed immediately. Otherwise, the application can specify a value as an absolute date in time, or as an event that triggers the acquisition of the virtual grid (e.g., a critical resource becomes available). The exact syntax for the at value specification is still to be determined. The resource description can be the application of an operator between two descriptions (with possibly multiple levels of parenthesization). Each such description can be just a node specification (Node-expression) that specifies requirements for a single resource, or an association (Associator) between node specifications or resource descriptions. At the moment we have three associators:

- LooseBagOf: Set of heterogeneous processors with possibly "poor" connectivity
- TightBagOf: Set of heterogeneous processors with "good" connectivity
- Clusterof: A set of homogeneous processors with "good" connectivity

Each such associator specifies a range for the number of associated elements (between MinNode and MaxNode). In addition, the user can optionally specify a notion of how long the resources are needed for. This is done be specifying a Number of either (i) *service units*, as defined in the traditional sense by, for instance, supercomputer centers, or as defined as number of cycles on a reference machine, or (ii) *units of time*, such as seconds. In the syntax defined above, a virtual grid specification could thus ask for a cluster for 100su (i.e., 100 service units), or for 100sec (i.e., 100 seconds). Finally, one can specify constraints of the individual resource that make up these elements with a node specification.

Aggregate properties were later added to the design.

Note that with the at keyword and the specification of needed work units or number of seconds, the Virtual Grid system can interact with resources that enable advanced reservation. If resources do not provide such capabilities, then the Virtual Grid system must rely on statistical estimates and predictions of resource availability and pick resources that can match the desired application needs with highest confidence, knowing that there can be no guarantees.

At the moment we provide 4 operators that can be applied between resource descriptions (close, far, highBW, lowBW). These operators indicate coarse notions of network proximity in terms of latency and bandwidths.

Note that the philosophical approach here is to make it possible to specify associators and operations among associators at a high and qualitative level. The implementation will use specific values to determine what qualifies as "close" and "far", and these values will evolve throughout years due to technology advances. Note again that applications that wishes to use quantitative measurements of, say, bandwidth, can query the grid information systems once Virtual Grid resources have been bound in order to apply possibly sophisticated scheduling algorithms and other optimizations. The point is that taking such qualitative data during resource selection among the vast numbers of available resources may be impractical and may in fact be of dubious overall benefit, hence a qualitative approach. This last point is up for debate and part of the VGrADS research will consist in validating or invalidating it.

The following are examples of possible virtual grid specifications written with the above language, based on our initial group discussion of virtual grids and our past GrADS experience:

- Iter = Clusterof<Node1>[8:32] ; Node1 = {constraint1, constraint2,...} at 01/05/2004:12AM
- Scalapack = Clusterof<Node2>[8:32] [3600sec]; Node2 = {constraint1, constraint2,...}
- Fasta = LooseBagof<Node3>[8:32]; Node3 = {have database, ...}
- EMAN = LooseBagof<Node4>[1:64]; Node4 = Clusterof<Node5>[8:32]; Node5 = {constraint1, constraint2,...}
- SAT = LooseBagof <Node6>[1:100000]; Node6 = {}
- LEAD/MEAD = Clusterof<Node7>[64:10000] [400su]; Node7 = {contraint1, constraint2,...}
- EOL = LooseBagof<Dnode>[8:32]; Dnode = {have database parts, ...} far
- LooseBagof <Wnode>[32:4096]; Wnode = {constraint1, constraint2,...}
- EOL2 = LooseBagof<LumpedNode>[1:100]; LumpedNode = (Dnode={have database parts, …} highBW LooseBagof<Wnode>[32:1024]; Wnode = {constraint1, constraint2,...})

We provide case-studies in Section V in which we develop Virtual Grid abstractions and explain how they related to different application execution scenarios.

IV.B Ranking Functions

Applications provides a "ranking function", which can be applied at each level—node choice, cluster size & configuration choice, bag size, etc. The ranking function therefore takes inputs like: cluster size, bag size, network speed, node configuration, and can also include predicates on elements of clusters or bags. The ranking function can be expressed as a function of the resources using the variable names. At the moment, we envision this function to be using arithmetic and boolean operators. Its specific syntax is still to be designed based on feedback from application studies.

IV.C. Discussion

Our simple virtual grid description is designed to facilitate rapid resource selection. The ranking function allows applications to express their preferences for specific resources. Virtual grids do not prevent an application from significant further optimization at runtime based on the actual resources selected and bound (e.g. custom code generation for the processor pipeline or custom scheduling and data decomposition based on the actual dynamic network performance). Further, the notion here is to avoid capturing the full complexity of the application's resource use in the resource description. Rather, the ideal is to move to simplified views of typical resource request structures and ultimately a notion of "classes of service".

A particular application can reasonably choose several different virtual grid abstractions—and are expected to do so. Such choices reflect the needs of applications developers to manage their application complexity and the effort needed to "port and optimize" their application for the execution environment. For example, in the subsequent sections which detail application examples, an evolution of possible virtual grid descriptions is given for each application.

Our current virtual grid language is performance-centric, and could be extended naturally by attributes such as reliability, availability, failure domain, etc. However, adding such attributes may not be the right approach – as orthogonality is not required. Rather, an approach based on classes of service that couple these attributes is likely to be most manageable.

V. Application Case Studies

We use several application case studies to demonstrate the flexibility and use of virtual grid abstractions in specific contexts. For the moment these applications are: Encyclopedia of Life, EMAN, and GridSAT.

V.A Encyclopedia of Life

The EOL application can be run with different levels of abstraction for the underlying resources. The lower-level the abstraction, the more sophisticated the application scheduling strategy can be, with both a bigger hope of extracting the best performance

out of the underlying resources and a bigger burden placed on the application developers. When referring to EOL, we specifically refer to the computationally intensive portion, which is the iGAP (integrated Genome Annotation Pipeline) described in the EOL document located on the VGrADS Website. Below we outline several scenarios that correspond to increasing levels of sophistication for EOL, as well as what the Virtual Grid user needs to specify in terms of a desired virtual grid abstraction. This virtual grid abstraction is expressed with the language presented in Section IV.A. Note that in all that follows we omit extraneous details concerning the logistical requirements of application execution (i.e., software is installed, is installable, software licenses are available, which binaries are available for which architectures). Indeed, our goal is to illustrate the expressive power of the virtual grid abstraction and how it affects expected application performance. Further details about application requirements can be easily added.

Sequential EOL (EOL1)

This is just a sequential execution of EOL on a single machine. All databases needed for running EOL are local to the machine, which is the only knowledge required for the virtual grid user writing the abstraction specification. In the specification below we assume that the user requires some amount of RAM on the machine (for instance so that each database can be loaded entirely in RAM to avoid repeated access to disk, which is typical with bioinformatics applications).

EOL1 = Node = { has databases; memory >= 1GB}

Rank(Node)=cpu;

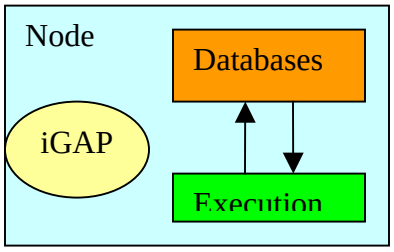

Figure 3. Sequential EOL (EOL1)

Parallel Workers EOL (EOL2)

In this scenario, the virtual grid user wished the EOL application to run on a collection of individual machines, or computational nodes (the CNodes below). The databases necessary for execution of the application are located on one machine (the SNode below). Given that the EOL databases are not very large and that the application is compute-intensive (at least the part of the iGAP pipeline that we are considering in this project), the user specifies that the database node can be “far” from the bag of nodes. (Of course, the Virtual Grid system may return a Virtual Grid binding in which the SNode is close to all the CNodes). The abstraction is specified as follows:

EOL2 = SNode ={has databases} far LooseBagOf<CNode>[32:4096];CNode = {memory >= 1GB}

Rank(CNode)=cpu;

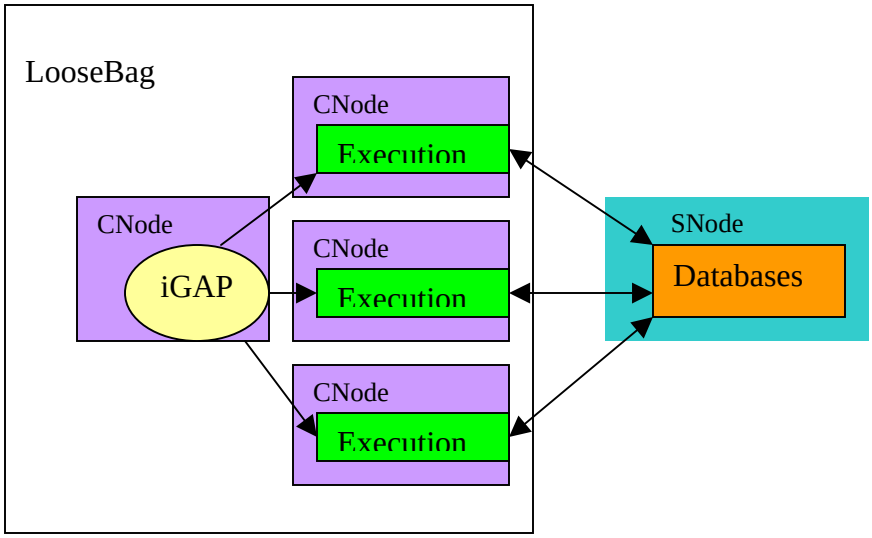

Figure 4. Parallel Workers EOL (EOL2)

Parallel Workers and Databases EOL (EOL3)

In this scenario, databases are fully replicated onto several nodes. This makes it possible to take advantage of greater range of resources. As in the previous scenario, the worker compute nodes, CNodes, can execute EOL in parallel by accessing the database remotely from SNodes. To each bag of CNodes is associated a SNode containing the databases, and the virtual grid consists of several such "LumpedNodes". The programmers need not know anything about grid or virtual grid abstractions to run in this mode, other than there can be separate database and worker nodes, but will need to split execution of the iGAP pipeline among LumpedNodes This is the default behavior of EOL when run on more than one servers or clusters.

EOL3 = LooseBagOf<LumpedNode>[1:32];LumpedNode = (SNode={has databases} far LooseBagOf<CNode>[8:128] CNode = {memory >= 1GB})

Rank(CNode)=cpu;
Rank(LumpedNode)=count(LumpedNode[])

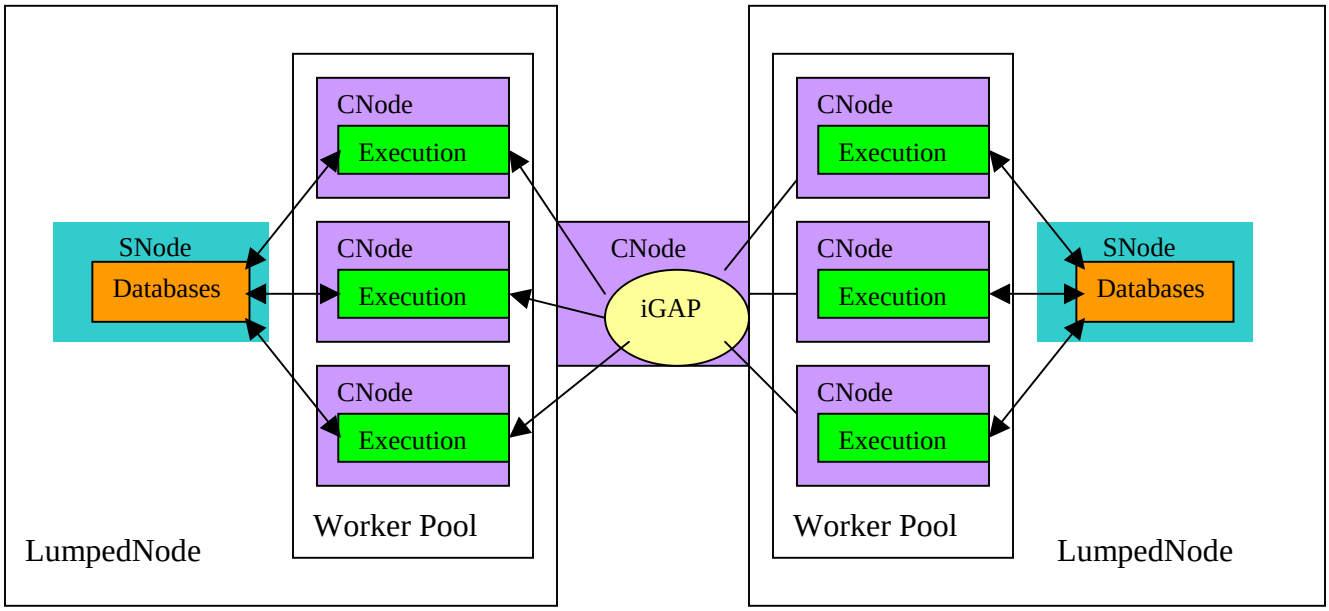

Figure 5. Parallel Workers and Databases EOL (EOL3)

Smart-Distributed EOL (EOL4)

Currently EOL is deployed using the APST software, which uses scheduling heuristics that have been shown to work well in practice. By contrast to EOL3, EOL4 expresses no fixed association between database nodes and particular worker pools. The application does database replica selection based on network latency and bandwidth. Additionally, APST would divide and map the workload based on the characteristics of the worker nodes to balance the workload. To use a virtual grid and optimize in this fashion, the programmer should understand how the placement of databases in relation of the worker nodes affects the performance of EOL.

EOL4 = LooseBagOf<SNode>[8:32]: SNode = {has databases} far LooseBagOf<CNode>[32:1024]: CNode = {memory >= 1GB}

Rank(CNode)=count(Cnode[]);

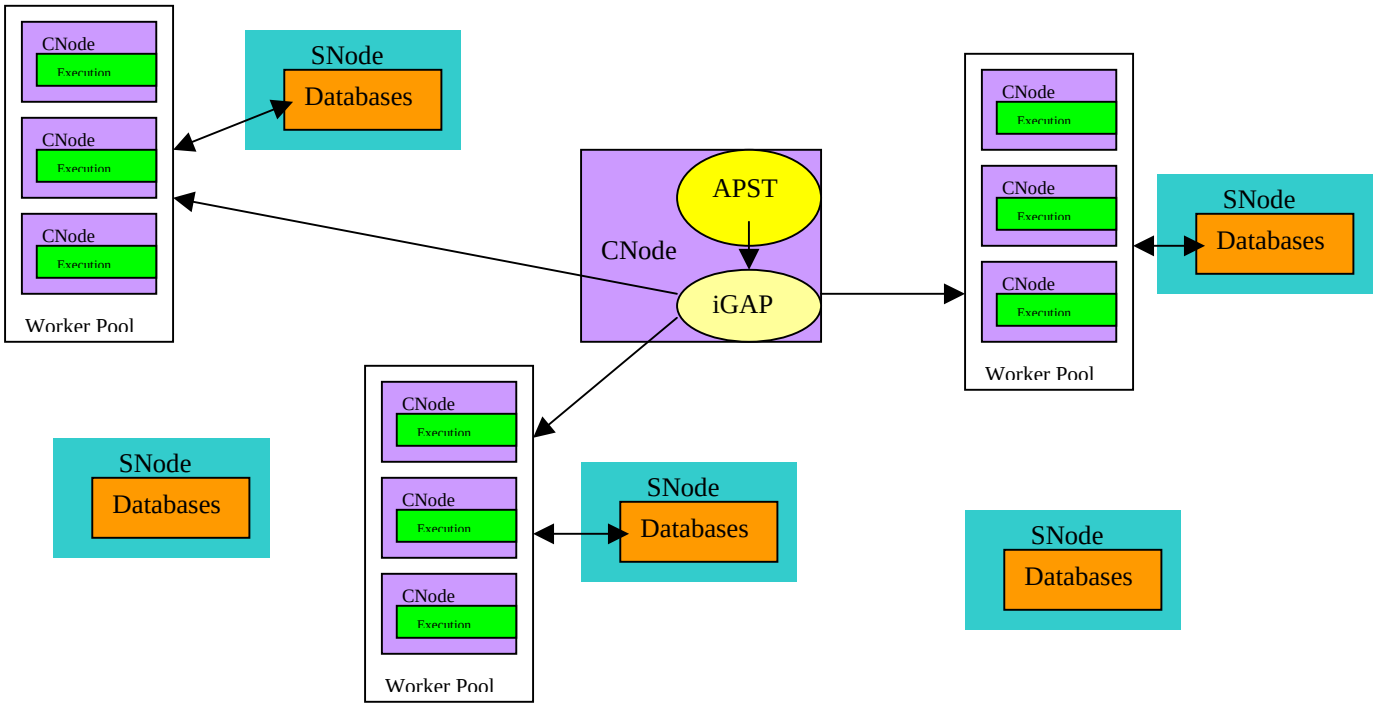

Figure 6. Smart-Distributed EOL (EOL4)

Data Smart Distributed EOL (EOL5)

EOL5 has all the advantages of EOL4, plus a notion of partial distribution of database parts (which is meaningful for many bioinformatics codes). This partial distribution admits more resource configurations and requires optimized workload division so that worker nodes are assigned load specific to a partial or full database and the results. In some cases, machines with smaller memories that can't hold a full database in memory can be employed. With only partial database replication, the application must manage load balance across servers and managing result aggregation. The current version of EOL cannot be deployed in such a fashion and would need to be modified to support splitting workloads among different partial databases. The virtual grid abstraction specification then corresponds to a flat structure in which there are storage nodes and compute nodes, as well as nodes that hold databases. The databases can then be split and moved to storage nodes when/if needed.

EOL5 = DBNode = {has all databases} HighBW LooseBagOf<SNode>[8:32]: SNode = {disk > 100GB} far LooseBagOf<CNode>[32:1024] CNode = {memory >= 500MB, disk >= 10GB})

Rank(CNode)=count(Cnode[]);

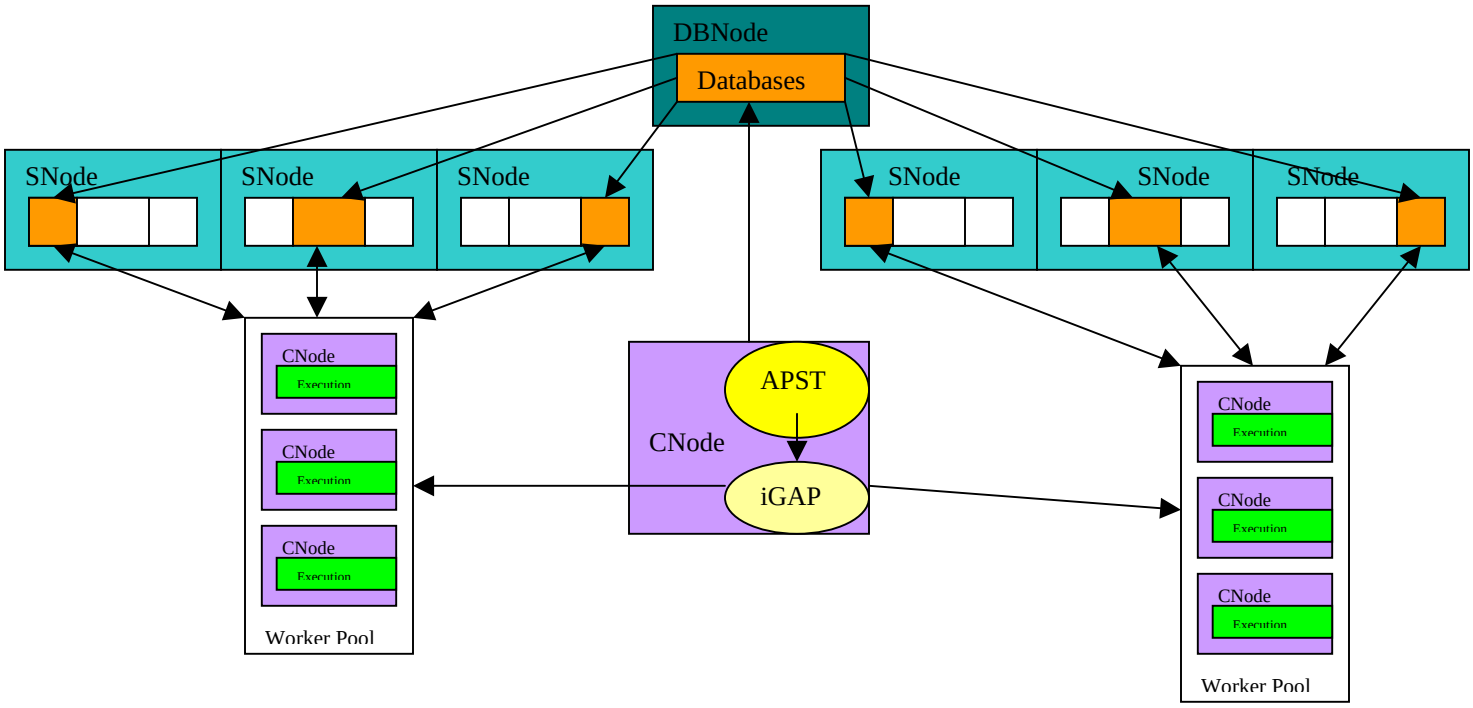

Figure 7. Data Smart Distributed EOL (EOL5)

V.B EMAN

The EMAN application can be run with different levels of abstraction for the underlying resources. The lower-level the abstraction, the more sophisticated the application scheduling strategy can be, with both a bigger hope of extracting the best performance out of the underlying resources and a bigger burden placed on the application developers. Below we outline several scenarios that correspond to increasing levels of sophistication for EMAN, as well as what the Virtual Grid user needs to specify in terms of a desired virtual grid abstraction. This virtual grid abstraction is expressed with the language presented in Section IV. Note that in all that follows we omit extraneous details concerning the logistical requirements of application execution (i.e., software is installed, is installable, software licenses are available, which binaries are available for which architectures). Indeed, our goal is to illustrate the expressive power of the virtual grid abstraction and how it affects expected application performance. Further details about such application requirements can be easily added.

Sequential EMAN (EMAN1)

This is just a sequential execution of EMAN on a single host. The input to the EMAN DAG should be stored on the host. The below example assumes that EMAN required 500MB of RAM to run, and that the input data + temporary data requires 1GB or storage (which is well above the current requirements of EMAN as it is ran for VGrADS).

EMAN1 = Node = { RAM > 500MB, Disk > 1GB }

Once acquired, it is the responsibility of the EMAN application to ship the initial input data to the host.

Single Cluster EMAN (EMAN2)

The EMAN application consists of a linear DAG of 8 tasks (in its current incarnation). 4 of the nodes are sequential, and 4 are parameter sweep nodes and can (conceptually) be executed on collection of loosely connected hosts. However, out of these last 4 nodes, some require a shared file system among all hosts (as in an actual cluster). So one easy way to run EMAN is just to run it on a single cluster that has a shared file system:

EMAN2 = Clusterof <Node> [32:128]; Node = { RAM > 500MB, Disk > 1GB}

In this scenario, a node in the cluster is picked to run the sequential tasks of the EMAN DAG.

Multiple Cluster EMAN (EMAN3)

Some of the parameter sweep tasks in the EMAN DAG can be run across multiple clusters. Given the fact that EMAN is compute-intensive (see the VGrADS EMAN document), it is not required that these clusters be close to each other.

EMAN3 = LooseBagof<Cluster>[1:4]; Cluster = Clusterof<Node>[8:32]; Node = {RAM > 500MB, Disk > 1GB}

This is the way in which EMAN is currently being executed. The application could pick the “best” cluster to run the tasks that require a cluster, and the other potential clusters can be used for those tasks that can be executed over multiple clusters.

Evolved EMAN (EMAN4)

It is expected that EMAN will evolve to encompass the execution of a non-linear DAG, with parameter sweep tasks as well as sequential tasks, and with portions of the DAG that are data-intensive. This corresponds to a restricted case of mixed-parallel applications. The general specification below:

EMAN4 = LooseBagOf<Node>[1:512]: Node = {RAM > 500MB, Disk > 1GB} far LooseBagOf<Cluster>[1:8]: Cluster = ClusterOf<Node>[8:64]: Node = {RAM > 500MB, Disk > 1GB}

would make it possible to use sophisticated scheduling heuristics for mixed-parallelism over a number of individual nodes and a few clusters. More refined descriptions could be written that specify that some of the resource should be close to each other.

V.C. GridSAT

GridSAT is a parallel and complete satisfiability solver used to solve non-trivial SAT problems is a grid environment (large number of widely distributed, heterogeneous resources). The application uses a parallel solver algorithm based on Chaff to essentially attempt to solve SAT problems of the form 'given a large, non-trivial Boolean expression, is there a variable configuration (and what are the variable values) which results in the expression evaluating to TRUE?'

GridSAT was designed explicitly to run in grid environments and has built in intelligent acquisition of additional resources, scheduling, and even migration/load balance mechanisms. GridSATuses of tightly-coupled bags of nodes for local load balancing, and depends on sparse high bandwidth connectivity amongst these bags for higher level work spreading (finding new resources and increasing parallelism). Beyond resource acquisition, GridSAT also uses sophisticated application and network performance information to make scheduling decisions.

GridSAT Resource Selection

To request the initial set of resources, GridSAT simply requests a loose bag of tight bags with particular node counts and node requirements.

```
GridSAT1 = LooseBag<BagNode>[16:1024]{BagNode =
TightBagof<Node>[8:32]{Node = memory > 128MB, cpu > 1Ghz}

Rank(Node) = cpu
```

Because connectivity amongst the BagNodes is important for load balance and work distribution, there needs to be some way to express desired connectivity. GridSAT wants BagNodes in the LooseBag to have high bandwidth connections to the other BagNodes, the following ranking function might be added.

```
GridSAT2 = LooseBag<BagNode>[16:1024]{BagNode =
TightBagof<Node>[8:32]{Node = memory > 128MB, cpu > 1Ghz}

Rank(Node) = cpu
Rank(BagNode) = count(BN in BagNode[] | BN HighBW BagNode)
```

In this case, the ranking function for the BagNodes expresses that the better they are connected, the higher the desirability. The current design of virtual grids does not support a full range of capabilities for augmenting an existing virtual grid with additional resources, but this can be done simply by equating the current BagNodes with the term BagNode[] in the above ranking and requesting an additional set of new BagNodes, something like the following:

GridSAT2-extension = NewBagNode = TightBagof<Node>[8:32]{Node = memory > 128MB, cpu > 1Ghz}

Rank(Node) = cpu
Rank(NewBagNode) = count(BN in BagNode[] | BN HighBW NewBagNode)

GridSAT Scheduling

In addition to using connectivity information to select initial resources, GridSAT uses network connectivity to perform scheduling and migration (mapping of dynamic load to resources) in the course of its execution. To do so, GridSAT uses a number of carefully tuned decision procedures that trade the expected cost of data movement against the expected increase in computational speed resulting from a migration or work spreading action. To support these decisions, it seems natural that a virtual grid might provide some information service interfaces. While there are many possibilities to address this, one approach would be to express this information as a hierarchical labeling on the LooseBag and the TightBags which can be used to identify candidate resources. For example:

Candidates = enumerate (BN in BagNode[] | BN HighBW currentHost

Or a more comprehensive system that presents a full topology of the LooseBag such as:

Forall BN1,BN2 in BagNode[] LooseBagHighBWTopology[BN1,BN2] = (BN1 HighBW BN2))

Armed with operations on the topology, it is then possible for a scheduler to adapt GridSAT's execution appropriately to the underlying resource structure. These labels or properties on the bound and prospective resources could be provided by an underlying resource characterization system such as NWS or OpenView.

VI. Issues and Future Work

The Virtual Grid descriptions defined in this document have the goal of being simple and yet expressive. As a result we have made a number of initial choices that attempt at striking a balance between simplicity and expressivity. Some of the open issues are summarized here:

- No means of dealing with aggregate resource properties

- Better way to deal with data properties of resources

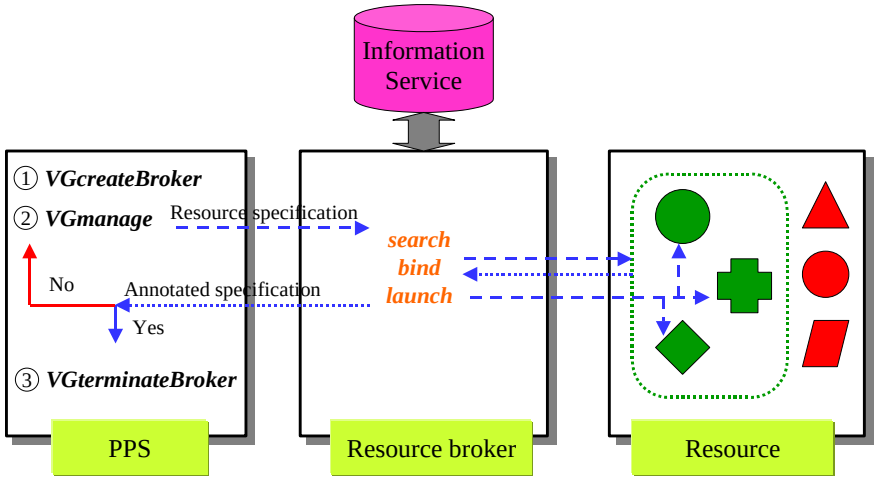

Figure 8: Integrated Selection and Binding

- Richer set of network properties / proximity relationships (quantitative?)
- No full set of attributes (limited list)
- No treatment of fault-tolerance, statistical properties, security properties, …

VII. Interaction Diagrams

This section includes a set of diagrams, which illustrate the interactions between application program, PPS, and the various parts of the execution system. Each of the interactions is labeled with the function name (defined in the Virtual Grids API document).

VII.A Integrated Selection and Binding

This figure illustrates the simplest use of the Virtual Grid system. The application requests a set of resources, and the execution system (the resource broker in this case) selects and binds an appropriate configuration, returning it to the application.

① PPS activates the virtual grid broker.

② The virtual grid execution system searches a resource collection, instantiates on it, and launches the script on each resource in the virtual grid by using VGmanage with a VG_MODE_SEARCH_BIND or VG_MODE_BIND_ON_EXISTING mode.

③ PPS deactivates the virtual grid broker.

VII.B Separate Selection and Binding

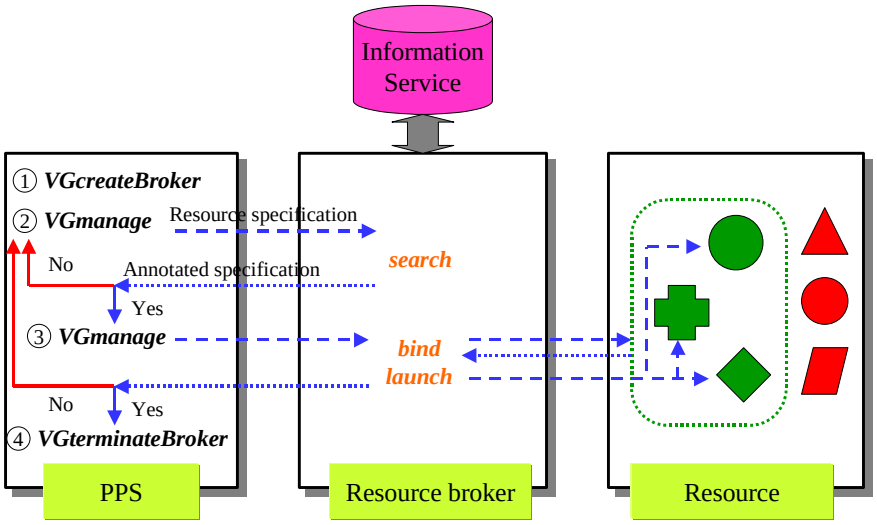

Figure 9: Separate Selection and Binding

This figure illustrates separate use of Virtual Grid resource selection and binding. First the application requests a set of resources, and the execution system (the resource broker in this case) returns a set that matches the request. Subsequently, the application requests that the set of resource be bound for the application. However, because there are other resource consumers in the system, success in binding those resources is not ensured. If the finding process fails, the application may need to return to the selection step to identify new resources.

① PPS activates the virtual grid broker.
② The virtual grid execution system searches a resource collection satisfying the virtual grid specification by using VGmanage with a VG_MODE_SEARCH mode.
③ The virtual grid execution system launches the script on each resource in the virtual grid.
④ PPS deactivates the virtual grid broker.

VII.C Application-Controlled Dynamic Resource Change

After a virtual grid has been created, adaptation of the resources can be achieved under application control. The resource monitoring can trigger callback to the application, which then requests the replacement of some resources or the application can of its own initiative request that a set of resources be replaced.

① PPS activates the virtual grid broker and creates a resource monitor for this application.

② The virtual grid execution system searches a resource collection, instantiates on it, and launches the script on each resource in the virtual grid.

③ When the resource monitor detects any significant changes of resources, it invokes a callback of the application.

④ In response to the callback, the application initiates a change of resource. For example, the application can replace the resource with poor performance with another resource with better performance. Alternatively, the application could initiate a change of resources of its own volition at any time.

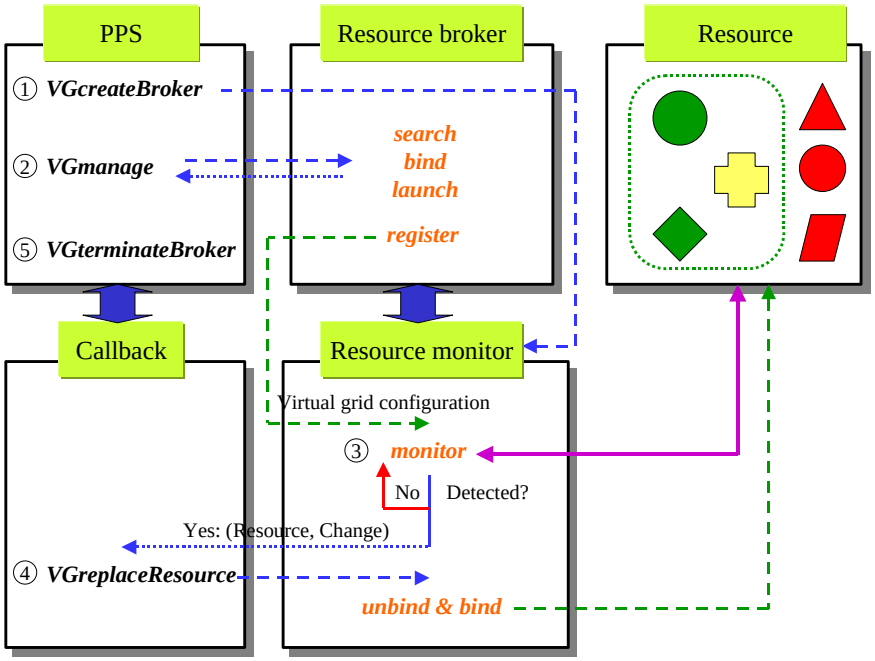

Figure 10. Application-controlled Dynamic Resource Change

⑤ PPS deactivates the virtual grid broker and terminates the resource monitor.

VII.D Transparent Dynamic Resource Change

It is also possible to design Virtual Grids that transparently adapt resources beneath an application. Because it is not our direct resource objective to create the virtualization required for transparent substitution, we presume those issues are addressed in a virtualization or checkpointing system layer or at the application level. In this scenario, replacement happens and is followed by a callback notifying the application.

① PPS activates the virtual grid broker and creates a resource monitor for this application.

② The virtual grid execution system searches a resource collection, instantiates on it, and launches the script on each resource in the virtual grid.

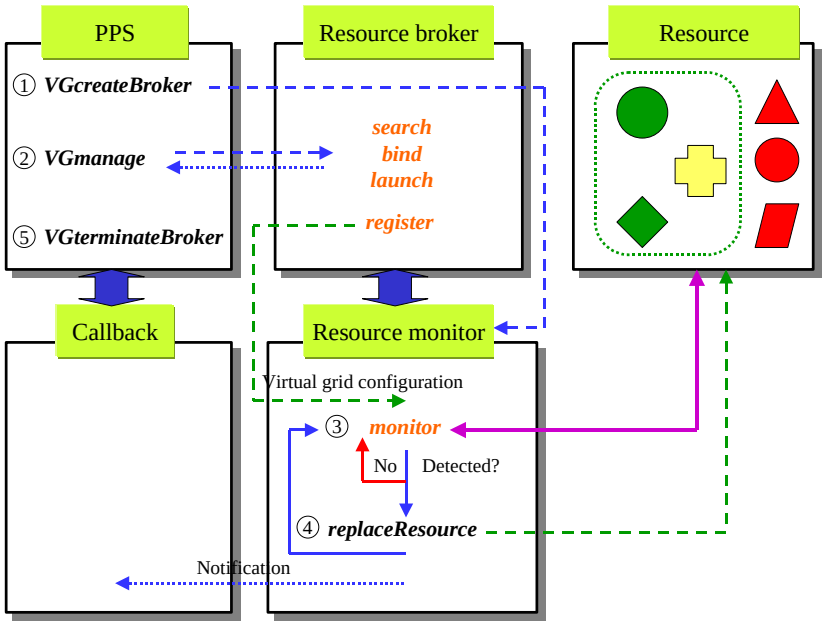

Figure 10: Transparent Dynamic Resource Change

③ When the resource monitor detects any significant changes of resources, it invokes internal routines to handle this.

④ The execution system can replace the resource with poor performance with another resource with better performance. The resource monitor notifies the application that there was resource reconfiguration.

⑤ PPS deactivates the virtual grid broker and terminates the resource monitor.

Acknowledgements

These ideas in this document have benefited from a number discussions and the generous input of many of our colleagues, including Rich Wolski, Jack Dongarra, Carl Kesselman, Dan Reed, Fran Berman, Ken Kennedy, Lennart Johnsson, Anirban Mandal, Chuck Koelbel, Mark Mazina, and other members of the Virtual Grid Application Development Software Team.

Supported in part by the National Science Foundation under awards NSF Cooperative Agreement ANI-0225642 (OptIPuter), NSF CCR-0331645 (VGrADS), NSF ACI-0305390, and NSF Research Infrastructure Grant EIA-0303622. Support from the UCSD Center for Networked Systems, BigBangwidth, and Fujitsu is also gratefully acknowledged.

Any errors or inaccuracies are of course the responsibility of the authors…