Vision and Plan for a Next Generation Resource Manager

LLNL-TR-636552-DRAFT

Dong H. Ahn, ahn1@llnl.gov
Jim Garlick, garlick@llnl.gov
Mark Grondona, mgrondona@llnl.gov
Don Lipari, lipari@llnl.gov

May 14, 2013

1 Overview

Resource Management (RM) software is critical for High Performance Computing (HPC). It is the centerpiece that allows efficient execution of HPC applications while providing an HPC center with the main means to maximize the utilization of its computing resources. However, several growing trends make even the best-in-breed RM software largely ineffective. As numbers and types of compute cores of HPC systems continue to grow, key RM challenges associated only with today's *capability-class* machines are becoming increasingly pervasive for *all* computing resources including commodity Linux clusters. The challenges include having to provide extreme scalability, low noise, fault tolerance, and heterogeneity management while under a strict power budget.

ned-review: Citation needed. The shortcomings identified in this section form the core justification for why NGRM is needed, so you should be prepared to back them up with references.

In addition, greater difficulties in code development on larger systems have begun to impose far more complex requirements on the RM. For example, without adequate RM support, debugging, tuning, testing and verification of the applications have become too difficult and time-consuming for end-users. The next-generation code development environments require the RM to provide effective mechanisms to support the reproducible results of program execution, to provide accurate correlations between user-level errors and system-level events, and to integrate and accelerate a rich set of scalable tools.

Further, a greater interplay among various classes of clusters across the entire computing facility makes the current paradigm of single-cluster scheduling largely ineffective. An application running on a compute cluster heavily utilizes site-wide shared resources such as I/O and visualization clusters. Thus, avoiding any significant site-wide bottleneck requires the RM to schedule the job to all dependent resources together. In short, without the RM that can effectively address all of these challenges, it has become apparent that HPC centers will suffer a significant loss in both user productivity and efficient uses of next-generation computing resources.

ned-review: Citation needed. As above, we need more than anecdotal evidence that Slurm can't meet future requirements.

Slurm [33] is arguably the best-in-breed, open-sourced RM designed for commodity Linux clusters. Livermore Computing (LC) at Lawrence Livermore National Laboratory (LLNL) designed and led its implementation in 2002 and since then has facilitated broad adoption outside of LLNL. Slurm's original design was for a moderate-size Linux cluster with $\mathcal{O}(2K)$ compute nodes connected in a single interconnect domain and MPICH-based, bulk-synchronous applications. In the last decade, beginning with multi-core support added by Hewlett-Packard [12], Slurm has been modified to meet emerging challenges that stray outside its core design. For example, recently it has been adapted to act as a scheduling and job submission layer on top of proprietary RM software on IBM Blue Gene and Cray systems, to support new run-times such as OpenMPI and MapReduce [14], to implement hierarchical communication to increase scalability, and to be tied into a Moab [7] grid. These adaptations have accrued without fundamentally changing the core paradigm and design of Slurm, which is that of a monolithic, centralized controller with compute nodes as the main, scheduled resource type. As a result the Slurm implementation of these add-ons is less functional and effective than it

could be with a redesign, and the Slurm code base grows increasingly unmaintainable, burdened as it is with such after-thoughts.

Our response to this critical need is the Next Generation Resource Manager (NGRM), an RM software *framework* that can solve the key emerging challenges in a simple, extensible, distributed and autonomous fashion. It aims at managing the whole center as one common pool of *diverse* resources. Hence, scheduling decisions will be far more efficient as well as flexible to accommodate emerging constraints such as a strict power bound. Further, NGRM integrates system monitoring, system administration, lightweight virtualization, and distributed tool communication capabilities that are currently provided by disjoint and often overlapping software. Integration of these facilities within the common framework designed from the ground up for scalability, security, and fault tolerance will result in a more efficient and capable system.

ned-review: What about the security model–should it be its own thrust area? It seems to me a robust security model will need to be incorporated from the ground up, so it should be one of the first things we work on. **jg:** Security model is developed in comms thrust.

We realize that NGRM represents a paradigm shift for HPC resource management, and yet we must address a wide range of challenges in relatively short order with limited resources. Thus, we organize our project plan around four research and development thrust areas and seek to advance them systematically. These areas are called: *Communications Framework*, *Resource Management*, *Monitoring*, and *Workload Runtime And Placement*. They are relatively independent but can significantly build on the strength of one another through well-known interfaces and common design and development principles. Thus, reaching all major milestones of these thrust areas will represent the completion of the first round of NGRM development.

Overall, NGRM will significantly improve operational efficiency for scientific application development and execution, and further for computing resources of the entire HPC center. It will also provide a foundation for further extension and customization, allowing agile responses to site-specific scheduling issues. Perhaps more importantly, NGRM positions us to cope with a blend of interrelated, diverse extreme-scale computing resources, the landscape of high-end HPC centers in just a few years down the road.

The rest of the paper is organized as follows. Section 2 presents NGRM's vision and its new capabilities in more detail. In Section 3, we discuss the key software design challenges and the conceptual models that embody the new paradigm while addressing these design challenges. Section 4 then describes our software design and also highlights the four research and development thrust areas and their relations. The following sections then go over and detail each of these areas including its work breakdown structure (WBS) and associated work items. Finally, Section **??** defines our schedules for milestones and deliverables.

2 Vision and New Capabilities

The vision of NGRM is to create a scalable RM software system that drastically improves operational efficiency and user productivity for workloads on *capacity-class* compute systems. With a trend towards ever-growing numbers and types of compute cores, however, this system class has been subject to the challenges that today's *capability-class* machines have been facing. These challenges include having to provide extreme scalability, low noise, fault tolerance, and heterogeneity management while under a strict power budget. Worse, the workloads themselves are also becoming increasingly diverse, dynamic, and large. Thus, fully realizing our vision through these challenges requires a *paradigm shift* in how the RM should manage, model, schedule, and allocate its resources.

In the new paradigm, the RM must be capable of imposing highly complex resource bounds to guarantee the highest operational efficiency at any level across the computing facility, while at the same time enabling most efficient execution and scheduling of the workloads within these bounds. Thus, the RM must manage the entire facility as one common pool of resources. The ability to see a broader spectrum of resources and their various constraints can then lead to most efficient scheduling strategies and execution environments. Further, the same ability will ease efforts to diagnose errors for both end users and support staff by associating jobs with other facility-wide events. The new paradigm also demands that the RM model various types of resources and their relationships beyond the traditional resource representation: i.e., a simple collection of compute nodes. The rich resource model will allow the RM to allocate computing resources tailored to the disparate limiting factors of

our applications: e.g., an application may be compute-bound while others are I/O-bound or power-bound. Under the new paradigm, the resource allocations must also be elastic. An application may have different phases with disparate performance-limiting factors; it must be able to grow and shrink its resource allocation dynamically. The global resource view, rich resource model, and elasticity represent the fundamental characteristics of the new resource management paradigm.

Further, the new paradigm must provide a central framework to integrate other relevant software. The software components should include system monitoring and administration, lightweight virtualization, and scalable tool communication. The integration will facilitate a higher level of leverage among these essential computing elements, and this will lead to significantly higher productivity for both end users and system administrators. In addition, as these capabilities are currently provided through disjoint and often overlapping software, the integration will substantially reduce the costs needed for developing and maintaining individual software. In the following, we further detail the key ideas and new capabilities under the new paradigm.

Center as a Cluster: Unlike the traditional paradigm of running an RM instance on each separately managed cluster, the new paradigm must manage the entire computing facility as one pool of resources. There must be a single site-wide system image from the perspective of users as well as system administrators. With this approach, file system servers such as Lustre clusters, and visualization and serial batch systems can be aggregated with compute clusters into one management domain. Hence, our RM can make better global scheduling decisions.

With integrated, site-wide monitoring, it becomes easier for the new paradigm to associate global Reliability, Availability and Serviceability (RAS) events such as a global file system failure with an affected job and to make that information part of the job's data *provenance* record. When the RM obtains a global view of resources including shared persistent storage, it becomes possible to consider the scheduling of I/O along with computation. In combination with I/O forwarding software, our RM could easily set up unique I/O forwarding topologies for each job.

This paradigm also has many possible advantages for system administration. The resource inventory for the center is managed from a central point and contains details that can drive center-wide configuration management. A *cluster* is diminished as a primary, user-facing data center entity and instead can be viewed as an arbitrary collection of resources, part of a larger system, that happens to be attached to a single interconect domain or that have other similar characteristics. A cluster downtime, formerly viewed as a period of unavailability, can be viewed in the new paradigm as a period of degraded performance. Through lightweight virtualization, users obtain a degree of independence from system software updates, which in some cases can quietly roll out across the center between jobs with minimal impact. As the new paradigm embraces heterogeneity within the larger system, new resources can be purchased and added on, as dictated by demand, without the need to build a standalone cluster entity, separately named and managed, for every new type of hardware introduced. In short, system administration activities can take place in a more centralized, less visible manner such that they are no longer perceived by users as at odds with their productivity.

Diverse Compute Resources: The traditional paradigm has solely focused on node- and/or CPU-centric scheduling. With the advent of hybrid compute systems utilizing specialized, heterogeneous resources and also of other bounding resources like power, this simple resource model has become largely ineffective. Rather than perpetuating a node-centric resource model with support for other resources as an add-on, the new paradigm must embrace the concept of generalized resources: the idea of a resource is kept as generic as possible. This will not only facilitate simpler handling of diverse resources, but also enable future expansion to resource types that have yet to be conceived. New resource types can be provided through configuration changes and/or simple extensions that inherit their attributes from base types to maximize reuse and foment collaboration. Various resource topologies can be encoded via configuration, and resources can also be given tags or labels to which resource requests may refer. The new paradigm must also provide a generic resource query language to allow flexible specification of resource requirements.

jeff-review: There have been efforts in the past to maintain provenance for simulations, but the ones that didn't fail were cumbersome. Having some of these features built into the RM is pretty cool and could be one of the nice value-added features in your selling points slide.

Data Provenance and Reproducibility: What about the security model–should it be its own thrust area? It seems to me As simulation plays an increasingly central role in scientific investigation, reproducibility of results is more important than ever before. A result should be accompanied by a data provenance record that can be used by others to reconstruct the inputs and conditions that led to that result. It should also record unusual system activities such as RAS events that might help in a post-mortem analysis when expected results are not obtained. The new paradigm must produce such a record for every job. Long running parameter studies or uncertainty quantification runs require stability for long periods of time.

Low Noise: As the number of processes in a parallel application increases, OS scheduling jitter affects their executions to a larger degree. Minimizing the user-space system software contribution to the OS jitter must be one of the primary goals of the new paradigm. Thus, the new paradigm must supplant the independent monitoring, remote shell, and cron services that contribute to noise today. The integrated services will allow users to dial up or down the verbosity and frequency of monitoring, depending on their debug/monitoring needs versus their application's noise sensitivity. Cron (periodic housekeeping) jobs and rsh (remote command executions) can be performed through the RM to minimize their impact, such as running them between jobs or synchronized across jobs. The new RM must also be flexible enough to allow implementation of other strategies for reducing the impact of noise, such as scheduling all system activity to a CPU core that is not shared with the application.

Fault Tolerance: As the new paradigm manages the entire computing facility, the RM's tolerance to hardware and software faults and failures is no longer optional. Thus, the new RM must have no single point of failure. Further, it must support version interoperability that allows *live* software upgrades and facilitate a rolling update across the center without negatively impacting overall availability of the facility and/or running workloads.

Security: The new paradigm must continue to support and strengthen privacy and integrity on the network to limit vulnerability to attacks involving physical access to a system or its networks.

Research and Tool Friendly: The new paradigm needs to facilitate development and use of scalable run-time code development tools to improve the productivity of users who must develop, debug, optimize, test and verify their code on the next-generation systems. Similarly, it must also facilitate research with the same goal in mind. Specifically, the new paradigm must provide highly scalable infrastructure and rich run-time interfaces on which tools can build. Further, it must have an ability to capture and publish sanitized system data at all levels to facilitate the use of this data in HPC research.

Extensibility: By all means, such a paradigm shift is an ambitious goal. Thus, the new RM must be extensible and customizable to accelerate the shift and provide features that lower the barrier of entry into the community of developers supporting and extending the RM. Plugins, when designed properly, can significantly help realize this vision, without having to sacrifice the stability of the core RM software. With plugins, various RM subsystems can be replaced and extended. This can be the mechanism for individual sites to customize their RM for their particular needs. In addition, the new RM must be designed for testability. Ideally it should be possible to test new plugins or even a complete new version of the RM system within the confines of a job created by a production version. This self-hosting capability would enable regular RM regression and stress testing to be automated without dedicated test resources or special production system arrangements.

3 Design Space

Before going into the design and implementation details of NGRM, we discuss some of the key design challenges that the new RM paradigm presents. They represent the main factors that NGRM's new

concepts and software design must effectively address in realizing the vision and new capabilities described in Section 2.

3.1 Design Challenges

- *Multidimensional scale challenge:* The new paradigm demands that the RM must manage the entire computing facility as one common pool of resources. Compared to the traditional paradigm, this presents fundamentally more difficult scale challenges to the RM design, not only in the concurrency of a single workload but along several other dimensions. As concurrency increases, every RM run-time service must scale and noise must be put at bay. The number of jobs and resources that the RM must manage will drastically increase; the amount of run-time information that the RM must monitor, trace and store will grow in the scaling limit of the facility. Thus, this challenge precludes any centralized design in an attempt to gain a wider view over the resources at the facility.

- *Diverse workload challenge:* The new paradigm must recognize that different applications have different performance-limiting factors, and this imposes more complex requirements to how the RM should model the compute resources. The traditional approach of modeling resources as a collection of compute nodes will only work well when the application is compute-bound. Modern workloads have grown in their complexity, and even today, only a small fraction of modern applications is compute-bound.

- *Dynamic workload challenge:* Not only must the paradigm support disparate performance limiters across different applications, but also must it suit varying performance limiters within a single application. Our applications and their programming paradigm are becoming increasingly dynamic with different resource requirements at different phases.

- *Power challenge:* As one specific example of emerging resource types, power is becoming critical. When the computing facility becomes power-bound instead of compute-node-bound, the new paradigm must help it to schedule workloads based upon the maximum power limit at any level at the facility. Thus, the resource representation of the new RM must be generalized enough to model consumable resources like power.

- *Scheduling challenge:* As more diverse attributes of resources are factored into scheduling, more stalls can occur in the schedule. For instance, N compute nodes may sit idling simply because they do not meet the network proximity requirement for a job that requested N nodes all connected at a same lower-level switch. Thus, our design must provide alternative ways to fill the stalls to meet this challenge.

- *Backward compatibility challenge:* The new paradigm must also be able to model the traditional paradigm, as its small subset. This then provides our design with a straightforward path to backward compatibility with legacy scripts from a traditional paradigm such as Slurm.

- *Integration risk:* In the new paradigm, the RM must integrate other software essential to the next-generation computation. But with higher integration comes the risk of hard-wiring assumptions that later prove to be confining. That can force changes down the road that are inconsistent with the initial design. This motivates an extensible framework design.

- *Higher downtime costs:* The impact of downtime under the new paradigm becomes much greater: if not designed adequately, a downtime can negatively affect the availability of a large portion of the facility and/or running workloads across it. Thus, the new paradigm must be tolerant of hardware and software faults and failures with no single point of failure and must also support live software upgrades.

- *Separation-of-concerns challenge:* Many attributes of the new paradigm motivate a much higher degree of separation between the software level visible to applications and system-level software. For example, the paradigm must be able to reconstruct the user-visible software level to provide better reproducibility of simulations while not locking the system software level.

ned-review: How to reconcile conflict with security requirements? For example, an attacker might request a known-vulnerable kernel version. **jg:** Restrictions on this are explained in *Lighweight Virtualiation Model* paragraph.

- *Security challenge:* As the new paradigm increasingly motivates a highly distributed, hierarchical software design, the importance of security across and within the components becomes greater.
- *Productivity challenges:* The new paradigm must improve end-user productivity in part through tightly-integrated support for development and use of scalable code development run-time tools and research.

3.2 New Conceptual Models

In this section, we describe some of the primary conceptual models that will embody this new paradigm while addressing the multitude of design challenges. The models form the basis for the software design of NGRM.

ned-review: The discussion here doesn't really make much sense until you read section 4. I would provide a forward reference to assure your readers that the concept will be explained in more detail later on.

Unified Job Model: Traditionally, a job is simply defined to be a resource allocation, a concept too weak to support the new paradigm. Rather, we unify the traditional job notion with the notion of a resource manager instance—an independent set of resource manager services. The RM instance must be delegated the main responsibility of managing the resources allocated to the job. Then, the unified job model becomes the foundation on which to build a hierarchical, resource-management scheme to address the *multidimensional scale challenge*. In addition, an RM instance can implement compatibility mode with a particular traditional paradigm only over its own allocation, providing a straightforward path to address the *backward compatibility challenge*.

Job Hierarchy Model: To scale the new paradigm in the scaling limit of the entire computing facility, we must avoid a centralized approach: the new paradigm requires a hierarchical management scheme with a well-balanced, multi-level delegation structure. For this purpose, we use a tree-based job hierarchy model that has many proven advantages for extreme scalability. In this model, a job is only required to manage its children jobs, which would be only a small fraction of the total number of jobs that are run across the entire computing facility. Further, several guiding principles throughout the job hierarchy strike a balance between the management responsibility of a parent job and delegation/empowerment of a child job:

- *Parent bounding rule:* the parent job grants and confines the resource allocation of all of its children.
- *Child empowerment rule:* within the bound set by the parent, the child job is delegated the ownership of the allocation and becomes solely responsible for most efficient uses of the resources.
- *Parental consent rule:* the child job must ask its parent job when it wants to grow or shrink the resource allocation, and it is up to the parent to grant the request.

In general, these rules enforce the first principle of the new paradigm: imposing highly complex resource bounds to guarantee the highest operational efficiency at any level across the computing facility, while enabling most efficient execution and scheduling of the workloads within these bounds. At the same time, this model is the most fundamental design concept, which forms the basis to address many of the design challenges including the *multidimensional scale*, *dynamic workload*, *power*, and *scheduling challenges*.

Generalized Resource Model: In the traditional paradigm, compute resources are modeled primarily as a collection of compute nodes, a simplistic perspective ill-suited for the new paradigm. Today's applications are diverse with disparate limiting performance factors beyond floating point computation. Further, computing centers are increasingly concerned about managing new resource types such as power and shared persistent storage. The generalized resource model is our concept to represent various resource types and their relationships that can impact how well applications perform and the computing facility operates. Our generalized resource model also includes a unified

resource specification and description language. Speaking the same resource description language for request specification provides transparency and fine-grained expressibility. Our generalized resource model addresses not only the *diverse workload* and *power challenges*, but the *scheduling challenge*. More specifically, the unified language approach allows users to express their resource requests more flexibly, e.g., using ranges or boolean expressions instead of hard amounts to allow requests to be fulfilled from several equivalent resource types. This makes the scheduling granularity of jobs finer and more malleable.

Resource Allocation Elasticity Model: As our applications and their programming models are becoming increasingly dynamic, the new paradigm must support an elasticity model where an existing resource allocation can grow and shrink, depending on the current needs of applications and/or the computing facility. We support the elasticity model within our job hierarchy framework above: a child job sends a grow or shrink request to its parent, which can go up the job hierarchy until all requisite constraints are known for this request. Also, combining this with the generalized resource model, the elasticity can be expressed for any resource including power consumption. Our elasticity model addresses not only the *dynamic workload* and *power challenges*, but also *scheduling challenge*. When a significant schedule stall is created with no small jobs to backfill, some of the currently running jobs can grow into these stalled resources and possibly complete sooner.

Common Scalable Communication Infrastructure Model: Our scalability strategy with respect to a large number of compute nodes is to provide a common scalable communication framework within each job. When a job is created, a secure, scalable overlay network with common communication service is established across its allocated nodes. Except for the root-level job, the existing communication session of the parent job assists the child job with rapid creation of its own session. A communication session is only aware of its parent and child and passes the limited set of control information through this communication channel. Thus, this model enables highly scalable communication within a job, while limiting communications between jobs, addressing both the *multidimensional scale* and *security challenges*. Further, this backbone per-job communication network supports many well-known bootstrap interfaces for distributed programs including many MPI implementations as well as run-time tools, and thus in part addresses the *productivity challenges*.

Self-Hosting Model: We use a self-hosting model to instantiate a new RM instance: the parent is capable of launching a standalone copy of itself as a child job, but possibly with different plugins. This makes it easier for developers or a quality assurance team to test new RM versions, helping addressing the *higher downtime costs* challenge. Further, self-hosting with new and experimental plugins encourages experimentation and facilitates research activities within a production instance, addressing the *productivity challenges*, too.

Lightweight Virtualization Model: The lightweight virtualization model is our response to the *higher downtime costs*, *separation-of-concerns* and *security challenges*. Full-fledged virtualization techniques like Xen and Kernel-based Virtual Machine (KVM) have many advantages for these design challenges, but that approach has proven to be ineffective for HPC due in large part its high overhead [43]. Instead, our virtualization strategy exploits Linux kernel-enforced resource management and isolation mechanisms to launch applications in containers with virtually no impact on performance [53]. Within a container, private file system namespaces allow the system and applications to have divergent file system views, and to access file systems with different constraints. For example, the RM and other system software might be launched from one version of the root file system with no access restrictions, while an application might be launched from another with *setuid* capability disabled. The decoupling and isolation will be the key to the much desired ability to upgrade the software levels in the machine's root file system without affecting any of the libraries that an application might be using, and vice-versa. Similarly, one can upgrade the system OS image at idle points between jobs, with user-level software remaining unchanged, or vice versa. The separation of concerns gives more flexibility to the organization in determining software update

chris-review: Is making OS-level virtualization part of the design going to limit NGRM's portability e.g. to exascale systems that may not run Linux? **jg:** Portability to non-Linux systems was not a goal of the project, however OS-level virtualization is available in FreeBSD jails, Solaris zones, and AIX WPARS. It will likely be implemented through NGRM plugins allowing for some portability or replacement with other container methods such as full virtualization, chroot jails, etc.

policy and in fact could allow users or code development teams to control the software levels affecting their application, independent of other applications.

The lightweight container approach will be used to tightly control the access that applications have to system resources. For example, it is possible to run the job in its own network namespace such that direct access to the system management network is unavailable, or limited by container-specific firewall rules. A job could be further isolated on its own virtual private network. Containers with private file system namespaces can limit visibility of file systems and other system resources to jobs based on the site policy, significantly addressing the *security challenge*.

4 Next Generation Resource Manager

ned-review: With this sentence it becomes clear that the self-hosting model is a core concept in the design of NGRM, and not just a nifty tool for NGRM developers. The earlier sections on self-hosting should emphasize this aspect.

To realize our conceptual models, NGRM uses a divide-and-conquer algorithm based on *job recursion*. As our *unified job model* dictates, we design NGRM so that every job is actually a full implementation of the NGRM system, and therefore users can submit new jobs to an existing job to access the full power of NGRM for managing resources assigned to it. The existing job and the new ones then form the parent-children relationship according to our *job hierarchy model*. In principle, our design enables this recursion to occur indefinitely so that the resources in the entire HPC center are *divided* and *conquered* up to any level suited well for the specific needs of the center and/or workloads. To simplify and generalize this scheme, we must first represent a wide range of compute resources in a tree-based hierarchy.

Figure 1 illustrates our representation for a modern HPC center. As shown in this figure, the root of the tree (Center) represents the entire center-wide resources. At the next level, these resources are refined to be some number of clusters (Cluster), the maximum power budget (Power), and software licenses (Licenses). Power and Licenses illustrate that this scheme can easily represent a wide range of resource types as well as their relationships—i.e., *our generalized resource model*. Expanding any of the second-level nodes, one can further refine its resource distribution. In this case, zooming on Cluster 2 refines its resources into some cluster-local file system (Storage) and an interconnect domain (Core Switch). Next, the Core SW resource has some number of Rack resources associated with it, and an arbitrary Rack has some number of compute nodes and its maximum Power budget. And similiar resource refinements and relationships hold true at the node level as well.

We now walk through mapping our job recursion algorithm to this hierarchical resource representation. When NGRM initializes, only a single *root* or *bootstrap* job exists. Analogous to the UNIX `init` process, the root job is the only job that does not have a parent job—i.e. it solely serves as the root of the tree-based job heirarchy. Instead, the root job gets *global* information about entire compute resources, users, and configuration from a scalable, peristent database that serves as our configuration repository. To show this concept, Figure 2(a) overlays the root job (`job 0`) on the entire resource tree.

Users of the root job then submit all new job requests to it, which are scheduled by the scheduler of the root job. Each child of the root job is itself a full NGRM instance according to our *unified job model*, thus users of a child job can further submit new job requests to the child job. The child may have different, customized scheduler and environment in accordance with our *self-hosting model*. As shown in Figure 2(b) and (c), we have a single job (`job 0.x`) managing the entire resources in a single cluster, and its child job (`job 0.x.y`) running on a subset of that cluster's resources. The root job and its progeny all have the following features:

- A job *owner* and users in its access list who are able to run within the current job and/or submit new jobs;
- A resource manager component configured with the list of resources allocated to the job, their topology and other information;
- A scheduler that accepts and schedules new jobs;
- A job database that records new, running, and completed child jobs;

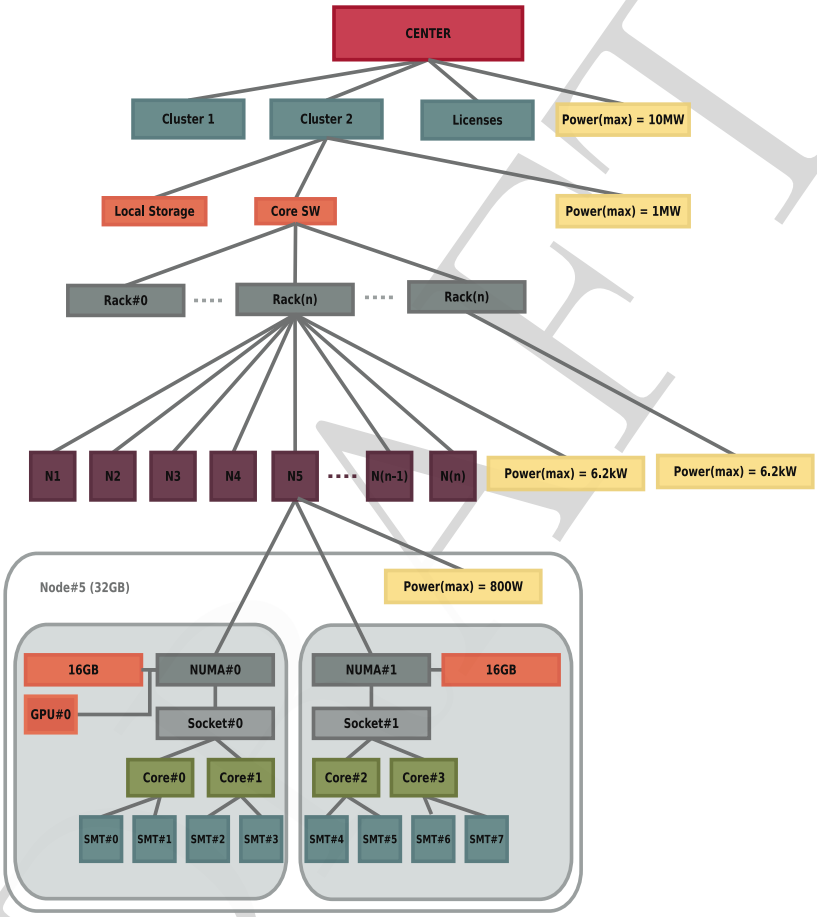

Figure 1: Hierarchical View of Resources in an HPC Center

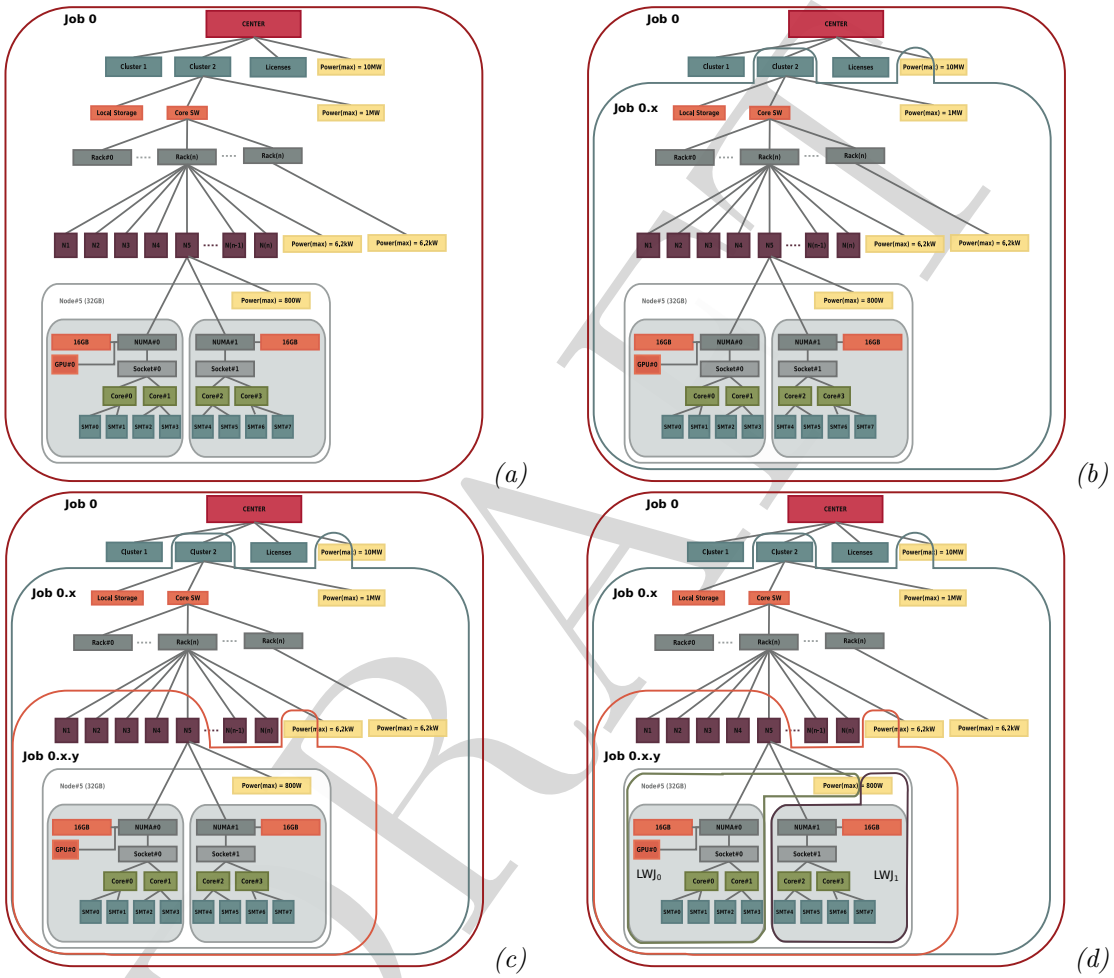

Figure 2: An example job hierarchy shown on top of a resource hierarchy. *(a)* Job 0 spans all resources, *(b)* job 0.x runs across all of Cluster 2, *(c)* job 0.x.y runs across nodes N1–5, and finally *(d)* two lightweight jobs are instantiated on N5.

- An "isolated" communications framework with *gateway* functionality for relaying messages to the parent—i.e., following our *common scalable communication infrastructure model*;
- A distributed and featureful run-time environment capable of launching or assisting the launch of parallel applications and distributed tools;
- An "interactive" node on which batch scripts and/or user logins are contained;
- A resolvable domain name based on a unique job id;
- A local monitoring domain;

Following our *self-hosting model*, we design many of these components via plugins or with plugin capabilities. We expect that several features will be user-selectable during job submission. For example, the root job may have a complex, distributed scheduler, whereas users may want to choose between FIFO, backfill, or other simpler schedulers depending on the in-job workload.

By default, only the owner of a job may submit new jobs or access the run-time services of a running job. While the ownership of a job should not be changed, it will be possible for users to add other users to the access list within their job, thus "inviting" the submission of new jobs by others. The requirement for this feature is clear when considering the root job, from which all other jobs are spawned. This job will be owned by a privileged user such as root, yet the system administrators will obviously want to open up this root job for access to all users who should be able to run jobs in the center.

ned-review: If a parent dies is the child "reparented" so it's data can still be reaped (i.e. becomes an orphan in the UNIX process model)

When a job terminates, either due to a time limit or the work submitted for the job has completed, the job releases most resources back to the parent job. The control functions of the job remain and wait for an asynchronous *reap* operation from the parent. When the job is reaped, the parent job reads in all data from the child's local job including its resource database. The parent then instantiates that data in its own databases. In this way, global information about jobs percolates back up through the job hierarchy, eventually back to the root job. The root job periodically updates the global, persistent databases.

In our job recursion, the so-called *base case* is a single job in which a single parallel application is invoked. In this case, a fully functional child job is actually not needed, and we introduce the concept of a *lightweight job* (LWJ) for efficient resource use. An LWJ is submitted and runs on the resources of the local job, but does not result in a new invocation of all the features of a full job. Where NGRM jobs are like processes in a UNIX process tree, LWJs are like threads running within a process. The final case in Figure 2 is an example of two LWJs that divide the resources of a single node.

LWJs have access to a feature rich, distributed, run-time environment with a shared key-value store, advanced placement services, and a plugin interface that allows extension of these services for unique requirements. This environment will enable the quick deployment of advanced run-time features such as fast parallel launch of MPI applications and seamless tool integration. Since LWJs use the same interface to job management as full jobs, their existence, assigned resources, duration, etc. will be recorded in the local job database for posterity. To run an LWJ, a user must be the owner of the current job or on the run-time access list for the local job.

4.1 Comparison with Traditional Job Schedulers / Resource Managers

In this section, we draw comparisons between NGRM's constructs and traditional terms to show that the traditional paradigm can easily be reduced to our new paradigm.

job In traditional terms, a user submits a request to a batch scheduler running on a particular cluster for an allocation of computing resources. This request is added to a queue of other such requests. When the scheduler grants a request, a job is begun and a set of resources are allocated to the job. At that point, one or more processes are launched to do the work of the job. The traditional job will be recognized as the base case of the NGRM job, although without the unifed RM instance and hierarchy. In NGRM, the request is submitted not to a batch scheduler managing one cluster,

but to another job (perhaps the root job) which manages a subset of resources. A scheduler running in that job allocates resources, and launches work as another job.

job step Depending on the system, processes are launched by a remote launcher such as *mpirun* or *srun*. In Slurm terms, each set of tasks so launched are known as a job step. There can be one or more job steps active within a job throughout the life of a job. These can run sequentially or in parallel, and can either run on dedicated or shared resources. In Slurm, they are processed within the job by a dedicated FIFO-based job step scheduler. By first approximation, the job step maps to the NGRM LWJ. Job scripts that invoke a series of job steps will translate to a series of LWJs in an NGRM job, processed by the regular NGRM scheduler and the same in-job resource management machinery that would be used to launch regular jobs. Because each job is endowed with the capabilities of the full system, there is immense flexibility in how a job structures its work internally. It can select a scheduler plugin appropriate to the task (or provide its own), express complex resource requirements and job interdependencies, and run work as LWJ's or full isolated jobs.

reservations Historically, batch schedulers provide a means to reserve a set of resources for exclusive use by a user or group of users. These reservations are typically created in advance and are offered as Dedidcated Application Times (DATs) for users' exclusive use. Often this process involves some manual setup and special announcements. In NGRM, job reservations will take the form of regular job (with a future start time) under which user jobs will run. As with all NGRM jobs, the owner of the DAT job (the DAT coordinator) is empowered to manage user access to the DAT, instigate custom monitoring, custom scheduling, arrange for access to dedicated file systems, etc. Thus in the new paradigm, the DAT is just another job that should not require special handling.

job accounting In Slurm, the history of a job includes a detailed accounting of each job step including the resources used and the duration of the job step. Job and job step accounting statistics are commonly saved to a database. In NGRM, the LWJ accounting information and other data is saved to the in-job job database. Upon termination, the job's data is reaped by its parent as described above, ultimately landing in the persistent copy at the root job level.

4.2 Project Organization and Thrust Areas

To hasten the design, development, and delivery of the initial version of NGRM we define four relatively independent thrust areas. Each thrust area will carry out the vision and high-level design articulated above with a focus on a particular subsystem or group of subsystems that have a natural coupling. The overall design of NGRM will evolve and require the whole team to weigh in on important changes, and the design of interfaces between subsystems will require collaboration between thrusts, but the work in each thrust can in large part progress independently, building on the strength of the other thrusts.

Deliverables in each of the thrusts will be structured to leverage the framework approach to get the system up and running early with simple plugins that are enhanced later. Early prototypes will be used to accelerate the design of subsystem interfaces and to obtain feedback from stakeholders and domain experts as soon as possible. Off-the-shelf components, external collaborations, and novel ideas from the research and development communities will be leveraged where possible. Thrust areas will adhere to a common set of project development practices and standards (see Appendix **??**).

Each of the thrust areas is briefly introduced below, then described in more detail in the sections that follow.

Communication Framework The Communications Framework thrust will realize our *common scalable communication infrastructure model*. Building upon mature and portable Internet protocols and services, the comms framework enables rich, scalable communications services within a job and more limited communication between jobs using the job's *control node* as gateway. A *comms toolkit*

provides a security model, tools for encoding and decoding messages, and tools for transporting messages between nodes using various messaging patterns including PUB-SUB (with multicast), RPC, and streaming. A *comms message broker* tracks node liveness, provides support for building fault-tolerant services within the job, and provides a multicast *scheduling trigger* used to synchronize system overhead within the job to reduce noise. A persistent *reduction network* provides the structure needed to build tools and services that have in-situ reduction capability, rooted at the control node.

Resource Management The Resource Management thrust encompasses the configuration, scheduling, and tracking of resources, jobs, and users in the NGRM system. To meet our new paradigm, this thrust must focus on making the RM subsystem generalized, flexible, and extensible. To that end, the NGRM RM system will develop a domain-specific *resource description language* which will be used to describe the configuration, current state, and topological organization of resources, while also being capable of describing *requests* for those resources. The RM thrust will also develop a set of scalable, generalized databases or repositories to store information about resources, jobs, and users – one set as a global, persistent datastore, and another short-lived set of databases at each job level. At the global, persistent level a *resource repository* will act as s top-level configuration database for resources. A read-write copy of this repository will be available within each job instance (at this level termed the *resource database*), such that any job may modify resource attributes as if it were a full instance of NGRM. The global *job repository* will be responsible for storing historical job information, including, but not limited to, resources assigned to the job and all its children jobs, job provenance records such as software levels and job environment, and any RAS events or other monitoring data associated with the job run. Within each job, a *job database* temporarily serves as the job repository until job destruction, and is also the interface to job submission, termination, and alteration activities. Finally, the global *user repository* is used as a source for user-specific information such as UID, and also possibly user preferences, roles, and so on. Within each job, the local *user database* can be used to control permissions for submission of new job requests, launching of lightweight jobs, and other access control activities.

The scheduler is also a critical component of the RM thrust area, as it is responsible for computing a job execution schedule based on available resources, constraints posed by users in their job request, and policy enforced by resource owners. The concept of the resource description language will be extended to jobs with the introduction of a *job description language* which is flexible enough to describe complex job resource and time requirements. The scheduler interface to NGRM will be via a powerful plugin subsystem, which will allow alternate job schedulers to be swapped in, possibly on-demand, as jobs are launched in NGRM. (For instance, the root job may use an advanced fairshare scheduler, while a job launched by a researcher for a parameter study may incorporate a simpler scheduler tuned for high throughput). The plugin interface for the scheduler should ease the development of scheduler algorithms for third parties, and shall not require that the scheduler software be built with access to NGRM source code. The scheduler interface will offer services that allow users to query when their job might run, for example by exporting the currently computed schedule as a diagram, or by estimating a start time for a given job or set of jobs.

Monitoring Resource managers must monitor resource health to avoid scheduling work on broken hardware. NGRM provides a comprehensive monitoring environment including a *plugin framework* with data reduction capability, synchronization of monitoring overhead across the job, and the ability to tune the monitoring period and change the plugin stack on a per-job basis. A *fault notification system* enables system software, runtimes, and applications to share fault information as a basis for building fault-tolerant workloads and for recording interesting events within the job record. NGRM monitoring interfaces with an *external log database* intended to support post-mortem analysis, and to an *external enterprise monitoring system* such as might be used in a site operations center.

Workload Runtime and Placement The NGRM runtime launches the user workload such that it is confined to allocated resources using a set of *confinement plugins* and is provided a set of services intended to support application runtimes and tools operating at extreme scale. These services include a *distributed key value store* which facilitates disseminating bootstrap information

to distributed tools, a *unified software bootstrap interface* which provides common interfaces to a variety of tools and runtimes, and a *job function synchronization* service which assists with the placement of tool processes co-located with application processes. The effort required to build a distributed tool on top of these services and the comms layer will be greatly reduced from today, enabling proof-of-concept research tools or even one-off tools to solve a particular problem to be created in a short amount of time.

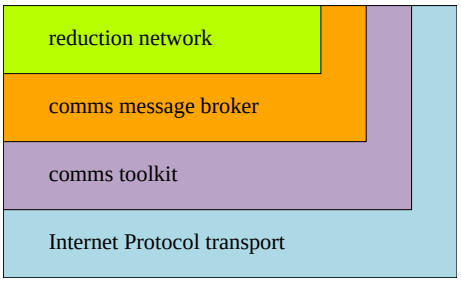

Figure 3: Communication Framework Layers

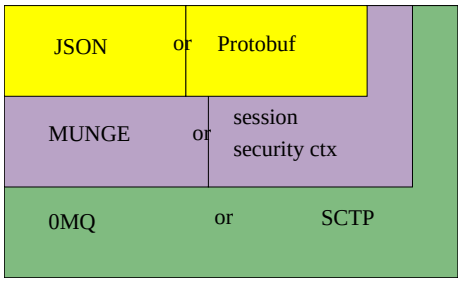

Figure 4: Comms Toolkit Layers

5 Communication Framework

The NGRM architecture is hierarchical and recursive (A.3, req. 4.1). A root instance of NGRM contains all the idle resources. A job spawned by the root instance contains its own NGRM instance, which may in turn spawn jobs, ad infinitum. When a job terminates, the job's instance terminates and resources return to the parent instance.

The communication framework supports this architecture by establishing a *comms session*[1] to contain each NGRM instance and provide a foundation for the distributed components of NGRM to be built upon. The framework enables secure, scalable communication within a comms session, limits communication between sessions, and allows new comms sessions to be created, resized, destroyed, and monitored by existing ones in a parent-child relationship.

ned-review: A worked example would be useful here. **jg:** for now, section 7 might be helpful.

A comms session is only "aware" of its parent and immediate offspring. Any communication between siblings would have to be orchestrated by the parent. This sandboxing arrangement should encourage the higher level components of NGRM to be built so they can operate at any level of recursion, thus improving their testability and making development of replacement components easier.

It should be noted that although NGRM obtains scalability from the job hierarchy, the *idle* root comms session must contain all the systems and resources managed by NGRM and therefore must handle the full 100,000 node scalability target (A.1).

The communication framework consists of four main layers: IP protocols and services, comms toolkit, comms message broker, and reduction network, shown in Figure 3. Layering is not rigid; that is, higher level NGRM components can use any of the layers directly as appropriate[2].

5.1 Internet Protocols and Services

The NGRM comms framework is layered upon Internet Protocol (IP)[3] and presumes complete IP level unicast and multicast connectivity across participating systems, so that any collection of nodes can be wired up in a comms session without the need to re-implement the equivalent of IP routing within the framework.[4] The addressing, routing, and subnetting of this IP network is beyond of scope of NGRM, except that its design should introduce no single points of failure (A.3, req. 1.2) and should avoid performance bottlenecks which would unnecessarily constrain the resource manager's node selection options.

The comms framework should support communication over multiple (fully-routed) network planes, for example using either a management ethernet or IP-over-IB or both according to the perfor-

[1] *Comms* is a shorthand for NGRM Communications Framework and is not related to MPI communicators.

[2] At this stage in the design of NGRM, we do not wish to overly constrain the solution space for the other components that will use the comms framework. For the same reason, distributed object oriented frameworks, such as those derived from CORBA, were rejected as too confining. This stance can be re-evaluated as the other components are designed.

[3] The low-latency and low-overhead bulk transfer properties of RDMA communications such as provided by Common Communication Interface (CCI) were considered and rejected as unnecessary for NGRM.

[4] Building a reliable 100K node IP internetwork is a solved problem. Building a hardened overlay network with similar properties is difficult and would limit our ability to leverage other software built on IP.

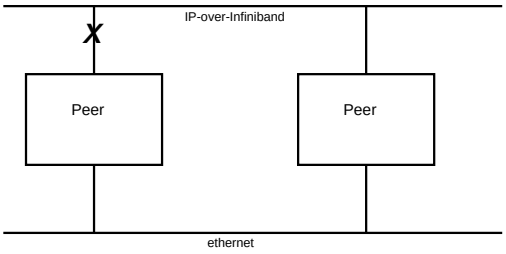

Figure 5: SCTP supports transparent multi-homing of one socket over multiple network planes, for example using both Infiniband and Ethernet in a cluster. If one network drops a packet, it is retransmitted on the other, which with proper tuning, improves both availability and responsiveness compared with TCP single-homed sockets.

mance/reliability requirements of the particular application.

Dynamic unicast IP address allocation must be available to support dynamically created virtual nodes (Linux containers launched with virtual network interfaces). In fact, it is worth noting here that we expect to make heavy use of virtual nodes in NGRM for our internal services that should not be co-located with computation. Similarly, dynamic multicast address allocation must be available to support private multicast groups within dynamically created comms sessions. These requirements can be addressed by existing technology such as DHCP [19] and MADCAP [30].

A private DNS [38] is used by the comms framework to map a hierarchical namespace to the comms session hierarchy. The root comms session has the root domain name, e.g. "`NGRM.`", and the root server contains address records for all hosts in the domain, e.g. "`n1.NGRM`", "`n2.NGRM`",..., "`n99999.NGRM`". Comms sessions spawned by the root session get their own sub-domain, e.g. "`s1.NGRM`", "`s2.NGRM`", "`s3.NGRM`", and contain address records for the nodes assigned to them, e.g. "`n1.s1.NGRM`", "`n2.s1.NGRM`", "`n3.s1.NGRM`". Sub-domains are created for each level of comms session recursion. Each session runs a set of DNS servers for its domain. When a node joins a new session, its DNS resolver is reconfigured to use the session DNS servers and to search the session's DNS domain first, thus each level of session overlays a new set of names over the root session's that provides job-centric naming uniformity. DNS SRV records [28] provide a rudimentary service location brokerage within the session. Well understood techniques for DNS fault tolerance, caching, and dynamic reconfiguration are leveraged to scale performance in large sessions such as the root session.

ned-review: Would we point resolvers outside NGRM at the NGRM internal DNS? **jg:** I envision NGRM operating in a private network. In that case, resolving private addresses from outside doesn't seem useful. However, we may want to consider whether we need some way to map external addresses with names to internal ones.

A comms session could optionally be spawned inside a virtual private network (A.4, UC21) such that IP communication is limited to within the comms session. IPsec [35] with a pre-shared session key could be used to provide session integrity and privacy at the IP layer if desired.

5.2 Comms Toolkit

The framework provides a toolkit for sending and receiving protocol messages privately and securely within a session, with the goals of providing a robust foundation for NGRM components and promoting rapid development, code reuse, and interoperability. The toolkit includes messaging libraries, protocol encoding/decoding libraries, and security options. Toolkit pieces can be mixed and matched according to application requirements. We may cull the comms toolkit as we learn more about the pieces while prototyping other NGRM components.

Two messaging approaches of interest are ØMQ [31] and SCTP [48]. ØMQ provides the ability to manipulate opaque, multipart messages, and carry them across various transports, including TCP and Pragmatic General Multicast (PGM) [46], a reliable multicast transport protocol, using a socket-like API. ØMQ sockets can exchange messages using patterns including REQ-REP (RPC), PUB-SUB, and PUSH-PULL (message stream). ØMQ can be used to build applications or custom message brokers. Complex message routing topologies such as tree-based overlay networks (Figure 8) can be built from simple components. ØMQ has a large number of language bindings.

SCTP is an IETF-standardized, message-oriented transport developed in the telephony world with an implementation in the Linux kernel. It offers multi-streaming, the bundling of *streams*

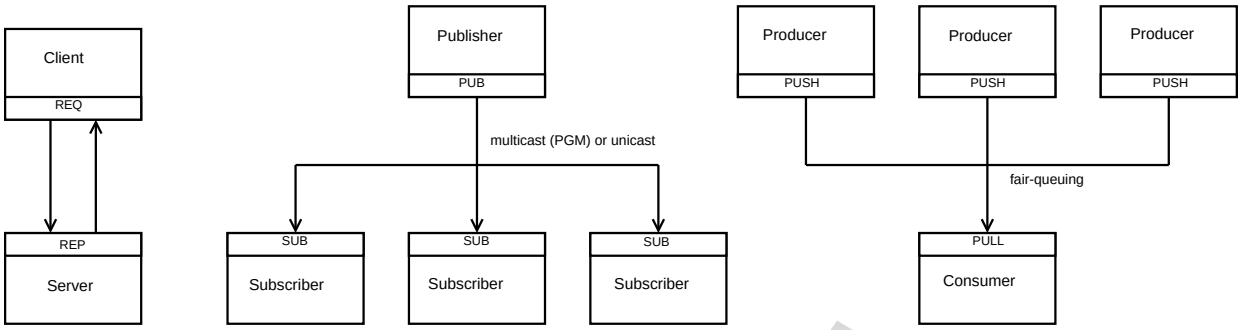

Figure 6: ØMQ provides a sockets-like API that supports several messaging patterns including REQ-REP, PUB-SUB, and PUSH-PULL. These basic patterns are complemented by other patterns used when building distributed message brokers. For example, XPUB-XSUB handles subscription forwarding to optimize multi-level broker based publish-subscribe networks.

in one *association* (connection). Individual streams can be configured for different ordering and reliability semantics. SCTP supports multi-homing for reliability and congestion avoidance, as shown in Figure 5, and can transparently generate and check an HMAC for messages using a pre-shared key to implement message integrity. Unlike ØMQ, SCTP is connection-based and does not implement a standardized reliable multicast mode.

Two widely used methods of encoding data in messages are JSON [17] and Protocol Buffers [27]. JSON is a self-describing format that supports protocol evolution without recompiling endpoints. It has many language bindings but it is also space-inefficient and slow. Protocol Buffers is a compiled format that supports only limited protocol evolution without recompilation. It has relatively fewer language bindings than JSON but is space-efficient and fast. Depending on the application either may be appropriate.

Message integrity and privacy can be obtained using either a session security context or via MUNGE [20]. Each comms session is allocated a *session key* by its parent which is used for establishing a shared security context for messages exchanged within the comms session. The shared security context allows communicating entities to have integrity and privacy (from children, siblings, and their children) without the overhead of a key exchange for each pair of communicating endpoints. This is especially useful for non point-to-point comms patterns such as PUB-SUB. The parent retains state about its offspring including their session keys. Children forget their parent's key; thus as comms sessions recurse, parents get privacy from children but not the reverse. An application acting as a gateway between parent and child would use the child's session key as it is known by both parent and child.

If the sender of a message needs to be authenticated, or if messages must be kept private from other users within the session or its anscestors, messages can be enclosed as payload in a MUNGE [20] credential. In order for this to work, NGRM's comms framework must operate within a single MUNGE security realm, which implies a single administrative domain with consistent user and group identities.

chris-review: Why should comms session forget parent's key? Is a new session formed for each level of recursion or is there a single thing? Document seems to imply the latter. **jg:** The high level concept is that the portion of the CMB running on each node is associated with one session at a time. It forgets the parent context when it joins a child session. However the parent retains the context of its children so it can interact directly with a child node if necessary, for example to reclaim resources from a hung session.

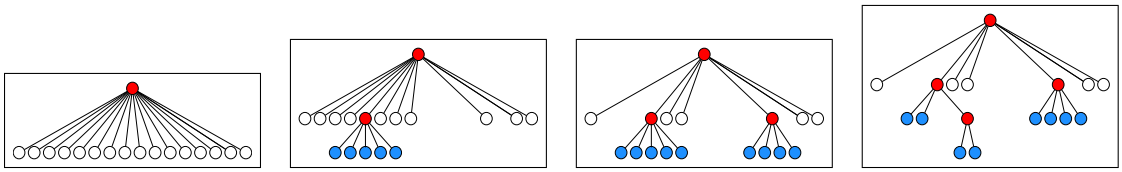

Figure 7: CMB Architecture: comms sessions are formed hierarchically. Session control nodes (red) are the interface between a session and its parent. An idle root session is shown on the left; one and two sessions running work (blue) under the root are shown in the middle, and one of those has spawned a child on the right. Note that although control nodes are depicted as being allocated from real nodes, depending on the session size, they may be created on demand as virtual nodes.

5.3 Comms Message Broker

Within a comms session, a distributed comms message broker (CMB) is established to provide basic session services. The CMB is responsible for launching new sessions, managing session membership, detecting and adapting to session node failures, providing a basic event messaging system, and starting other NGRM components.

Architecture The CMB is a distributed service with nodes interconnected in a tree-based topology. A distinguished *control node* serves as the heart of the CMB and the root of the tree. The control node is distinguished because it alone communicates with the parent session, and it holds the master copy of the session state. The hierarchical relationship between comms sessions with the control node acting as gateway is depicted in Figure 7. The details of any internal tree-based interconnections are not depicted in the figure, and are a future design activity.

chris-review: Is the root of the tree in the CMB design a single point of failure? **jg:** This is one of the issues that will need to be addressed in the fault tolerance design for the CMB, a future work item.

Session State The CMB implements a simple key-value store to manage the internal state enumerated in Table 1. The session state for the largest session is small enough to easily fit in memory. The master state for a session lives on the session control node. Slave caches on other nodes in the session are loosely consistent with the master copy; that is, reads may utilize a local or peer cache, which may be slightly out of date relative to the master copy, while writes are through to the control node. Each write on the control node updates the key's generation number, its value, and triggers a state update event which can be used to update caches and synchronize other software using the state. If the control node crashes, state can be recovered from one of the slave caches.[5]

Event Messaging The CMB implements a session-wide event messaging service. Clients of the CMB on any node can publish a $(tag, message)$ event tuple. Other clients can subscribe to events by tag. The CMB ensures that event messages are routed internally from publishers to subscribers. The event service is reliable, and for events originating on the same node, sequenced for in-order delivery. Events are not queued for late subscribers. There is a special `event.sched.trigger` event sent out periodically to synchronize the CMB's internal functions (and those of any other subscribers to the event) across the session with the goal of minimizing disruption to bulk-synchronous workloads running within the session.

FIXME: The per-job scheduling trigger probably has insufficient emphasis here. It is an important benefit that reduces our noise footprint and could be generally useful to other system/tools software written to run within a job.

Session Membership The CMB maintains the current *nodeset* as part of the session state. The CMB arranges for the nodeset to be mirrored in private DNS servers serving up the session's subdomain. The nodeset may grow or shrink in response to higher level software allocating/freeing nodes from the parent, or creating/destroying virtual nodes within the session. Nodeset updates will be accompanied by internal topology updates, provided by the software making the nodeset change or by the CMB itself depending on the situation. State update events will be published for the nodeset

[5]The fault tolerance strategy here is to be designed. One approach is that the parent can determine if a control has stop functioning (see *Liveness Monitoring* below) and intervene to establish a new one with restored state from slave caches.

Name	Description
cmb.cred = *key*	My session key.
cmb.fqdn = *name*	Fully qualified domain name for the session.
cmb.nodeset = *nodelist*	List of my nodes.
cmb.addrs.*node* = *addrs*	List of addresses for *node*, for forming DNS address records.
cmb.topology.up.*node* = *nodelist*	Upward peers for *node* in CMB topology.
cmb.topology.dn.*node* = *nodelist*	Downward peers for *node* in CMB topology.
cmb.alive.*node* = *yes*|*no*	Liveness for *node*.
cmb.alloc.*node* = *yes*|*no*	Allocation status for *node*.
cmb.attrs.*node* = *attrlist*	Role attributes assigned to *node*, e.g. "dns" and "control".
cmb.subscribe.*key* = *nodelist*	List of nodes subscribed to *key*.
cmb.exec = *cmdline*	Executable to bootstrap on each node.
cmb.child.sessions = *sessions*	List of active child sessions.
cmb.child.*session*.cred = *key*	Child session key.
cmb.child.*session*.control = *nodelist*	Control node(s) for *session*.
cmb.child.*session*.dns = *nodelist*	DNS server nodes for *session*, for forming DNS NS records.

Table 1: A small amount of data comprises the comms session state, which is stored in a simple key-value store replicated across the session.

and topology changes. While nodes are allocated to a session, they remain in the parent nodeset, tagged as *allocated*. They forget the parent's session state and key. When nodes are freed back to the parent, the parent CMB, having subscribed to the child's nodeset update events, contacts the freed node (using the child session key) and brings it back online in the parent sesssion.

Liveness Monitoring The CMB maintains the *liveness* of its nodes as part of the session state. Liveness is assessed by forcing member nodes to communicate with the CMB at minimum intervals, synchronized by the aforementioned trigger event. If the CMB has not heard from a node for some number of trigger periods, it is marked down. If it finally is heard from, it is marked *up*. In some cases the CMB control node may adjust its internal topology to account for such changes. As described above, session state changes trigger state change events. Higher level software wishing to react to node liveness changes can subscribe to state change events. The parent continues to track liveness of nodes allocated to a child by subscribing to liveness updates via the child's control node. A trivial utility that asks the CMB for a list of down nodes in the current session and all of its progeny can be written that works equally well at any level of recursion, even at the level of the root session.

Node Bootstrap A cold started node (or restarted CMB daemon) joins the root session, obtaining the root session key and the identity of a peer to copy the session state from out of band in a secure manner. If the CMB is cold starting after crashing while assigned to a session other than the root, the CMB of the root and subsequent owning sessions re-add the node from child to child until there are no children left or it is evicted according to the policy of an owning session at any level of hierarchy.

Executable Bootstrap In order to bootstrap other NGRM components, the CMB daemon, upon joining a new session, launches a single process on each node, determined by the `cmb.exec` state variable. This process is terminated when the node alters its session membership to join a child session or return to a parent session. If the process terminates early, an event is generated.

Session Creation The CMB is responsible for the creation of child comms sessions. A child session is created by building the child session state, updating the current (parent) session state, then sending the child control node(s) the full session state, and the rest of the nodes just the new session key and sufficient information to wire up to their peers in the internal toplogy. The DNS

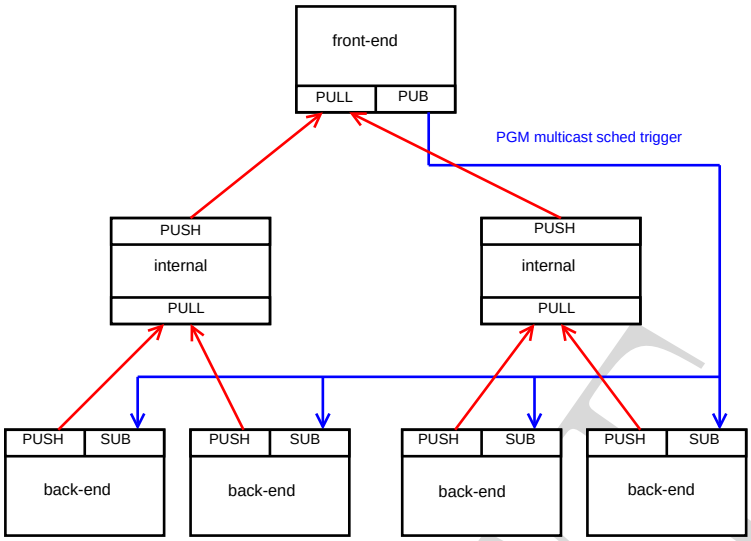

Figure 8: Reduction network speculatively implemented with ØMQ. A front-end node communicates with back-end nodes using PGM multicast in the *downstream* direction. *Upstream* traffic makes its way from back-end nodes, through internal nodes that perform data reduction on messages, and on to the front-end.

is updated in the parent to delegate authority for the new subdomain to the child's DNS servers. DNS servers are bootstrapped in the child, and the resolver is updated on member nodes to reflect the new session.

Session Destruction The CMB destroys a child comms session by sending a message to the child's control node requesting that a shutdown event be sent out session wide. As nodes leave the child nodeset, the parent reclaims them as described above in the Session Membership paragraph. If something goes wrong, the parent can short-circuit the "clean shutdown" and actively reclaim nodes as above, as though they had already left the child nodeset. The DNS is updated in the parent to remove references to the session's subdomain.

5.4 Reduction Network

The Tree-Based Overlay Network (TBŌN), exemplified by MRNet [44], is regarded as a useful communications substrate for scaling distributed debuggers and similar tools. The NGRM reduction network offers a similar capability within a comms session for general use by distributed NGRM components, for example monitoring and stdout/stderr capture. While MRNet is focused on portable tools that instantiate their own TBŌN with a custom topology for exclusive use of the tool and tear it down when the tool exits, the NGRM reduction network topology tracks that of the CMB, is persistent for the life of the comms session, and can be shared among components. It shares the elasticity and fault-tolerance properties of the CMB. Data passed over the reduction network "counts" towards CMB liveness tracking. The reduction network will obtain privacy and integrity using the comms session security context. Message handling could be accomplished with ØMQ (Figure 8) or by a combination of ØMQ and SCTP.

Topology The reduction network topology tracks the CMB topology. Its front-end is rooted at the control node. Its back-ends span every node in the comms session, including those running internal and front-end processes. The location of internal processes will be dependent on the design of the CMB. Although the topology of the reduction network must have the elasticity and fault tolerance properties of the comms session, we anticipate that for a session in stasis, the topology will be fixed. For example, branching factor and depth will not dynamically adjust for performance. However, it

may be possible to set tunable parameters for the job that would affect the initial CMB toplogy and thus that of the reduction network.

Downstream communiciation The reduction network utilizes IP level multicast (e.g. PGM) to send data from the front-end to the back-ends. As with event messages, downstream messages are a $(tag, message)$ tuple with the tag used to distinguish different services sharing the reduction network and to implement application-specific addressing, for example to address a subset of back-end processes.

jg: Is there a case for unicast store-and-forward like is used in MRNet? Why did they choose to do it that way? Reliable multicast (PGM) seems better as long as the volume of data remains low.

Upstream communication Communication from the back-ends to the front-end is the main function of the reduction network. It is unicast-based. Scalability is obtained by reductions that are performed by internal processes, for example aggregating duplicate messages, forwarding a weighted average of discrete samples, or simply concatenating messages. There may be any number of levels of internal processes, with each internal process operating on raw data or data that has already been reduced by a previous level. As with downstream communication, upstream messages are a $(tag, message)$ tuple with the tag used to distinguish different services sharing the reduction network.

chris-review: Be careful that user contributed traffic e.g. stdio doesn't starve out system control messages. Also avoid situation encountered with srun where putting srun to sleep causes hierarchical communication to get backlogged.

Programming interface The programming interface for the reduction network is a design activity that can be informed by the MRNet API [6], however because the reduction network is persisent and shared, it has somewhat different requirements in that it must interface to components running as distinct UNIX processes and be resilient to component failure. One approach is for applications that use the reduction network to provide a plugin that claims a message *tag* space and implements a *socket activation* scheme similar to systemd [23] or D-Bus [22] that associates the tag space with named UNIX domain sockets and/or executables at the front-end, back-end, and internal locations.

Fault tolerance The CMB provides notification messages when nodes cease to respond so that other services can manage failures. The reduction network will use this facility and track the CMB topology to remain functional when faults occur. However, although we are encouraged that fault tolerance has been achieved for certain use cases in MRNet as described by Arnold and Miller [10], we recognize that it will be a challenge to design our reduction network to be generally reliable and fault-tolerant. Therefore we leave the possibility open that the design will provide these attributes only for selected use cases and failure modes.

FIXME: The fault tolerance strategy both for the CMB and the reduction network is rather poorly developed in the description thus far. At minimum it needs to be called out in the WBS as a significant R&D activity.

kim-review: Fault tolerance should be more fully designed in the comms layer before moving forward.

6 Resource Management

6.1 Overview

At a high level, we designate the *Resource Management* (RM) thrust as concerning the configuration, scheduling, and tracking of resources, jobs and users in the NGRM system. The RM thrust is a key component of NGRM because it embodies the user interface to a site's resources and offers the capabilities for users to submit jobs and thus do useful work with those resources. Therefore, it is paramount that the RM components of NGRM not only be generalized, flexible, and extensible, but also *powerful* and *user-friendly*.

In this section, we will discuss the components of NGRM that provide our resource managemnt interface and functionality. With these components, we aim to fulfil the goals described above and in Section 4.2. We will start by describing our plan for a powerful, extensible domain-specific language for use in describing resources and requests for those resources. We will continue by outlining the design of a set of global databases that will be used by our RM software for configuration and historical data. Next, we will describe the architecture and functionality of a *job* within NGRM's unified job model, and how jobs will interact between parents and children within the resource management domain. Finally, we will discuss a flexible interface for job scheduling within the NGRM job, and some details about job scheduling implementations in the new system.

6.2 Resource Description Language

In any resource management system there must be a method by which the configuration of resources are communicated to the system. Typically this functionality is achieved via some form of static configuration, that is via a text file or database that is created manually or with the help of some kind of tool. In kind, users need to describe the resource *constraints* for their jobs in some form – typically via a combination of command line options, features requests, and or using some sort of resource specification language such as the Globus RSL [2].

For NGRM we propose a single resource configuration and specification language simply termed the resource description language (RDL). The RDL shall be a Domain Specific Language (DSL) that will be used to describe the hierarchical configuration, topology, and other data about resources in the system. The language will be structured, extensible, human-readable, and hierarchical, while being capable of representing resources and their relationships in a generic and flexible fashion. It is expected that this language will serve not only as the base *configuration language* of NGRM, but that this language will be the de-facto communication substrate for gathering resource information such as topology, current resource state, categorization, as well as constraint specification in resource requests.

6.2.1 Related Work

Fortunately, there exists a large body of literature in this field on which we can draw when designing the RDL for NGRM. While our pragmatic design may not focus on ontological formalism, there has been work in this area for describing distributed resources in a Grid [15, 36, 40] which show promise for semantic matching algorithms on generic resources. Van Der Ham et al. and others have extended the Resource Data Framework (RDF) specification from the semantic web to describe networked resources [49–51], and Koslovski et al. have developed the Virtual Resources and Interconnection Networks Description Language (VXDL) [37] for use in describing the end resources description and virtual network topology in on-demand virtual infrastructures.

Several other resource management projects have also explored this area. Possibly most importantly, the Condor project implements *Classified Advertisements* (ClassAds) which is a language for expressing not only resources and their attributes, but requests for these resources [1]. The suitability of resources for resource requests (jobs) are matched using an array of multi-dimensional gangmatching approaches. The ClassAd language is available as a standalone library. Additionally, the OAR resource manager defines resources in a MySQL database with a static schema, but it does organize resources hierarchically, allows generic resource definition, and allows users to request

resources using a resource description language [13]. In the Legion [16, 39] Resource Manager an inheritance based model is used for defining resources as "objects" that are extensible. Another example is CCS [34], which implemented the Resource and Service Description (RSD) language and its predecessor the resource description language (RDL). The RSD in CCS exports not only a language interface for resources description, but a graphical user interface and API as well.

As part of the design of our RDL, it is expected that we will do a more complete survey of existing work in this area such that we can apply common practice and lessons learned to our own implementation of a resource description and configuration language.

6.2.2 Resource Description Language Design

As mentioned above, as part of NGRM development we hope to create a domain-specific language that is easily extended and embedded, is human readable, and has the power to express our hierarchical resource data, topologies, and advanced resource constraints. The RDL should additionally have to ability to contain the topology of resources (which may be different than the heirarchy of the resources).

While a suitable existing encoding may be discovered during a full literature search, it is our feeling that a solution based on a static data description or markup language like RDF or XML is not going to have the ultimate flexibility and features that we require in the NGRM. For this reason, we currently propose that the Lua [32] language would be a good fit for our requirements.

Lua is a language that was designed to be embedded in other languages, so it would be easy to embed parsers for the RDL in various tools. It is also embarassingly easy to extend the language using modules or directly in native Lua. Finally, the core datatype in Lua (in fact the only datatype), is the Lua *table* (an associative array), which lends itself nicely to the expression of hierarchical data as we have noted will be necessary for the RDL in NGRM. There are many extant examples of the use of Lua as a Data Description and Domain Specific Language. See the Programming in Lua (PiL) Book [32] for examples.

```
ComputeNode = {
  type = 'host',
  attributes = { hostname = '', ipaddr = '', memory = ''},
  children = {
     { type = 'NUMANode',
       id = '0',
       children = {
          { Type = 'Memory', id = '0', value = 16384 },
          { Type = 'Socket', id = '0',
            children = {
              { Type = 'CPU', id = '0', children = {}},
              { Type = 'CPU', id = '1', children = {}}
            }
          },
       },
     },
     { type = 'NUMANode',
       id = '1',
       children = {
          { Type = 'Memory', id = '1', value = 16384 },
          { Type = 'Socket', id = '1',
            children = {
              { Type = 'CPU', id = '2', children = {}},
              { Type = 'CPU', id = '3', children = {}}
            }
          },
       },
     },
     { type = 'GPU', id = '0' },
  }
}
```

Listing 1: A naïve example of a Lua table describing an excessively simple resource hierarchy

Listing 1 shows a very simple example of a Lua table used to describe a hierarchy of resources within a compute node. Note that the attributes of the node resource are currently left empty, to

possibly be filled in as the table is copied and appended to the `children` table of another resource, such as a cluster. Used in this fashion – as a data store – a Lua table is very similar to JSON, another popular data interchange format.

It is not expected that the RDL in NGRM will use the naïve approach as in Listing 1. Because Lua is a full language instead of just a data interchange format, there will be many optimizations and syntactic abbreviations we can make to ease working with and using the NGRM RDL.

For example, we expect to use the *object inheritance* support in Lua to allow resources definied within the RDL to inherit from other resources. This should support collaboration and research by allowing sharing of resource definitions as RDL "libraries" or resource definition sets. For example, a base type might be a `Node` class which implements a set of interfaces that are common to all types of nodes (such as *has a* hostname, ip address and so on). Specific types of nodes can inherit from the base node object and add features (such as specialized devices, default tags etc).

Additionally, with the power of a full language at their disposal, system managers and users can develop a range of scripts and extensions that ease working with data in the RM system. For example, a sysadmin could write a script to create the definition of an entire cluster by reading a comma-separated value text file, or other formatted data.

The risks of using a full language like Lua as the RDL for NGRM are also numerous. Since the RDL will be used for resource requests and definitions, priviliged code within our RM system may be compiling and running untrusted code. While Lua has very good native support for sandboxing [3], this is a notoriously difficult practice to get right, and code using the RDL may need to be hardened extremely well for any use in a production environment. Evaluation of user-supplied code should be done within unprivileged processes as much as possible. Also, when using a dynamically typed, runtime-compiled language like Lua, there is an increased risk of runtime exceptions, so extra care must be taken to handle errors correctly, and it is likely that a specialized "RDL validation" function must be written.

6.3 Global, Persistent Data in NGRM

In the *Unified Job Model* of NGRM we combine the concept of a traditional job with the idea of a resource management *instance* which provides the traditional features of a resource manager and batch scheduler. However, the top-level *root job* in such a system will need to be initialized from somewhere. Additionally, in order to be useful, the RM system will need some sort of persistent record of jobs that ran on the system, for how long, and on which resources.

To satisfy these requirements, we introduce the global, persistent *Resource Inventory* and *Job and User Repositores*. These facilities operate outside of the *hierarchical job model* in NGRM and act as a source of ultimate configuration and historical data about the RM system. Each of these facilities is described in more detail in the sections below.

6.3.1 Resource Inventory

As noted above, the Resource Inventory is a global, persistent database which acts as the top-level configuration for all jobs within the NGRM system. The resource inventory itself will support being initialized, modified, and queried using the RDL, and thus will be optimized for the storage of hierarchical resources and their topology.

To satisfy our generalized resource model we must strive to build a resource inventory that is itself generalized and flexible. To this end, we propose that the resource inventory implementation support arbitrary *tagging* of resources. Resource tagging is a more general approach than the practice of giving nodes features or properties as in traditional resource managers. We also propose that the tagging approach is powerful enough to support RM features such as marking resources *down* or *drained*, an even *allocated*. Furthermore, it is common practice for tagging databases to allow users to supply their own tags to data in the system. Use of this kind of *collaborative tagging* or folksonomy [52], could be very useful in creating a socialized system of resource management.

In order to enable the development tools around the resource inventory, and to support notification of resource changes and reconfiguration to jobs in the NGRM system, we propose that the

jeff-review: We as a center need one go-to place to find resource inventory information. We have multiple web pages, spreadsheets, and files that are being maintained separately right now. It'd be nice if your resource inventory DB was the central place for this info – what are the resources in the center, what are their properties, etc. I love the idea of being able to subscribe to this repo to be alerted of changes – much better than the current model where someone has to notice or remember a change was made.

how to store hierarchical data and topology information together is a subject for further research.

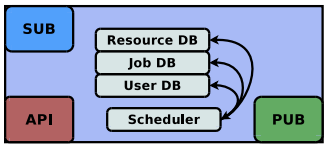

Figure 9: Components of a Resource Manager Instance

resource inventory export a *subscribe* interface in addition to a more traditional API. The subscribe interface will support filtering so that tools (and jobs) can get notifications of specific changes (perhaps to a subset of resources, or changes to a particular tag). For instance, the root job in NGRM will subscribe to the resource inventory to get updates to resources, such as resources that are drained or modified.

6.3.2 User Repository

jeff-review: Any advantage/possibility in leveraging LC's LDAP for this?

Since NGRM is a software system that will be used to provide *users* with access to compute resources, it will require some place to store information about those users. For this purpose, we will require a database of user information alongside the resource inventory. We call this global, persistent user data store the *user repository*.

At the very least, the user repository will store minimal data about users of the current deployment of NGRM. However, the implemntation should be flexible enough to allow the storage of other user-specific information, such as defaults, limits, roles, accounts, qualities of service, and so on.

6.3.3 Job Repository

jeff-review: If users can add their own data to the provenance database, how is long-term storage managed, especially if users decide they want to store a movie or data file as part of the permanent record? **jeff-review:** To do (data provenance) "right", each job would have to have a true UUID, which is an ugly thing to work with for humans. So maybe there'd be a regular job id that's local to the domain (cz, rz, etc.) that is just an auto-incrementing id, plus the 16-byte uuid that truly is unique? **jeff-review:** Currently SLURM bogs down if a person does an extensive long-running query (sreport) against the job database; how would your design avoid this?

The *job repository* is the final global and persistent data store in the NGRM system. This top-level database is a historical record of all jobs that have completed in the NGRM system. Along with the obvious data about a job – the start and end times, the resources assigned to the job, and the owning users – the NGRM job repository will also store a complete provenance record for the job. The provenance record will (configuration permitting) contain data such as the job environment, namespace or list of installed packages, fault stream and other job-specific monitoring data. It may also be beneficial to store other job information such as input files and stdin/stdout streams, so the job repository will not be designed with a rigid schema.

Another challenge in designing a job repository for NGRM is storing job data from the *hierarchical job model*. Similar to the resource inventory, the job repository will need to be optimize to store hierarchical data. The intent of the job repository is to store *all* job data, down to even the lightweight job invocations in the leaf jobs of the system, so there will be a mass of hierarchical data here. Developing a system that not only efficiently represents that hierarcy, but allows advanced and intuitive queries for the data should be a top priority.

6.4 Job Anatomy

The resource inventory and job and user repositories are not tied to any one job – they exist at the global, persistent level in NGRM and export a common API. To fulfil the unified job model, however, each NGRM job must contain all the same features implemented by these global databases, but local to the job. For this purpose, we introduce the concept of an NGRM *instance*, which is the embodiment of all resource management features at the job level.

The components of an NGRM instance are outlined in Figure 9. These components largely match the global, persistent data store described in sections above, but since jobs in our RM system may be ephemeral, these implemenations are lightweight and may be started on-demand instead of being always instantiated. This practice of minimizing the functionalitly instantiated per instance will not only serve to reduce the cost of starting and stopping the software for every job, but optimizes for the *base case* job in NGRM – a single job running a single parallel application.

Within an instance, we give these temporary counterparts to the global, persistent databases a new name to differentiate them. They become simply the resource, job, and user databases. Each of these job-local databases contain only the subset of information which applies to the current job. For example, the resource DB will have information only about resources assigned to the job, the user DB will have only the users allowed to run within the job and the job owner, and the job database will contain information only for jobs that have run in the current job or a completed child.

As a whole, the RM instance exports similar API and publish interfaces as the top level persistent databases. For instance, most interaction with the instance functionality of a job will be through the well defined API. However, other functionality may be more adapted to a PUB/SUB interface, for example a parent job will subscribe to its children for events of interest, and child jobs may subscribe to parents for information trickling down from higher levels in the hierarchy. Interesting changes in this context may include data about resources in the child (this resource is dead), information about the child itself (I'm dead), or information about resources from the parent (this resource is being drained in 20 minutes).

Instance API and pub/sub interface is a topic of future research.

Below, we discuss in slightly more detail the resource, user, and job databases. Detailed descriptions of these components in the context of the job lifecycles are described in later sections. We also take this opportunity to touch upon the scheduler interface within the RM instance. However, a detailed scheduler discussion is saved for a later section.

6.4.1 Resource Database

The resource database within a job is expected to be implemented as a read-write *cache* of the job's subset of the parental resources. The resource data for a job must be writable in order to satisfy the *self-hosting* requirement of NGRM. An owner of a job should be able to update tags and configuration of resources within their job, however, non-privileged user tags and configuration will be marked as such, and so shall not affect upstream or ancestoral job or resource configuration.

6.4.2 User Database

The user database within a job contains, at a minimum, the job owner along with an optional list of users that are allowed to access the job. As described above, the user database may be as simple as a list of user names, but it is extensible enough to be used for an array of other user-specific data – preferences, roles, limits, and qualities of service are just some examples.

In NGRM we would like to grant users the ability to *invite* other users to submit jobs to a job for which they are the owner. A simple example of this functionality would be the root job, which is owned by user *root* and to which *all* users are invited. However, other use cases for this functionality include dedicated application times (DATs) in which a coordinater may invite a group of users, or a debug session on a large running running job where debugging experts may be "invited" to run some diagnostics. To support this functionality, we propose that the job *owner* should be able to add users to the user database, and that those users would then be able to submit new job requests to the instance. Users with *submit* access shall not have access to runtime functionality of the job, nor the ability to initiate lightweight jobs. However, it other circumstances it may be useful to have the ability to expand the list of users that *do* have such access. Therefore the user database must support both *submit* and *runtime* access lists for jobs, though it may be wise to make these lists optional based on configuration.

6.4.3 Job Database

The job database serves the same purpose as the global job repository within an instance, but additionally acts as the data store for pending and running jobs within the instance. Jobs are submitted to a local instance via the API. The syntax and validity of job requests are checked and the job request is passed through a set of plugins that may modify or reject the job based on site-specific requirements. Valid job requests are inserted into the job database, at which point a "job created" event is generated. Scheduler and other plugins may then act upon this event.

The job database will also support generating "job start" and "job end" events.

The exact list of events generated by the job (and other) database is a topic of future study. It may be useful to research a method to allow *arbitrary* events from all these datbases. However, the "job state change" events are key, so they are called out explicitly here.

6.4.4 Scheduler Plugin

Another major component of the RM instance is the job scheduler. The scheduler is discussed in more detail below, however here we note that the scheduler implementation is provided by a plugin, and that the loaded scheduler is configured at the time of job instantiation. Thus, while it is expected that the root job will likely have an advanced, complex scheduler plugin loaded, most jobs running in the RM system will have a default, simplistic scheduler implementation, or perhaps no scheduling code loaded at all, if it is unnecessary. It is also important to note a scheduler plugin (or any other RM instance plugin) is running within the context of a NGRM job. Therefore, these plugins have access to the full power of the the distributed job environment for use in parallelizing work and distributing state for fault tolerance and scalability.

6.5 Job Physiology

We have briefly described the RM components in the NGRM above. In the sections below we expand on this discussions and begin to describe how processes within a job interact with the containing job, and how the RM instances within jobs in the *hierarchical model* interact with eachother.

6.5.1 The Root Job: *Job 0*

In our hierarchical job model, every job in NGRM has a parent and zero or more children jobs. It is obvious, however, that the root job (or *job 0*), does not have a direct parent. In this special case, the resource inventory acts as the indirect parent of job 0. When the root instance is bootstrapped by the CMB, the root job connects to the global, persistent NGRM databases and recieves just enough resource, user, and other configuration data to initialize. The root instance then *subscribes* to the resource inventory so that it is notified of important changes.

Again, the list of "important" changes is a subject of further study. However, an administrator could tag a resource 'down' in the resource inventory and job 0 would then be notified immediately through its subscription. This would trickle down to the appropriate progeny via the pub/sub channels in each instance

Once the root job is initialized, it may become the de facto interface for users to the system, i.e. the instance within job 0 may provide the interface for all users of NGRM at a site for job submission, queries, and other RM-related work. In this case, jobs are submitted to the root job's API and are inserted into the job database as described in Section 6.4.3, and the job 0 scheduler will prioritize and schedule these requests.

In our system, however, this is not the only scenario. The flexibility of the unified job model allows a site to invoke sub-jobs of job 0 and reference those jobs as respective interfaces to subsets of resources within a datacenter. For instance, to run in a more traditional mode, a centerwide root job could spawn one sub-job for each cluster within a datacenter. These cluster-specific jobs would then act as separate "resource managers" for these clusters, and could therefore have different scheduler plugins loaded, a different set of users, or other separate policies. However, even in this case, it is important to note that job 0 still spans all the resources in the center, so job data is eventually collected in the top-level job repository, all resources in the center are configured in a singular resource inventory, and a centerwide runtime and monitoring environment exists for use by administrative tools.

6.5.2 The NGRM Job: *Job* n

As explained in the sections above, a job requests in NGRM are always submitted to an existing NGRM job via the instance API. This job request is passed through a set of configurable plugins which check the validity of, and make any modifications to, the new job request. If the request is valid, and the requesting user has submit access to this job, then the job request is inserted into the job database as described in section 6.4.3 above.

After the scheduler has determined the job should run and resources have been selected for the job, a new comms session is spawned, and the CMB intiates the new instance for this job, which we call job n. The instance is initialized using the universal resource description language of NGRM which will describe the set of resources assigned to the job. Once the new instance is initialized, it will subscribes to its parent instance (for notification of upstream resource changes, and other items "of interest"), and the parent subscribes to the child instance. Finally any script or interactive

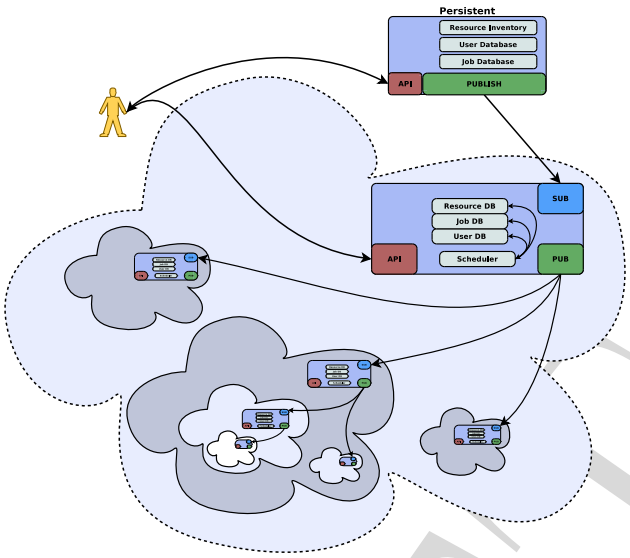

Figure 10: Resource Manager High Level View

session is spawned within the job, with configuration such that RM utilities and APIs will utilise the URIs specific to the job n instance.

TODO: Flesh out the details here. Need to describe batch vs interactive vs "pure allocation" jobs.

Upon initialization, ownership of job n is set to the submitting user, and that user would typically be the only user allowed access to the runtime environment of the job. However, the list of users with submit access and runtime access is configurable and may be set after job invocation or during the job request itself.

It is also not necessarily the case that the instance invoked by the job – that is, the set of daemons and code that provide the RM instance features in NGRM – are of the same versions as the parent job. In fact, part of the design of the self-hosting, unified job model of NGRM is to allow not only *rolling updates* of software, but to allow future and experimental versions of the software to be deployed as an NGRM job. This means that the version of the software system used within a job may be user-selectable, or for privileged users a version of the tool could be loaded directly from a build directory.

A study of the security implications of these features is outstanding. However, it is expected that *certain* pieces of software would be deemed safe to allow blanket per-job user replacement rules. (e.g. the scheduler)

Once job n is running and fully initialized, user processes have access to a rich and powerful private resource management system for use in directing their computational work. Information about resources can be gathered and stored in the local resource database, which in turn can direct and optimize behavior of the in-job scheduler. Chained jobs and workflows can be inserted into the local job database and child jobs can be created which in turn use the power of their local RM system to manage related work. Every sub-job and parallel invocation is captured in the local job database, along with monitoring data streams and other job data which can be used in postmortem analysis and job efficiency studies.

A graphical representation of a typical set of jobs showing jobs running within jobs and their respective instances is shown in Figure 10.

6.5.3 Lightweight Jobs

As described earlier, lightweight jobs (LWJ) are similar to normal jobs in the NGRM system, but they do not result in a new instance. Thus, these jobs run within the current instance and utilize the same scheduler, resources, and other instance specific data. It is expected that lightweight jobs will be the vessel by which parallel applications are launched and managed, and this topic will be explored further in the workload and runtime section.

For the RM instance within the job, all the same steps are taken when handling a lightweight job vs. a normal job. There may only be a flag or tag that indicates this job will be *lightweight*

instead of requesting the instantiation of a full job. This approach is beneficial because it reduces code duplication in handling the distribution and management of processes local to the runtime of the current job. Another benefit of treating lightweight jobs as a special case of regular jobs is that the lightweight job information will be captured in the job database and preserved, just as any other full job.

One final note about lightweight jobs is that it will likely be necessary to allow lightweight jobs instantiated on the system to overlap. For example, one LWJ might be a parallel MPI application running across all nodes of a job, and second, overlapping LWJ would be a parallel debugger session used to attach to one or more of the processes in the original job. Again, both of these jobs would be captured in the job database.

It is assumed that the lifecycle management of LWJs will be handled at the level of the runtime environment. It may not be possible in all cases to determine exactly when a LWJ is "complete", and it may be up to the user code creating the LWJ, or the runtime subsystem doing so on behalf of the user, to explicitly notify the RM instance when a LWJ has completed.

TODO: What happens when a LWJ request comes from within a LWJ. I think it is Dong's assumption that LWJ's have their *own* hierarchy – so the current level in the hierarchy would need to be communicated to the RM instance, and the instance would need to manage that hierarchy.

6.5.4 Job End-of-Life Care

We will adopt a model from UNIX process management and have the parent instance "reap" its children. It is here that the job db from the child can be "pulled" up from the child into the parent job db. When instance 0 reaps jobs, this job data can be pushed up to its parent, i.e. the persistent job repo.

In this model, historical job data makes its way up to the top-level job repository as child jobs complete. Each "running" job instance has information about its historical job lineage in its local job db, so this information can also be queried directly from within the context of a job. (e.g. in a DAT, if you want to only query information about jobs from the DAT, you can query the DAT job instance. In fact, information about jobs from the DAT are not populated to the top-level job repository until the DAT "ends")

TBD – how to store child job information in the parent such that the historical lineage of the jobs and sub-jobs within the child are preserved. Maybe the reaping should be abstracted down in the comms layer with callbacks to allow the higher level subsystems to reap their analogs in children.

We will define how jobs indicate that they are "done" and need to be reaped. A DAT "job" may need to be killed and reaped at the end of its time limit. Normal batch jobs are complete when the batch script running on the control node exits. A direct allocate/launch with WRAP would be reaped when the processes being launched exit. A job may be forcibly killed by the user or an administrator, at which point a depth-first recursive termination is issued.

TBD – What happens to an active job hierarchy when the top-level job is killed? What happens to pending job requests when a job is terminated? Killing a job might result in something like:

1. Freeze local job db (i.e. disallow new job submissions)
2. Terminate and reap children jobs
3. Kill local tasks

Perhaps 2,3 could be swapped or done in parallel.

6.5.5 Job Fault Tolerance

While discussing job interactions, it rapidly becomes apparent that an important feature of the NGRM job will be its tolerance of faults in the RM instance. If there is data loss within the instance, then this may affect the data of all completed child jobs, since hierarchical data such as the job database are generally collected in the parent after a *reap* operation. While fault tolerant design is a goal of NGRM design and a topic of further study, we mention here some initial ideas for job recovery after RM instance failure.

One possibility is to build the resource, job, and user databases on top of a distributed data store with replication. The runtime environment of job will already contain a distribute key-value store, so it may be possible to build a fault tolerant data store on top of that. With replication enabled, the failure of a node on which the RM instance is running would not be fatal because the data could be reconstructed from replicas.

Another proposal is to replicate data to other instances running in the system. For example, a job could replicate data to its children and parent, and a lost instance could then be reconstructed by the parent by querying missing data from the children. This method would require a systematic method for determining location of child jobs without help from the parent, but it is expected that this challenge is not insurmountable.

Because the impact of losing job instances while running may be large, in fact larger than in some other RM software systems, our development should focus on testing this case, and ensuring that there is no undesired data loss when an RM instance crashes or its host goes down should be part of a routine test plan.

6.6 Job Scheduler

chris-review: Plugins described in scheduling section seem like they are each pigeon-holed for a particular function, compared to SPANK where one plugin is called in different contexts to provide a suite of related functions.

The NGRM Job Scheduler is responsible for scheduling computing resources to users' jobs. Users submit to the scheduler requests for resources to run their job. The scheduler implements management's policy to decide when and where to allocate the resources for each job.

This section summarizes the requirements for the NGRM Job Scheduler, a rough design which meets those requirements, and a work breakdown structure for developing the scheduler component.

6.6.1 Motivation

Scheduling batch jobs across a collection of networked computing resources connected in a grid has been a common paradigm for at least two decades. A batch scheduler receives users' job requests, selects a cluster for each job, then dispatches the job to that cluster's resource manager to be executed.

The NGRM Job Scheduler represents a departure from traditional monolithic "grid masters". The scheduler functionality will be a service provided by the NGRM's job model. As such, the scheduling activities will be distributed across the center's resources and provide functionality not available in any commercial or open source project.

NGRM Job Scheduler's scheduling services will schedule jobs across resources in a computing center without regard to current cluster boundaries. A job will be able to request resources containing a common feature (like connectivity to the same high speed switch) or fitting within a limited power envelope.

The NGRM Job Scheduler will support plugin modules that provide unique scheduling behavior and job prioritization. Each job will have the option to independently load its own scheduling plugin. In so doing, the NGRM Job Scheduler's scheduling capabilities will range from scheduling all resources in the center to scheduling jobs on dedicated resources (DATs) to scheduling LWJ's (job steps).

Most importantly, the traditional boundaries between a job scheduler and the resource manager will be redefined under NGRM. Instead of a resource manager that manages every resource of a cluster, the resource management services will be instantiated as part of the job and be restricted to only the resources allocated to the job.

In order to continue to meet the needs of current users, the NGRM Job Scheduler must continue to provide all the services that production schedulers provide. Our goal is to surpass our existing schedulers in the following areas: performance, accuracy, reliability, resiliency, ease of use, flexibility, security, diagnostics, and need for manual intervention.

6.6.2 Requirements

While a more detailed list of requirements is presented in A.3, the following provides an overview of the functionality that the NGRM Job Scheduler will be expected to deliver.

Fundamental Requirements The following is the most definitive list of basic scheduling requirements. The job and resource repositories as well as the job submission facility are external to the NGRM Job Scheduler.

chris-review: Why are "job prioritization" and "job scheduling" independent activities? Shouldn't prioritization just be one aspect of scheduling?

- Prioritize each job
- Schedule each job based on its resource requirements

Further Scheduler Requirements In addition, more elaborate scheduling plugins will be provided to do the following:

- Support complex job dependencies, e.g. as in scientific workflows
- Backfill lower priority jobs whenever possible

- Facilitate dynamic job growth and reduction
- Preempt running jobs to free up resources needed by higher priority jobs
- Calculate estimates of when each job will begin
- Provide scheduling answers to "what if" scenarios
- Scheduling different resources to a job over time

Policy Enforcement NGRM implements the center's policies for providing access to its computing resources. The following are responsibilities, traditionally associated with a batch scheduler, that will be borne by the larger NGRM system:

- Reject job submissions for jobs which cannot or will never run
- Remove jobs that exceed time limits
- Enforce established limits on users, groups, projects (banks), etc
- Honor service level agreements and service quality requests

Organization Components The NGRM Job Scheduler functionality is broken down into the following components.

Job Prioritization. This the facility for prioritizing jobs based on potentially multiple factors. The system shall offer a job priority plugin framework to allow custom algorithms for determining job priority. The priority of each queued job must be continually recalculated as the queue of jobs and workload factors change.

Job Scheduling. For each job removed from the prioritized queue, computing resources must be reserved and eventually allocated. The collection of resources to schedule must be available from the resource inventory with the state and status of each resource updated in real-time. The scheduler must honor multiple resource requests simultaneously as it seeks to allocate cores, GPUs, nodes, switches, bandwidth, power, etc.

chris-review: There should be multiple levels of preemptability.

If users can add their own data to the provenance database, Here too, the system shall offer a plugin framework to support custom algorithms for scheduling jobs to compute resources. An essential scheduling algorithm which must be included is backfill scheduling (lower priority jobs are scheduled to run if they do not delay the start of higher priority jobs). In addition, qualities of service must be implemented in the scheduler such that running jobs can be preempted if needed to free up resources for more important jobs. This involves not only selecting the best resources for a job, but also identifying the set of jobs to preempt when such a policy is enforced.

The output of a the job scheduling process is a schedule of which jobs are mapped to which resources over a future, rolling period of time. A by-product of this schedule is a projected start time for every queued job that is included in the schedule.

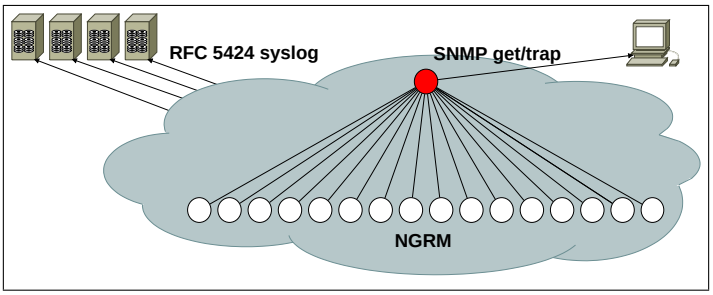

Figure 11: The NGRM monitoring system interfaces with an external enterprise monitoring system using SNMP, and with a persistent log database using RFC 5424 structured syslog.

7 Monitoring

The primary function of NGRM's integrated monitoring is resource health tracking. This information is required in real time by schedulers and runtimes to ensure that work is not launched on broken resources, and that when things do go wrong, appropriate action can be taken. In addition, NGRM monitoring can be extended and customized by sys admins and job owners to cover additional monitoring needs that may be site, hardware, or job specific (A.3 req. 3.1). Ideally NGRM will be flexible enough to meet all monitoring data aquisition requirements on compute nodes, where our model is to synchronize monitoring interruptions within each job and allow the system noise impact of monitoring to be tuned by the job owner (A.3 req. 3.0 and 3.2).

As shown in Figure 11, monitoring interfaces with an external log database and enterprise monitoring system. The log database is intended to be a comprehensive, schemaless, site-wide store that will support a high insertion rate, large storage capacity, and scalable queries. As a record of all events in the data center, it will facilitate postmortem analysis, enabling the correlation of job data with other interesting occurrences that might not have been recorded in the job database, or anticipated as something the job would normally "care" about. Users will be permitted to inject application-level information into this store and perform analysis with the system-level context there as well (A.3 req. 3.6).

The enterprise monitoring system is the mechanism used by operations center staff and system administrators to monitor site systems which might include NGRM as well as facilities, network devices, storage appliances, and non-NGRM clusters. This system is likely to already be in place at a site, thus common protocols are chosen to reduce the effort required to integrate with NGRM.

Monitoring state for a job is stored in the *resource database*, and fault events are recorded in the *job database*. These databases provide extensibility and persistence features described in Section 6.4. Monitoring follows the NGRM job recursion pattern, and is layered upon the comms framework, which assigns each job a unique domain name within the NGRM private DNS namespace. Live monitoring data can be obtained by using the resource manager API to query the resource and job databases on the job's control node, using the job's domain name.

Some applications and runtimes will require notification when system faults occur. For example, when a node that is part of a job crashes, or is about to crash, some applications may be able to request a replacement node and recover. To facilitate sharing of fault information, NGRM will implement a *fault notification service*. Applications and runtimes use the fault notification service to produce and consume fault events within the job. In addition, a gateway on the job control node allows software within the job to subscribe to faults generated externally (such as by a file system used by the job), and to publish certain faults that may be of interest to others. Faults can change resource health state maintained in the resource database. The *fault stream* produced by a job is logged in the job database and can be considered part of the job's provenance record.

NGRM Monitoring thus consists of the plugin framework, log database interface, fault notification system, and enterprise monitoring interface. Each of these parts is discussed below.

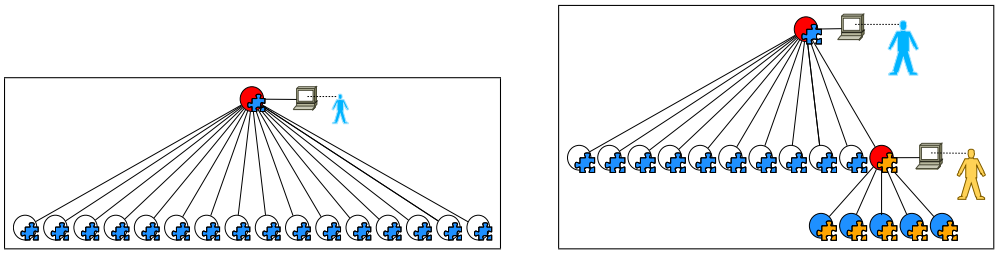

Figure 12: Monitoring follows the job hierarchy. Control nodes (red) store resource status information in the resource database and optionally export it via SNMP. The monitoring plugin stack (jigsaw pieces) is customizable for each job. Plugin execution is coordinated by the job's scheduling trigger event.

7.1 Plugin Framework

The monitoring plugin system provides a mechanism for arbitrary code contained in a user- or admin-provided plugin to be periodically executed across a job. The primary function of a plugin is to update resource health information in the resource database at the control node, using the reduction network. Plugins may also publish fault events and send data to the log database. A default plugin stack is inherited by a job from its parent. The set of active plugins as well as the trigger event period which synchronizes execution can be tuned to a degree by the job owner. The plugin framework is depicted in Figure 12.

Plugins have three main functions: data source, data reduction, and data sink. Depending on the role of a node within the job or its position on the reduction network, one or more functions may be enabled on the node. For example, a compute node may run only the data source function, while the control node may run only the data reduction and data sink function. Any function can publish fault events and send data to the log database in addition to performing its role on the reduction network.

The data source function is driven by the scheduling trigger. Its its purpose is to perform the first level of sampling or testing of an object that is being monitored, and inject the resulting data into the reduction network.

The data reduction function accepts data coming from upstream source or reduction functions of the same plugin. Its purpose is to reduce the data in some way to improve scalability. Reduced results are sent downstream, eventually to the plugin's sink function. The execution of the data reduction function is driven by incoming data and timers, not by the scheduling trigger.

The data sink function accepts reduced data from the same plugin on the reduction network. Its main purpose is to update resource state in the resource database, although it could dispose of the plugin's data in other ways such as by interfacing directly with a tool, or updating a database supplied by the job owner.

Special care must be taken in the design of the plugin execution environment to minimize disruptive impact on compute nodes. For example, some monitoring systems like Chukwa [42] require data source functionality to execute in a Java VM, which would have an unacceptably high memory and scheduling impact on some workloads. Others like Nagios [4] rely on shell scripts that may have a similarly high or unpredictable impact. NGRM monitoring plugins should leverage a lightweight execution environment such as an embedded Lua [32] interpreter.

7.2 Log Database Interface

Monitoring interfaces with an external log database intended to be a comprehensive, schemaless, site-wide store that will support a high insertion rate, large storage capacity, and scalable queries. As a record of all events in the data center, it will facilitate postmortem analysis, enabling the correlation of job data with occurrences that were not actively tracked by the job during its execution, thus not part of the canonical job record.

The log database, although implied by our requirements, (A.3 req. 3.6), is "outsourced" by NGRM with a generic interface so that sites can choose the technology to use to build such a system. Some sites may prefer proprietary systems such as Splunk®, while others may wish to build one from the many horizontally-scalable NoSQL databases such as CouchDB [9] or mongoDB [41]. Still others, operating at modest scale, may employ a traditional flat file or relational database. At LLNL, an in-progress Laboratory Directed Research and Development feasibility study on HPC log analytics [24] may spawn a separate project for the log database.

The syslog protocol, modernized in RFC 5424 [25], includes a provision for STRUCTURED-DATA content, an easily parsed format that is user-extensible. Since log data may be voluminous at times, and scalability may require a *distributed* log database implementation, it is not desirable to use the NGRM reduction network to funnel all log messages through the single control at the root of the job. Instead, we allow monitoring plugin functions, or applications through the standard syslog API[6], to inject data directly into an orthogonal syslog transport. Syslog implementations already have standardized filtering, forwarding, and security capability so there is no need to reimplement this within NGRM.

To improve scalability in some situations, the reduction network can be employed without necessarily resorting to control node funneling. A monitoring plugin can log from the function (source, reduction, or sink) that gives the right amount of reduction/funneling for the data managed by that particular plugin.

Limiting unchecked log growth While it is well and good to design a capability for scalable, persistent logging that is available both to system software and user applications, growth of the log database should not be completely unbounded and unmanaged. Two features could ease this problem: a *circular debug log buffer*, and a *log insertion quota*.

Syslog verbosity is tunable by selecting the *level* of each *facility* that is to be logged, from LOG_EMERG (system is unusable) to LOG_DEBUG (debug-level message). Usually system log levels are set somewhere in the middle, but that means valuable log information leading up to a failure is sometimes not available. A solution to this problem is to create a local circular debug buffer that logs at the maximum verbosity, and tie the logging system into the job's fault notification service. If a fault occurs in a particular facility, the circular buffer can be dumped to the log database; otherwise the data is discarded as it is overwritten.

Some log databases such as Splunk®, have licensing based on ingest rates. Such a system, or indeed other systems that we wish to limit, could be operated with consumable resources held by the resource manager. For example, a job could request a certain quota of log messages for the duration of its run. When that number is exhausted, a fault occurs. This enables the NGRM scheduler to ensure that the ingest rate of the log database remains under control, while giving users a new capability and a motivation to implement in-situ data reduction.

7.3 Fault Notification System

The CIFTS group has argued [29] that a wholistic, full-system approach to fault notification is required in order to enable fault-tolerant applications and runtimes to make good decisions when things go wrong. For example, faults occuring on Lustre file system servers may be of interest to a program controlling a suite of application runs, but traditionally, file system failures are not reported in the application domain except through system call failures. NGRM addresses this need with a fault notification system based on the comms layer's reduction network and PUB-SUB event notification service. Fault notifications are published (by monitoring plugins, applications, etc) to subscribers within the job. A *fault gateway* allows configured local faults to be published externally, and configured external faults to be published locally.

We have adopted the CIFTS Fault Tolerance Backplane API [5] (FTB-API) an emerging standard for fault notification available in some MPI implementations and other software as the programming

[6]It is not clear that any available syslog API's handle RFC 5424 STRUCTURED-DATA except as an opaque component of a textual log message. If one cannot be located, likely we will want to write one to make management of structured data easier on users and ourselves.

interface for the fault system. We will use CIFTS event namespace conventions as well. This choice reduces the overhead of porting fault-tolerant runtimes and other software between systems, and allows FTB-API based implementations to be immediately functional on NGRM.

While the CIFTS reference implementation defines a *backplane* architecture for fault notification, our fault notification system leverages both the hierarchical relationship between jobs (local vs global fault scope), and the reduction network within a job (reducing duplicate or same-root-cause faults) to achieve greater scalability and more intelligent fault processing. Faults can be directly published across the job via the comms event notification service, but when data reduction is desirable, monitoring plugins can be employed to route faults through a reduction sieve to the control node where they are published in processed form. Monitoring plugins are also employed when the occurence of a fault needs to affect resource health state in the resource database.

The fault gateway, running on the job's control node, integrates the job's fault notification domain with the system's. It re-publishes a configurable set of local faults to the parent job, which forwards them to its parent, and so on, reaching subscribers anywhere in the system. Conversely, the fault gateway subscribes to a configurable set of global faults in the parent job and re-publishes them locally. In this way, external faults "of interest" to any job become part of the job's local fault domain.

The job's *fault stream* thus becomes an important record of anomolous conditions occuring within the job and those of interest occuring outside the job. It is stored in the job database.

7.4 Enterprise Monitoring Interface

jeff-review: MyLC and the new HM will both be providing notifications for various events, so we need to develop a centralized mechanism that can support this and make sure NGRM can be plugged-in as an event source. (Sounds like the pub-sub model Mark described would work nicely. **jg:** FIXME: This section describes how NGRM can act as an SNMP event source (trap) and provide SNMP state transfer (getbulk) to external monitoring. Missing is the ability to go the other way, e.g. get events and state on externally monitored objects. **jg:** FIXME: Other external protocols could be supported by other gateway implementations. The gateway is just a thin layer between resource db and external protocol engine.

Site operations centers and system administrators use monitoring to track problems that may require someone's intervention, and to gain insight into how the systems they are responsible for are being used. The systems that are monitored may include clusters as well as facilities and networking equipment. NGRM monitoring gathers resource health information that is an important component of that view, and indeed aims to replace other compute node resident monitoring frameworks in an effort to synchronize monitoring interruptions across jobs. Although NGRM will provide tools for viewing its internal operation including resource health, it must also export resource health information to external enterprise monitoring software to allow NGRM to be integrated into a site-wide monintoring strategy, perhaps already deployed. We accomplish this using our hierarchical job model to ensure that *actionable* fault information is available in the root job resource database, and develop a gateway to export this information to external enterprise monitoring software.

The health state of resources is maintained in the resource database of each job. This state is initialized from the parent resource database at job inception and is transferred back to the parent at job teardown. In addition, some fault events within a child job are published to the parent and thus may alter resource state in both the parent and the child (perhaps driven by the need for enterprise monitoring to see them). Therefore, although each job is given a delegation to monitor the resources allocated to it, resource health state does, with varying degrees of latency, eventually recurse to the root job's resource database, thus it is appropriate to interface enterprise monitoring there.

SNMP [47] is the de-facto standard protocol for enterprise monitoring. An SNMP gateway that speaks both the resource database API and SNMP will run on the root job's control node. State can be *pulled* out of NGRM via SNMP GET and GETBULK requests. After an initial state transfer, state changes can be *pushed* by NGRM via SNMP TRAP requests. This model supports multiple external management entities. Theoretically the SNMP gateway could run on any job, if desired. For example, long running service entities like Lustre server clusters could be "peeled off" of the root instance, assigned to a child job, and monitored separately. Practically speaking, jobs configured for external monitoring should be long lived.

A base set of NGRM management data will be defined in an NGRM enterprise-specific management information base (MIB) module. The set of data exported by the SNMP gatway can be extended by adding new MIBs that map new SNMP objects to resource database objects.

Term	Description
u	the containing job (universe)
r_u	the overall resource bound the scheduler set for u
j	an LWJ in u
$parent(j)$	j's parent in the LWJ hierarchy (parent of top-level LWJs is u)
c_j	resource criteria for j
d_j	some data to be recorded by j
r_j	compute resources allocated to j where $r_j \subseteq r_{parent(j)}$
$cnew_j$	criteria for additional resources for j
$rnew_j$	additional resources where $rnew_j \not\subseteq r_j$ and $rnew_j \subseteq r_{parent(j)}$
e_x	a run-time environment where $e_x \in E = \{e0, e1, ..., e_{m-1}\}$

Table 2: Definitions for Basic WRAP Service Parameters

8 Workload Run-time and Placement

kim-review: WRAP section uses different terminology than other sections.

The Workload Run-time And Placement (WRAP) thrust area concerns all aspects of executing transactions within a single job. While the scheduler sets the overall bound for both resources and time that a job can use, it does not dictate how to execute the various transactions of the job. Thus, it is WRAP's responsibility to ensure that these transactions get executed most efficiently within this scheduler-set bound. To embody NGRM's new resource management paradigm, however, WRAP must provide the run-time services beyond what the traditional paradigm requires. In addition to the traditional services such as bulk process launch and management, WRAP must provide advanced services to support conceptual models such as job hierarchy and resource elasticity described in Section 3.2.

8.1 Lightweight Jobs and their Hierarchy

chris-review: Why do we need a hierarchy of LWJ's? Are we reinventing the same stuff we already built for jobs? Can't we just do LWJ stuff with jobs?

As explained in Section 4.1, the traditional approach models various transactions that a job executes simply as a set of compute steps—e.g., `job steps`. As with other limitations of the traditional paradigm, this simple model is ill-suited for designing our run-time services after it. Instead, WRAP requires a more powerful and flexible mechanism to organize and group the processes that a job executes in accordance with their distinct functions or purposes. For example, all of the parallel processes of an MPI application may form a single compute function; all of the distributed processes of a parallel debugger program may form a tool function that should be logically separate from the compute function; further, the compute function may refine itself into several sub-functions to serve independent power capping functions [45] to different subsets of its processes.

We use the notion of the lightweight job (LWJ) to realize the new model. An LWJ is a group of processes with a distinct function that has its own resource confinement that is a subset of the overall resources assigned to the job. The most significant difference between the LWJ and the full job is that an LWJ can share the compute resources with other LWJs of the same job. Further, the processes grouped by an LWJ can be refined into smaller LWJs, and thus LWJs also form a hierarchy. Effectively, this bridges our general *job hierarchy model* into the fine-grained scope of "within-job," following the same "parent-child" rules stated in Section 3.2.

LWJs are the main means to provide group identifiers to WRAP services so that WRAP can manage any meaningful set of processes as one coherent object. An LWJ may use a WRAP service to relate itself to another LWJ simply by passing the target LWJ's identifier. This would be a common operation for tool LWJs as they often want to locate, synchronize with, and attach to MPI processes grouped through a compute LWJ. Similarly, WRAP may move a portion of the compute resources (e.g., maximum power use) that one LWJ has been using to another LWJ. That forms our basis to enable our elasticity model at the within-job scope.

8.1.1 WRAP Service Primitives

Table 2 defines the basic elements relevant to WRAP run-time services. For example, j represents an LWJ in the hierarchy within the job, and r_j denotes the compute resources to that this LWJ is confined. Using these as our foundational parameters, WRAP now defines the following service primitives to enable the new paradigm.

- $alloc(j, c_j)$: allocates r_j to j from $r_{parent(j)}$ according to c_j.
- $realloc(j, cnew_j)$: allocates $rnew_j$ from $r_{parent(j)}$ according to $cnew_j$ and updates r_j such that $r_j = r_j \cup rnew_j$.
- $release(j, subset(r_j))$: releases a subset of r_j to $r_{parent(j)}$ and updates r_j such that $r_j = r_j - subset(r_j)$.
- $contain(j, e_x)$: contains j in e_x.
- $launch(j)$: spawns, maps and binds processes of j on r_j according to c_j. If this is an incremental launch, this spawns and binds only additional processes.
- $destroy(j)$: kill processes of j running on r_j. If this is a partial destroy, this kills only a subset of processes.
- $bootstrap(j)$: bootstraps processes of j across r_j including dissemination of connection information. If this is a partial bootstrap–e.g., additional processes have been spawned or some processes have been killed, it only adjusts j for the change
- $split(j, marker)$: creates new child LWJs that includes all of the calling processes of j with the same marker.
- $record(j, d_j)$: records j's attributes such as its c_j, fingerprint for e_x, and arbitrary information (d_j) about j.
- $query(j)$: queries about j.
- $sync(j_k, j_l)$: putting an LWJ, j_k, into a known state and providing another LWJ, j_l, with j_k's identify info.

8.1.2 Higher-Level Services

Composing these primitives allow us to further build high-level services such as the following. The primitives and high-level operations are our conceptual tools to test WRAP services to a myriad of requirements and use cases of NGRM.

- $init(j, c_j) = < alloc(j, c_j), launch(j), bootstrap(j) >$
- $cont\_init(j, c_j, e_x) = < alloc(j, c_j), contain(j, e_x), launch(j), bootstrap(j) >$
- $grow(j, cnew_j) = < realloc(j, cnew_j), [launch(j), bootstrap(j)] >$, where *launch* and *bootstrap* are optional because they are only needed when *grow* needs to launch additional processes. For instance, if $cnew_j$ is a power bound increase request, these operations are unnecessary.
- $shrink(j, subset(r_j) = < release(j, subset(r_j)), [destroy(j), bootstrap(j)] >$, where *destroy* and *bootstrap* are optional because they are only needed when *shrink* needs to kill some processes of j.
- $monitor(j, j_{mon}, c_j) = < init(j, c_j), init(j_{mon}, c_j), sync(j, j_{mon}) >$, where the first *init* should be passed a special flag to cooperate with the subsequent *sync* operation.
- $log(j, j_{logger}, c_j) = < init(j, c_j), init(j_{logger}, c_j), sync(j, j_{logger}) >$, where the first *init* should be passed a special flag to cooperate with the subsequent *sync*.

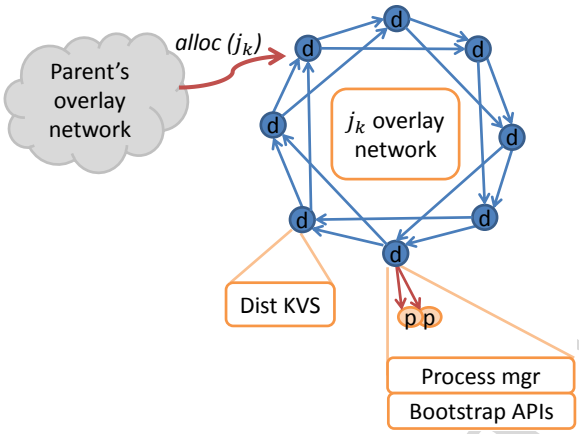

Figure 13: Base WRAP Architecture

8.1.3 Elasticity Support

The hierarchy of LWJs allows varying grain sizes of control for both process counts in sibling LWJs and resource distribution across these LWJs. This has many interesting properties. For example, a compute LWJ can further refine itself into smaller LWJs representing subsets of processes with independent resource confinement domains. As these smaller LWJs allocate, grow or shrink within the resource limit of the parent LWJ, diverse set of resources including consumable ones like power can be unevenly or evenly distributed across these LWJs. In the case of power management, LWJs that are not in the critical path in the parallel execution can reduce their power bound and return the leftover power resources to its parent. The parent LWJ can then use the returned resources in granting the *grow* requests that come from other LWJs that are in the critical path. This mechanism can support emerging power-aware computing that may want to cap power use differently and dynamically across different groups of MPI processes based on their execution patterns.

8.2 WRAP Software Architecture

In this section, we will detail our WRAP software architecture and mechanisms that are needed to implement the proposed WRAP services. We will first describe the base architecture that demonstrates the basic WRAP capabilities of executing an LWJ on a set of compute nodes, as our base resource type. Next, we will describe how we can extend this architecture to enable other advanced services such abilities to synchronize two independent LWJs, to grow compute node resources allocated to an existing LWJ, and to handle other types of resources such as power.

8.2.1 Base Architecture to Execute an LWJ

Figure 13 shows our base software architecture that can provide a single LWJ (j_k) with WRAP service primitives: *alloc*(), *launch*(), *bootstrap*(), *contain*(), *record*(), and *query*(). It assumes that the overlay network for *parent*(j_k) already exists and that this existing network can support highly scalable, resource-efficient communications for j_k upon granting the *alloc*(j_k) request. Our architecture requires that any overlay network instance recursively supports communications of a child LWJ either through an explicit instantiation of a separate overlay network or through the existing overlay network. However, the latter model requires to support a growth and/or reconfiguration of the existing network to be elastic. WRAP then uses the overlay network for j_k as well as a key-value store to serve scalable process management to j_k's processes. The following explains distributed key-value store mechanisms and key process management services in more detail.

Note that during the detailed design phase, WRAP and Comms Framework thrusts will co-design the actual communication mechanisms.

Distributed Key-Value Store (DKVS) provides a scalable mechanism by which the processes of j_k can share arbitrary information in a key-value pair amongst them. Conceptually, DKVS

represents a global key-value tuple space and any process of j_k can store its data by associating them with a unique key. To be memory-efficient, however, DKVS must store the data in a distributed manner. Thus, the overlay network of j_k must be capable of hashing the key to route its tuple to the home key-value store location. Global synchronization mechanisms such as collective fence will be provided to force memory consistency of DKVS across all processes.

Because our overlay network will have built-in routing schemes to support requisite distribution schemes, each home database itself can be a simple in-memory KVS. It is for this reason that we we will first consider a readily available, simple database implementation such as `Redis`. Depending on our overlay network topology, KVS can be fully distributed across all of the overlay network daemons. An example topology to support the full distribution is a "forest" with $log(N)$ connections wherein any daemon can be the root of a binomial tree.

Our DKVS will support a hierarchical tuple space for tighter data encapsulation per LWJ, which can further lead to a higher level of protection. More specifically, when an LWJ is allocated, a new name scope for that LWJ will be created in the DKVS, and information on the resource allocated to that LWJ will be stored as part of the default *record* service: e.g., j_k::resource $\rightarrow <$ core_count(1024), power_bound(100kw), ... $>$. This information is accessible by j_k as well as its immediate child LWJ j_{k+1}. Similarly, when a daemon creates an MPI rank process, it will add the personality of that process under this LWJ name scope such as j_k::rank(128) $\rightarrow <$ host IP, pid, executable path, ... $>$. Finally, DKVS will allow any process of the LWJ to store arbitrary information as part of $record(j_k, d_{j_k})$. All information stored with *record* will be pushed to $parent(j_k)$ when j_k is destroyed. Thus, the information will ultimately find its place in a persistent repository through the job hierarchy. In addition, DKVS will further support *query*, providing the calling process with underlying resource allocation information.

Scalable Process Management (ProcMan) Services can be implemented using the overlay network and DKVS as their basic scalable mechanisms. The scalable process management run-time services includes, but are not limited to, the following:

- **Bulk Process Creation and Stop**: The head daemon of the overlay network receives a process creation request through $launch(j_k)$ and propagates that command to the rest of the daemons in $O(log(N))$. Upon receiving the request, each daemon forks and execs local processes. If the request contains an optional `sync_assist` flag, the daemons stop the processes immediately after the creation to support a subsequent *sync*() issued by another LWJ. In either case, the daemons store information on the created target processes such as their process id, executable path and hostname, into the DKVS.

- **Scalable Propagation of Environments**: The head daemon receives the environment variables list and propagates that to the rest of the daemons in $O(log(N))$. The daemons then concatenate this master environment variables list to their local environment variables and export them to their processes. If the head daemon receives an optional *contain* request, that request is also being propagated. The specified containing-environment is used to contain the target processes.

- **Process Mapping, Binding and Confinement**: The daemons provide the newly created processes with topology information to support their mapping, binding and confinement to the underlying resources. The daemons retrieve the topology information from the key-value store.

- **Scalable `stderr/stdout` Handling**: The daemons receive `stderr` and `stdout` from their processes and scalably push and merge them through a tree in the overlay network towards the head daemon. Output aggregation techniques will include ways to reduce the output progressively at every merge step in the tree either by applying a readily available reduction filter or a user-provided one.

- **Scalable signal/`stdin` Forwarding**: The head daemon receives a UNIX signal or an input through `stdin` and scalably propagates it to a specified set of daemons through a tree in the

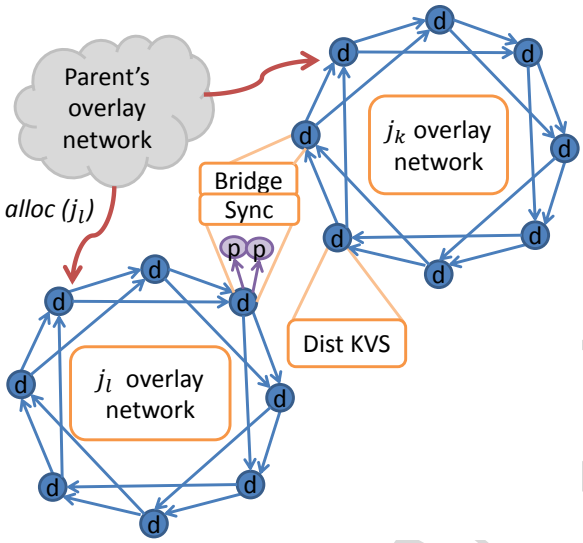

Figure 14: Architectural support for $sync(j_k, j_l)$

overlay network in $O(log(N))$. Upon receiving the signal or `stdin`, each daemon routes it to their corresponding processes.

- **Scalable Process Termination Detection and Analysis**: When one or more processes in the LWJ are normally or abnormally terminated, their home daemons detect the event and notify the head daemon through a tree in the overlay network. A time-out filter can be used at every step of the tree network to merge the return codes and, if abnormal, the stack traces of the terminated processes. Upon receiving the aggregated event, the head daemon will clean up the entire LWJ and also present to the users concise information about the termination.

A Unified Bootstrapping Mechanism will be used to ease integration of various types of LWJs. DKVS will be designed to support a wide range of existing bootstrap interfaces for distributed software including PMI 1 and 2 [11], PMGR Collective and COBO, LaunchMON [8] and LIBI [26]. With support for these interfaces, WRAP will be able to bootstrap various types of LWJs including a myriad of MPI implementations, tools communication infrastructure such as MRNet and also end-user tools such as STAT, TotalView and OpenSpeedShop. Specifically, the PMI layers will be a very thin layer on top of our DKVS implementation. We will support PMGR Collective, COBO, LaunchMON, and LIBI such that each created process opens up an ephemeral TCP port and store it as j_k::rank(128) $\rightarrow <$ host IP, port $>$. Then, each process in the binomial tree in these bootstrappers will simply find the connection information of its parent as well as its children using their ranks as the keys.

8.2.2 Architectural Support for Synchronizing LWJs

WRAP services must have an ability to relate an LWJ to another LWJ through *sync*. For example, a tool may need to be co-located with an MPI program and attached to its processes. Thus, we must extend the base architecture to support this concept. As shown in Figure 14, when $alloc(j_l)$ is granted for a new additional LWJ (j_l), the parent overlay network instantiates another overlay network for j_l and helps manage the processes of j_l in the same manner as the base case. To support $sync(j_k, j_l)$, however, a connection needs to be made between j_l's overlay and j_k's overlay network. For this purpose, we introduce the concept of bridge. The bridge allows processes of j_l to be able to access DKVS of j_k, which includes the mapping of the global MPI rank of a process of j_k to its host name, pid, executable path [18], information necessary for j_l to locate all of j_k's processes.

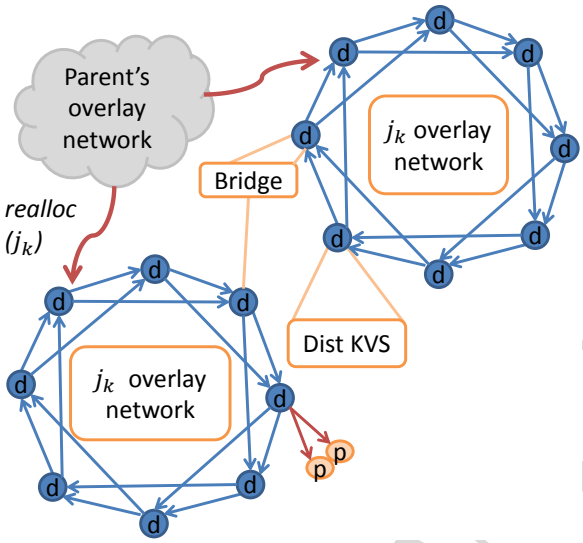

Figure 15: Architectural support for $grow(j, cnew_j)$

Alternatively, if the existing overlay network of j_k is capable of allocating and managing a new sibling LWJ within itself, the need for a connection between two overlay network and DKVS is eliminated.

8.2.3 Architectural Support for Growing Compute Node Resources

WRAP services must have an ability to grow the compute node resources that an LWJ (j_k) uses by reallocating an additional set of compute nodes from the resources allocated to $parent(j_k)$ and launching and bootstrapping additional processes across them. Figure 15 shows the architectural support for this elasticity. The scheme is essentially the same as that of *sync*. Upon granting a reallocation of $rnew_j$, $parent(j_k)$'s overlay network instantiates an overlay network across $rnew_j$ and connects it to the existing overlay network of j_k. The bridge support is again central for the connection. Additional processes launched through the new network will be able to access the DKVS associated with the existing j_k name scope. Conversely, DKVS associated with these additional processes will be made available to the existing j_k. Alternatively, if the existing overlay network of j_k is capable of allocating and managing these additional processes within itself, the need for a connection between two overlay network and DKVS is eliminated. However, the existing overlay network of j_k must be capable of growing and reconfiguring itself to join the additional compute nodes.

8.2.4 Architectural Support for Power as a Resource Type

As our *generalized resource model* will use the power bound as a resource type, WRAP services must have an ability to allocate, grow or shrink a power bound on an LWJ as well. As j will get the power bound allocated to it, a power capping mechanism like RAPL [45] can be used to set the power bound of the allocated compute nodes.

The basic scheme computes the per-node power-bound average and sets that average bound on each compute node. For more advanced power-aware computing, subsets of processes will be assigned to new "power capping domain" LWJs using *split*(). The new power capping LWJs will independently issue *grow*() and *shrink*() operations on the power bound as the resource type. This will allow the distribution of power bounds across the compute nodes to vary while still providing the overall power bound guarantee. These new LWJs will create new name scopes in the DKVS under their parent name scope: e.g., j_k::j_{k+1}::resource $\rightarrow$ < size(128), power_bound(12.5kW) >.

8.3 Component-Wise Work Breakdown

In this section, we summarize the requirements and work items for the major components of the proposed WRAP system.

Note that this sub-section is optional for reviewers to read, as it describes some work breakdowns. But I'm leaving this sub-section so that reviewers may gain further insight into what need to be implemented to support the proposed WRAP architecture.

Note that the network requirements still need to be reconciled with section 5.4.

8.3.1 Overlay Network and Infrastructure Requirements

The heart of WRAP lies in scalable and flexible overlay network support. The overlay network's attributes that WRAP requires include:

- an ability to support our elasticity model efficiently and scalably through expanding and shrinking compute nodes and LWJ processes launched on them;
- an ability for an arbitrary head daemon to broadcast or multicast data to a set of other daemons through a binomial or binary tree with $O(log(number\_of\_daemons))$;
- an ability for an arbitrary head daemon to aggregate data sent from a set of other daemons through the binomial or binary tree with $O(log(number\_of\_daemons))$;
- an ability to allow both preset and custom-made reduction operators to reduce the aggregated data along the tree;
- an ability to control data aggregation and reduction synchronously with a mechanism to set an arbitrary time-out threshold: zero time-out means pass-through; infinity time-out means global synchronization; and in-between means partial synchronization with varying degrees;
- an ability to route a key-value pair efficiently and scalably by automatically hashing its key to find its home DKVS server in support of get/put and global memory synchronization operations.

8.3.2 Process management

To support scalable process management service described in Section 8.2.1, the following components need to be investigated, designed and implemented:

- an extensible communication protocol that allows our process manager to conduct command and control for all of its services;
- a common data aggregation and reduction framework, techniques and API;
- process management service components that make use of the communication protocol and aggregation and reduction framework/API and expose the services through high-level APIs;
- executable commands such as an LWJ launcher that combines the service components to implement certain types of process management services for end users;
- topology-aware process binding and mapping components as well as communication mapping APIs that client software can use for efficient mappings between its communication patterns and the topology.

8.3.3 DKVS

DKVS requires the following components:

- name scoping specification and API;
- direct get/put methods on a readily-available key-value server;
- remote get/put methods on distributed key-value servers by integrating servers to the communication infrastructure;
- synchronization mechanisms and APIs that guarantee memory consistency across the distributed servers.

Work Item	Description	Dependency	Deliverable
Protocol design	design/prototype procMan command/control comm. protocol	comm. co-design	paper and review
Aggregation/reduction design	design/prototype aggregation reduction framework and APIs	comm. co-design	paper and review
DKVS name scoping design	design/prototype name scoping specification and APIs	comm. co-design	paper and review
Key-value store investigation	evaluate key-value store servers via direct put/get methods using bootstrap and MPIR emulation	none	finding summary
Power controller investigation	investigate Intel RAPL and ways to control power bound	RM design	paper and review

Table 3: Phase 1 BBB design milestones and deliverables

8.3.4 Bootstrap Interfaces

The following components should be implemented and demonstrated using DKVS support:

- PMI2 and port a version of MPICH as a reference implementation;
- PMGR Collective and/or COBO and port a version of MVAPICH as a reference implementation;
- LIBI and port a version of LIBI-enabled MRNet as a reference implementation.

8.3.5 Job Function Synchronization

- `MPIR_proctable` gatherer that gathers `MPIR_proctable` spread throughout the DKVS into a central location including the address space of the LWJ launcher;
- MPIR debug interface [18] that makes use of DKVS and process management services to support parallel debuggers;
- co-locator that co-locates an additional LWJ's processes with the target processes;
- LaunchMON Back-end API that makes use of DKVS to allow another LWJ processes to discover the locations of target processes scalably.

8.3.6 Power-Aware Computing

- *split()* that allows a compute LWJ to create smaller LWJs, and each serves as an independent power capping domain;
- Dynamic power bound controller that manages expanding and shrinking of power bounds across these LWJs.

8.4 Phase-Based Work Breakdown

To bring up the proposed WRAP system progressively and expediently, we use a phased approach. WRAP requires four phases: the outcome of earlier phases becomes the fundamental building blocks for the later phases.

Similarly to the previous sub-section, I'm leaving this sub-section to get some early feedback about our bring-up approach for WRAP.

8.4.1 Phase 1: Basic Building Blocks (BBB) design

During this phase, we will design and prototype the basic building blocks required by WRAP. This layer represents the lowest building blocks for the WRAP thrust and has fundamental dependencies on the overlay network infrastructure design as shown in Table 3. Thus, they must be co-designed.

Work Item	Description	Dependency	Deliverable
ProcMan design	design/prototype process manager package and APIs	comm. infra	paper and review
Topo-aware binding design	design/prototype topo-aware binding and mapping mechanisms and APIs	RM design	paper and review
Remote DKVS	investigate DKVS via remote put/get/sync using bootstrap and MPIR emulation	comm. infra	finding summary
BBB Implementation	implement ProcMan comm. protocol aggregation/reduction framework and API DKVS name scoping API direct key-value store power controller	BBB proto	software drop

Table 4: Phase 2 SBB design milestones and deliverables

Work Item	Description	Dependency	Deliverable
LWJ utility design	design/prototype LWJ utilities such as LWJ launcher	SBB proto	paper and review
LWJ sync design	design/prototype LWJ synchronizers	SBB proto	paper and review
PMI2, PMGR, LIBI design	design/prototype bootstrappers: PMI2, PMGR/COBO and LIBI	SBB proto	finding summary
SBB implementation	implement ProcMan package Topology-aware binding Remote DKVS	SBB proto	software drop

Table 5: Phase 3 USI design milestones and deliverables

8.4.2 Phase 2: Service Building Blocks (SBB) design and BBB implementation

During this phase, we will use the design and prototypes of basic building blocks to design and prototype higher-level service layer called service building blocks (SBB) packages. In addition, that effort will further validate the BBB design and prototypes and hence we will lock in the BBB design and produce production-quality BBB implementations during this phase. As shown in Table 4, this phase is designed to provide core WRAP functionality engine except for the actual user interfaces to expose.

8.4.3 Phase 3: User Service Interfaces (USI) design and SBB implementation

During this phase, we will use the design and prototype of service building blocks to design and prototype the user-visible service layer called user service interfaces (USI). In addition, once SBB prototypes are demonstrated that they are well-suited for the designed USI, we will lock in the SBB design and produce production-quality SBB implementations. As shown in Table 5, this phase will lay out the design of WRAP's external interfaces.

8.4.4 Phase 4: USI implementation

Once we demonstrate that USI prototypes sufficiently support both users and other run-times via reference implementations, we lock in the SUI design and produce production-quality USI implementations. Table 6 shows deliverables. The software drops will include the demonstration on client software such as MPI, MRNet and other run-time tools.

Work Item	Description	Dependency	Deliverable
LWJ utility implementation	implement LWJ utilities such as launcher	NGRM framework	software drop
LWJ sync implementation	implement LWJ LWJ synchronization	NGRM framework	software drop
PMI2, PMGR, LIBI design	implement PMI2, PMGR/COBO and LIBI	NGRM framework	software drop

Table 6: Phase 4 USI implementation

A Requirements

A.1 High Level Requirements

Scalable	Resource Manager will scale up to 100,000 compute nodes per cluster and will scale to many clusters to allow management of even the largest centers
Reliable	Resource Manager will not have a single point of failure and shall never require scheduled downtime for software upgrades. Fault tolerance will be incorporated at every level.
Secure	Resource Manager will support wire protocols with built-in privacy and data integrity.
Extensible	Resource Manager will support plugins with clean interfaces wherever possible to facilitate collaboration, customization, and novel functionality.
Research Friendly	Resource Manager design will incorporate features that allow experimentation and research analysis, including the ability to export sanitized logs, job data, and the ability to run experimental features within the Resource Manager framework.
Generalized	Resource Manager will attain maximal flexibility by abstracting resources as much as possible. i.e., a compute node is a pool of resources, a cluster is a pool of resources, a center is a pool of resources, where a resource can be a consumable or a collection of such consumables.
Integrated	Resource Manager will allow easy integration with other tools and frameworks, including monitoring, logging, and remote execution.

A.2 Architectural Components

Scheduler	The Scheduler manages the priorities of a queue of jobs requesting access to resources controlled by NGRM.
Resource Manager	Tracks resources available in the system and arbitrates access to these resources.
Remote Execution	Handles launch of processes across one or more resources managed by NGRM, including authorization, authentication and management of IO, environment, etc..
Monitoring/Logging	Manages collection and storage of monitoring and log data across the NGRM.
Provisioning	Manages root and other filesystem images across the NGRM.
Communications Network	Network channel through which all components of NGRM communicate.

A.3 High-Level Functional Requirements

1. **Zero Downtime**	
1.1.	Scheduled upgrades of NGRM components shall not require a downtime.
1.2.	The NGRM shall incorporate fault tolerance at every level, so that a failure of one component does not affect functionality of the system as a whole.
1.3	The NGRM shall support version interoperability, so that the deployed system as a whole is not required to be at the same level of software.
2. **Efficient Center-Wide Resource Management**	
2.1	The NGRM shall have the ability to function as a single instance managing all clusters within a network.
2.2	The NGRM shall be able to efficiently display a global view of resources to users when running in the mode described in R2.1.
2.3	The NGRM shall support the specification of generic resources such as GPUs, IO Bandwidth, Power, etc in addition to CPUs, Memory, and nodes.

2.4	The NGRM shall allow the hierarchy of resources to be specified in configuration or during discovery. For example, the definition of a node should allow the topology of that node to be recorded in the NGRM configuration (i.e. which CPUs are in which sockets, NUMA placement of memory, etc.) Similarly, there shall be the ability to record/discover the topology of nodes within a cluster, location and bandwith to IO storage from those nodes, access to and number of licenses, etc. FIXME: This needs rewording
2.5	The NGRM shall support allocation of "interactive" resources from the compute pool, allowing more intelligent and dynamic creation of "login" nodes
2.6	The NGRM shall support resource "tags" (similar to existing features) that can be applied to any resource, and a method to allow users requesting resources to select from tags
3. **Integrated Monitoring and Logging**	
3.0	NGRM monitoring and logging shall be designed to reduce system noise as much as possible
3.1	NGRM shall provide an integrated monitoring plugin API to allow system monitoring to be extended to handle future requirements.
3.2	NGRM shall provide the ability for users to tune the level of monitoring on the nodes of their jobs, allowing users to decrease monitoring levels and/or intervals for noise-sensitive jobs, or increase levels if they so choose.
3.3	NGRM system monitoring events on resources connected with a particular job (including file systems) shall be made available to users monitoring their job or in post-mortem reports or queries.
3.4	NGRM job log data shall have a public interface usable by tools such as sqlog to avoid duplication of data.
3.5	NGRM job log and system monitoring data shall be made available in sanitized form for use in job scheduling research, simulator testing of NGRM releases, etc.
3.6	NGRM job log and system monitoring data shall be collected, annotated, and saved to facilitate failure analysis.
3.7	NGRM shall provide a library that can be linked to user codes that will collect and store generic key/value pairs. This would replace LLNL's tracker tool and LANL's reportjob tools.
4. **Communication Network**	
4.1	NGRM components shall exchange messages and route log and monitoring data through a heirarchical, fault tolerant network.
4.2	NGRM network shall support messages, streaming data, and RPCs
4.3	NGRM network shall support privacy, integrity, and authentication
4.4	NGRM network shall export a API for use by tools and users to reuse established global and job-wide communication hierarchy.
5. **Remote Execution**	
5.1	NGRM shall provide a service for remote parallel execution across jobs for users as well as globally for administrators
5.2	NGRM remote execution service shall have the ability to manage instantiation of private namespaces for jobs
5.3	NGRM remote execution shall allow transport of all process environment attributes including environment variables, resource limits, namespace attributes, etc.
5.4	NGRM shall have the ability to bootstrap specialized environments such as a copy of NGRM itself (recursive execution), or an instance of Slurm (for backwards compatibility), or other frameworks such as Hadoop.
5.5	NGRM shall support launch of MPI jobs up to 100,000 nodes with an arbitrary number of tasks per node.
6. **Provisioning**	
6.1	NGRM shall separate user and system execution envioronments by running within a minimal "core" system root, and invoking all jobs within separate (possibly user-selected) root filesystem
6.2	NGRM shall provide service to allow users to run jobs with filesystem configuration of their choosing using private namespaces. The filesystems will need to be mounted without setuid for security purposes.

7. **User Interface**	
7.1	The NGRM shall export a rich API from which to build user interfaces. Command line as well as web based interfaces should be feasible with this API.
7.2	NGRM should support a centralized data store for management and analysis of system usage. The store should support highly concurrent and frequent accesses in as close to real time as possible without affecting job performance. A memory-based database supporting a pub/sub mechanism, such as Redis, would be ideal. (Jeff Long, Joel Martinez)
7.3	NGRM should provide an HTTP based REST API. Ideally the API would support a variety of different data formants (JSON, XML, etc...). The API should be implemented with a single-sign-on solution or API key implementation to avoid the need for additional user authentication. (Jeff Long, Joel Martinez)
8. **Scheduling**	
8.1	The Scheduler in the NGRM shall operate as a plugin or separate process or other easily replaceable component.
8.2	The Scheduler API shall have a clean interface such that the scheduler does not need to be upgraded/developed in lockstep with the NGRM core
8.3	The Scheduler interface in the NGRM shall allow the scheduling implementation access to all resources information gathered by the RM, including topology, heirarchy, data locality, IO bandwidth, etc.
8.4	When running in recursive mode (e.g. see 5.3), the user should be allowed to select from a list of alternate scheduler implementations or even provide their own.
8.5	NGRM scheduler shall support high job throughput
8.6	NGRM shall support overlapping resource partitions (queues) NGRM scheduler shall support complex job dependencies
9. **Site Integration**	
9.1	NGRM must implement fair-share scheduling, and provide a mechanism for administrators to set usage targets by fair-share account and user. (Greg Tomaschke)
9.2	For utilization reporting, NGRM must recognize four mutually exclusive states for resources: allocated, reserved, idle, and down. NGRM must provide an interface to report the time spent in each of these states, for any given set of resources over any given period of time. (Greg Tomaschke)
9.3	NGRM must provide a means for users to associate each job with a project ID, independent of their fair-share account. A user may have a default project ID, and the capability to override it at job submission. (Greg Tomaschke)
9.4	NGRM must provide an interface to report the time used by all jobs, broken down by fairshare-account, user, project ID, assigned resources, or any combination thereof, over any given period of time. (Greg Tomaschke)
9.5	NGRM must provide a means to dump a record of all jobs that have executed on the system, including any state transitions of assigned resources, and RAS events that occurred during execution. (Greg Tomaschke)

A.4 Use Cases

UC1	**Use NGRM recursive execution to manage dedicated application test time** Currently a DAT (dedicated application time) is managed by draining an entire cluster and then "giving" the test team access to the cluster via support staff with expedite privileges. With NGRM recursive execution, the DAT could be submitted to the RM as a job with constraints to run on all or part of a cluster. Once the job has been allocated on the cluster, team members would instantiate interactive instances to the job, and the recursive feature of NGRM would make it appear as if they had access to an empty cluster. Jobs could then be submitted to this instance from within the original NGRM job or the users' interactive instances. This scenario is also useful for testing new versions of the NGRM or other system software.

UC2	**Sysadmin ability to "drain" only a portion of the resource hierarchy** Currently, the resource management "drain" functionality is limited to a single node. Consider a case where a resource within a node goes bad, e.g. a GPU. Since the rest of the node is fine, the NGRM should allow just the GPU resource to be "drained" and the rest of the node is usable for compute jobs that do not need a GPU resource. When draining a full node or group of nodes, all the resources within the hierarchy of those nodes should also be drained. This would also allow an entire cluster or subset of resources to be drained by draining at the top level "cluster" resource.
UC3	**Power utilization as a resource** As a datacenter manager I want to deploy a cluster and provision less power to it than the max theoretical peak. The NGRM should allow the total available power to be specified at the cluster and/or PDU level within the cluster, and should treat power as a consumable resource. The NGRM will ensure that the set of resources use only the power that has been budgeted. If realtime power utilization is available, then those values can be aggregated and used (though this may not be safe when applications with fluctuating power demands are running), otherwise the NGRM should allow some kind of power utilization model to be registered (e.g. 90W per cpu + 100W per GPU + 200W base utilization per node...)
UC3.1	**Energy usage part of job report** The total energy consumed by a job should be reported upon job completion. (Barry Rountree)
UC3.2	**Track power efficiency of components** CPU's have varying power efficiency. Periodically measure this and record in resource db to be combined with a static power model. Also, scheduler could use this info to schedule the fastest nodes to the critical path. (Barry Rountree)
UC3.3	**MPI runtime able to set power MSRs** The MPI runtime can identify slow ranks and adjust power clamping to gain more performance. Application-driven variation in power consumption must not exceed breaker trip levels in a system plumbed for less than peak power consumption. (Barry Rountree)
UC3.4	**Provide ability for job to specify power ranges** User should be able to specify the maximum (and minimum?) power their job will consume. Scheduler will only schedule job when it can configure the resources to remain within the specified power envelope. (Barry Rountree)
UC3.5	**Node vs. Time Scheduling becomes Power vs. Time** See M. Etinski paper [21] (Barry Rountree)
UC3.6	**Limiting Site Power Swings** NGRM must provide the ability to constrain power usage ramp-up and ramp-down rates to meet facilities requirements. (Trent D'Hooge)
UC4	**Generic aggregate resources** As a generic case of UC3, imagine a resource that is distributed across nodes or a whole cluster. A contrived example might be bandwidth to a SAN or other storage pool. The NGRM should have the ability to account for, manage, and schedule such generic resources in a way that is extensible to unforeseen resources, since we cannot imagine all possibilities beforehand. The NGRM should offer the ability to develop plugins or helpers to enforce and monitor and display these aggregate resources.
UC5	**Center-wide cron** As a sysadmin I may want to schedule a periodic job to run on systems of a certain type, so the NGRM should have cron-style capabilities at the center level. A cron "job" should be registered in the queue or in configuration that is run by the NGRM on the specified interval, on nodes matching the resource constraints specified in the "job". Cron work scheduled on compute nodes, such as required NAPS tests, should be run through NGRM such that they do not create OS jitter for running jobs. For example, they could only run between jobs or if they must run during a job, they could run synchronized. Users should be able to submit cron jobs too (at least some users like hotline do this).

UC6	**Live user feedback for job progress** As a user I would like access to resource manager data live while my job is running as a sort of job progress indicator. It would be useful to have IO statistics (perhaps with built-in IO Watchdog functionality), power utilization, network bandwidth, and the ability to register handlers if no progress is detected (by my own definition of no progress detected).
UC7	**Allow users to inject application specific data into data stream for jobs** As a user it would be really useful if I could inject application specific data into the data stream for a job. The job database could then keep this data in perpetuity, instead of having me keep data about my runs in an ad-hoc fashion. Data should be free form to allow for the greatest usability. Examples might be name of code, run ID, and iterations as they occur with timestamps. This would be useful in conjunction with UC6 as well.
UC8	**dsh—dshbak** As suggested by req. 5.1, pdsh—dshbak would be submitted via NGRM. There should be a way to limit concurrency (like pdsh fanout), a way to select specific hostnames (like pdsh wcoll), and a way to submit the dshback such that output is reduced in a distributed fashion (e.g. on intermediate nodes of the comms tree).
UC9	**User control over system software levels** As a user, for testing or reproducible results, I want to the ability to rerun a job or set of new jobs under a previously supported level of system software. I do not want to be required to record important system software versions manually, so the resource manager should track this data for every run and keep a record in a historical database so that I can go back and determine what software level was running when any one of my jobs ran.
UC10	**Testing system software releases** Major system sofware releases (e.g. TOSS major releases that break binary compatibility with old releases) should be testable in advance of being configured as the default through NGRM option at run time.
UC11	**Integrated tool support** NGRM should be able to efficiently launch and tear down STAT, totalview, and other distributed tools with a presence on compute nodes. NGRM should provide hooks for those tools so they can avoid reimplementing NGRM features. (See also MPI 3.0 Tools Support WG de Supinski, Schulz)
UC12	**Allocate spare resources from a common pool** Support a "spare resources" model whereby fault tolerant MPI jobs could dynamically pull in replacement nodes or other resources from the common pool rather than allocating spare resources privately at job submission.
UC13	**Checkpoint/restart** Support user-level checkpoint/restart such that NGRM can signal a job to begin checkpointing, receive an acknowledgement when checkpointing is complete, manage the checkpoint data (ensure it is on stable storage), clear the job from the system, then at a later time, stage in the checkpoint data and restart the job. (See also BLCR, SCR, OpenMPI c/r).
UC14	**Ephemeral file system instances** Allow user to request a dedicated parallel file system to be instantiated for the life of their job. NGRM could allocate disks from pool of storage resources, optionally stage data on/off of this file system at setup/tear-down.
UC15	**Integrated I/O forwarding support** Set up and tear down I/O forwarding daemons for exclusive use by a job. Allow flexibility in the number and placement of such daemons and their mapping to compute nodes. Collect I/O statistics from daemons and make available as part of job monitoring stream/records.
UC16	**Hadoop framework** NGRM should be able to launch hadoop framework on allocated resources (including storage).

UC17	**User database instances** NGRM should support starting a database server such as MySQL on allocated nodes/storage, loading a data set, and running a series of jobs that use the database, in some way telling the jobs how to connect to the server, and managing access.
UC18	**Detect and report out of spec components** As suggested by req. 3.6, NGRM should make it easy to detect when hardware / software components are performing out of spec or isolate components common to failed jobs.
UC19	**Record HW configuration** NGRM should record known HW configurations for each job executed. The data should have an schema-less, self-describing format to facilitate future previously unknown hardware configurations.
UC20	**Provide "pre-job" resource information** Users or admins should be able to query resource information such as network topology, I/O bandwidth, CPU speed, node memory, etc. This allows for users to alter jobs for "optimum" execution. For example, MPI communications algorithms or requested node allocation could be changed based on network topology.
UC21	**Virtual private networks for jobs** It should be possible to run a job in a virtual private network environment separate from that of the NGRM software, such that the job's network access is restricted to resources appropriate for it to access. This could be used to manage access to private services, such as the database in UC17. It might also be possible to organize user communities and their file stores such that the same protection offered by a firewall could be achieved without partitioning stateless compute resources.
UC22	**Verbose Logging Triggered by Fault Event** The monitoring system should facilitate capturing verbose debug logs prior to a fault event. One way to do this is with a circular trace buffer that is dumped into the monitoring stream on notification of a fault. (See "self propelled instrumentation", e.g. paradyn project) (Kathryn Mohror, Don Lipari)
UC23	**I/O Staging** The user should be able to specify files at job submission time that will be moved to storage close to compute nodes in advance of job launch. Files could be directed to be moved the other direction after job termination. This should be done in a way that does not impact other jobs, and can occur concurrently with allocating/freeing other job resources that aren't involved in moving files (Kathryn Mohror)
UC23	**Give me a fat node for rank 0 and top it up with thin or fat ones** In sbatch to ask for a fat node (more memory) and then a couple of other nodes (fat or thin), and then have the batch script start on the fat node, so that rank 0 is placed there by default by the MPI. (Kent Engstrom request to slurm-dev)

References

[1] Classad language reference manual. Retrieved Dec 14, 2012 from `http://research.cs.wisc.edu/htcondor/classad/refman/`.

[2] The Globus resource specification language RSL v1.1. `http://toolkit.globus.org/toolkit/docs/5.2/5.2.5/gram5/developer/#gram5%-rsl`. Accessed: 02-25-2013.

[3] Lua sand boxes. Retrieved Feb 22, 2013 from `http://lua-users.org/wiki/SandBoxes`.

[4] Nagios. Retrieved Feb 10, 2013 from `http://www.nagios.org`.

[5] Coordinated infrastructure for fault tolerant systems, fault tolerance backplane (FTB) API version 0.5, document revision 0.3. Interface specificiation, The CIFTS Group, Aug 2010.

[6] MRNet API programmer's guide, release 4.0.0. Interface specification, Paradyn Tools Project, Computer Sciences Department, University of Wisconsin, Madison, April 2012.

[7] Moab workload manager version 7.2.6. Administrator guide, Adapting Computing, 2014.

[8] D. H. Ahn, D. C. Arnold, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Overcoming scalability challenges for tool daemon launching. In *Proceedings of the 37th International Conference on Parallel Processing*, pages 578–585, 2008.

[9] J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc., 1st edition, 2010.

[10] D. C. Arnold and B. P. Miller. A scalable failure recovery model for tree-based overlay networks. Technical Report TR1626, Computer Sciences Department, University of Wisconsin, 2008.

[11] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur. PMI: A scalable parallel process-management interface for extreme-scale systems. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, EuroMPI'10, pages 31–41, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] S. M. Balle and D. J. Palermo. Enhancing an open source resource manager with multi-core/multi-threaded support. In *JSSPP*, pages 37–50, 2007.

[13] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, CCGRID '05, pages 776–783, Washington, DC, USA, 2005. IEEE Computer Society.

[14] R. H. Castain, W. Tan, J. Cao, M. Lv, M. Jette, and D. Auble. MapReduce support in SLURM: releasing the elephant. Retrieved from `http://schedmd.com/slurmdocs/slurm_ug_2012/MapRedSLURM.pdf`, 2012.

[15] S. Castano, A. Ferrara, S. Montanelli, and G. Racca. Matching techniques for resource discovery in distributed systems using heterogeneous ontology descriptions. In *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 1, pages 360 – 366 Vol.1, april 2004.

[16] S. J. Chapin, D. Katramatos, J. Karpovich, and A. S. Grimshaw. The Legion resource management system. In *Proceedings of the 5 th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 162–178. Springer Verlag, 1999.

[17] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[18] J. DelSignore, D. Ahn, R. Castain, and J. Squyres. The MPIR process acquisition interface version 1.0. Interface specification, MPI Forum Working Group on Tools, 2010.

[19] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), Mar. 1997. Updated by RFCs 3396, 4361, 5494.

[20] C. Dunlap. MUNGE uid 'n' gid emporium. Retrieved Nov. 9, 2012 from `http://code.google.com/p/munge/`.

[21] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Optimizing job performance under a given power constraint in HPC centers. *2010 International Green Computing Conference*, 2010.

[22] Freedesktop.org. D-Bus. Retrieved Feb. 26, 2013 from `http://www.freedesktop.org/wiki/Software/dbus`.

[23] Freedesktop.org. Systemd. Retrieved Feb. 26, 2013 from `http://www.freedesktop.org/wiki/Software/systemd`.

[24] T. Gamblin and K. Mohror. Leveraging log analytics to understand application I/O. Proposal for Laboratory Directed Research and Development LLNL-PROP-559051, Lawrence Livermore National Laboratory, 2012.

[25] R. Gerhards. The Syslog Protocol. RFC 5424 (Proposed Standard), Mar. 2009.

[26] J. D. Goehner, D. C. Arnold, D. H. Ahn, G. L. Lee, B. R. De Supinski, M. P. Legendre, B. P. Miller, and M. Schulz. LIBI: A framework for bootstrapping extreme scale software systems. *Parallel Comput.*, 39(3):167–176, Mar. 2013.

[27] Google, Inc. Protocol buffers. Retrieved Nov. 16, 2012 from `http://api.developers.google.com/protocol-buffers`.

[28] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782 (Proposed Standard), Feb. 2000. Updated by RFC 6335.

[29] R. Gupta, P. Beckman, B. H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra. CIFTS: A coordinated infrastructure for fault-tolerant systems. *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2009.

[30] S. Hanna, B. Patel, and M. Shah. Multicast Address Dynamic Client Allocation Protocol (MADCAP). RFC 2730 (Proposed Standard), Dec. 1999.

[31] P. Hintjens. ØMQ - the guide, updated for version 3.2. Retrieved Nov. 5, 2012 from `http://zguide.zeromq.org/page:all`.

[32] R. Ierusalimschy. *Programming in Lua, Second Edition*. Lua.Org, 2006.

[33] M. Jette and M. Grondona. SLURM: Simple linux utility for resource management. *ClusterWorld Conference and Expo*, 2005.

[34] A. Keller, A. Reinefeld, and E. Reinefeld. CCS resource management in networked HPC systems. In *In Proc. of Heterogenous Computing Workshop HCW98 at IPPS*, pages 44–56. IEEE Computer Society Press, 1998.

[35] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401 (Proposed Standard), Nov. 1998. Obsoleted by RFC 4301, updated by RFC 3168.

[36] R. Koning, P. Grosso, and C. de Laat. Using ontologies for resource description in the cinegrid exchange. *Future Gener. Comput. Syst.*, 27(7):960–965, July 2011.

[37] G. P. Koslovski and P. V.-b. Primet. VXDL: Virtual resources and interconnection networks description language. *Networks for Grid Applications*, 2:1–17, 2009.

[38] P. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.

[39] A. Natrajan, M. A. Humphrey, and A. S. Grimshaw. Grid resource management in Legion. 2004.

[40] A. Pernas and M. Dantas. Ontology based service for grid resources description. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 1, pages 154 – 159 Vol. 1, may 2005.

[41] E. Plugge, T. Hawkins, and P. Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA, USA, 1st edition, 2010.

[42] A. Rabkin and R. H. Katz. Chukwa: A system for reliable large-scale log collection. Master's thesis, EECS Department, University of California, Berkeley, Mar 2010.

[43] N. Regola and J.-C. Ducom. Recommendations for virtualization technologies in high performance computing. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 409–416, Washington, DC, USA, 2010. IEEE Computer Society.

[44] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 21–, New York, NY, USA, 2003. ACM.

[45] B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. Schulz. Beyond DVFS: A first look at performance under a hardware-enforced power bound. In *IPDPS Workshops*, pages 947–953. IEEE Computer Society, 2012.

[46] T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, and L. Vicisano. PGM Reliable Transport Protocol Specification. RFC 3208 (Experimental), Dec. 2001.

[47] W. Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Third edition, 1999.

[48] R. Stewart, M. Tuxen, and P. Lei. SCTP: What is it, and how to use it? *BSDCan 2008: The Technical BSD Conference*, 2008.

[49] J. Van Der Ham, F. Dijkstra, P. Grosso, R. Van Der Pol, A. Toonk, and C. De Laat. A distributed topology information system for optical networks based on the semantic web. *Opt. Switch. Netw.*, 5(2-3):85–93, June 2008.

[50] J. van der Ham, P. Grosso, F. Dijkstra, and C. T. de Laat. Semantics for hybrid networks using the network description language. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[51] J. J. van der Ham, F. Dijkstra, F. Travostino, H. M. A. Andree, and C. T. A. M. de Laat. Using RDF to describe networks. *Future Gener. Comput. Syst.*, 22(8):862–867, Oct. 2006.

[52] Wikipedia. Folksonomy — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Folksonomy&oldid=539451265`, 2013. [Online; accessed 26-February-2013].

[53] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose. Performance evaluation of container-based virtualization for high performance computing environments. *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2013.