Architectures, Design Methodologies, and Service Composition Techniques for Grid Job and Resource Management

Per-Olov Östberg

LICENTIATE THESIS, SEPTEMBER 2009
DEPARTMENT OF COMPUTING SCIENCE
UMEÅ UNIVERSITY
SWEDEN

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

p-o@cs.umu.se

ISBN 978-91-7264-861-6
ISSN 0348-0542
UMINF 09.15

Printed by Print & Media, Umeå University, 2009

Abstract

The field of Grid computing has in recent years emerged and been established as an enabling technology for a range of computational eScience applications. The use of Grid technology allows researchers and industry experts to address problems too large to efficiently study using conventional computing technology, and enables new applications and collaboration models. Grid computing has today not only introduced new technologies, but also influenced new ways to utilize existing technologies.

This work addresses technical aspects of the current methodology of Grid computing; to leverage highly functional, interconnected, and potentially under-utilized high-end systems to create virtual systems capable of processing problems too large to address using individual (supercomputing) systems. In particular, this thesis studies the job and resource management problem inherent to Grid environments, and aims to contribute to development of more mature job and resource management systems and software development processes. A number of aspects related to Grid job and resource management are here addressed, including software architectures for Grid job management, design methodologies for Grid software development, service composition (and refactorization) techniques for Service-Oriented Grid Architectures, Grid infrastructure and application integration issues, and middleware-independent and transparent techniques to leverage Grid resource capabilities.

The software development model used in this work has been derived from the notion of an ecosystem of Grid components. In this model, a virtual ecosystem is defined by the set of available Grid infrastructure and application components, and ecosystem niches are defined by areas of component functionality. In the Grid ecosystem, applications are constructed through selection and composition of components, and individual components subject to evolution through meritocratic natural selection. Central to the idea of the Grid ecosystem is that mechanisms that promote traits beneficial to survival in the ecosystem, e.g., scalability, integrability, robustness, also influence Grid application and infrastructure adaptability and longevity.

As Grid computing has evolved into a highly interdisciplinary field, current Grid applications are very diverse and utilize computational methodologies from a number of fields. Due to this, and the scale of the problems studied, Grid applications typically place great performance requirements on Grid infrastructures, making Grid infrastructure design and integration challenging tasks. In this work, a model of building on, and abstracting, Grid middlewares has been developed and is outlined in the papers. In addition to the contributions of this thesis, a number of software artefacts, e.g., the Grid Job Management Framework (GJMF), have resulted from this work.

Preface

This thesis consists of a brief introduction to the field, a short discussion of the main problems studied, and the following papers.

Paper I E. Elmroth, P. Gardfjäll, A. Norberg, J. Tordsson, and P-O. Östberg. Designing General, Composable, and Middleware-Independent Grid Infrastructure Tools for Multi-Tiered Job Management. In T. Priol and M. Vaneschi, editors, *Towards Next Generation Grids*, pages 175–184. Springer-Verlag, 2007.

Paper II E. Elmroth, F. Hernández, J. Tordsson, and P-O. Östberg. Designing Service-Based Resource Management Tools for a Healthy Grid Ecosystem. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science, vol. 4967*, pages 259–270. Springer-Verlag, 2008.

Paper III E. Elmroth and P-O. Östberg. Dynamic and Transparent Service Compositions Techniques for Service-Oriented Grid Architectures. In S. Gorlatch, P. Fragopoulou, and T. Priol, editors, *Integrated Research in Grid Computing*, pages 323–334. Crete University Press, 2008.

Paper IV E. Elmroth, S. Holmgren, J. Lindemann, S. Toor, and P-O. Östberg. Empowering a Flexible Application Portal with a SOA-based Grid Job Management Framework. In *The 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, to appear, 2009.

Paper V E. Elmroth and P-O. Östberg. A Composable Service-Oriented Architecture for Middleware-Independent and Interoperable Grid Job Management. *UMINF 09.14*, Department of Computing Science, Umeå University, Sweden. Submitted for Journal Publication, 2009.

This research was conducted using the resources of the High Performance Computing Center North (HPC2N). Financial support has been provided by The Swedish Research Council (VR) under contract 621-2005-3667.

Acknowledgements

A number of people have directly or indirectly contributed to the work in this thesis and deserve acknowledgement. First of all, I would like to thank my advisor Erik Elmroth for not only the opportunities provided and all the hard work, but also for the positive environment he creates in our research group. I would also like to thank my coadvisor Bo Kågström for inspiring discussions and the unique perspective he brings to them. Among my colleagues in the GIRD group I would like to thank Lars Larsson and Johan Tordsson for lengthy discussions of all things more or less related to our work, and Francisco Hernández, Daniel Henriksson, Raphaela Bieber-Bardt, Arvid Norberg, and Peter Gardfjäll (in no particular order) for all their contributions to our collective effort. Among our research partners I would like to thank Sverker Holmgren, Jonas Lindemann, and Salman Toor for interesting collaborations, and the support staff of HPC2N for their contributions and knowledge of the Grid systems we use. Finally, on a personal level I would like to thank my family and friends, without whom none of this would be possible, for all the love and support they provide.
Thank you all.

Contents

Chapter 1

Introduction

In the past decade, Grid computing has emerged and been established as an enabling technology for a range of computational eScience applications. A number of definitions of Grid computing exist, e.g., [33, 37, 64], and while the scientific community has reached a certain level of agreement on what a Grid is [58], best practices for Grid design and construction are still topics for investigation. The definition used in this thesis details Grid computing to be a type of distributed computing focused on aggregation of computational resources for creation of meta-scale virtual supercomputers and systems.

As a paradigm, Grid computing revolves around concepts such as service availability, performance scalability, virtualization of services and resources, and resource (access) transparency [40, 58]. The current methodology of the field is to leverage interconnected high-end systems to create virtual systems capable of great performance scalability, high availability, and collaborative resource sharing [37]. The approach taken in this work employs loosely coupled and decentralized resource aggregation models, assumes resources to be aggregated from multiple ownership domains, and expects all Grid services and components to be subject to resource contention, i.e. to coexist with competing mechanisms.

Grid technology and infrastructure have today found application in fields as diverse as, e.g., life sciences, material sciences, climate studies, astrophysics, and computational chemistry, making Grid computing an interdisciplinary field. Current Grid applications occupy all niches of scientific computation, ranging from embarrassingly parallel high-throughput applications to distributed and synchronized data collation and collaboration projects.

Actors within, and contributions to, the field of Grid computing can broadly be segmented into two main categories; application and infrastructure. Grid applications often stem from independently developed computational methodologies more or less suited for use in Grid environments, and are often limited (in Grid usage scenarios) by how well their methodology lends itself to parallelization. Motivations for migration to Grid environments vary, but often

include envisioned performance benefits or synergetic collaboration effects.

Typically, Grids are designed to provide a level of scalability beyond what is offered by individual supercomputer systems. System requirements vary with Grid application needs, and usually incorporate advanced demands for storage, computational, or transmission capacity, which places great performance requirements on underlying Grid infrastructure at both component and system level. These conditions, combined with typical interdisciplinary requirements of limited end-user system complexity, automation, and high system availability, make Grid infrastructure design and resource federation challenging tasks.

The focus of this thesis lies on research questions related to Grid infrastructure and application design, with emphasis on job and resource management issues. In particular, abstraction of Grid job management interfaces, and related application and infrastructure component integration issues have been studied in the context of federated Grid environments. The methodology of this work includes investigation of architectural design patterns inspired by the notion of an ecosystem of Grid infrastructure components [62], and exploration of Service-Oriented Architecture (SOA)-based [51] implementation techniques. The concept of an ecosystem of Grid infrastructure components, where applications are composed through selection of software from an ecosystem of components, and individual components are subject to meritocratic natural selection, is further described in Section 5.

Two of the overarching goals of the GIRD [63] project, in which this research has been performed, are to investigate and propose architectures for abstraction and provisioning of Grid functionality, and to provide proof-of-concept implementations of proposed architectures. Scientific contributions of this thesis include investigation of architectural design patterns, development of Grid infrastructure task algorithms, and contributions to formulation of design methodologies for scalable Grid infrastructure and application components.

The rest of this thesis is structured as follows. Section 2 provides an introduction to Grid applications and outlines a few of the requirements Grid applications impose on Grid infrastructures and environments. Section 3 discusses Grid infrastructure and covers some of the trade-offs involved in Grid infrastructure design and development. Section 4 provides an overview of the Grid job and resource management problem, and structures the area into constituent processes while briefly referencing some of the work within the field. Section 5 sketches the notion of an ecosystem of Grid components, and serves here as an introduction to the perspective chosen in this work. Section 6 summarizes the contributions of this thesis, and relates the thesis papers to each other. Finally, Section 7 outlines some future directions for this work, and references current efforts related to the work of this thesis.

Chapter 2

Grid Applications

Utilization of Grid technology affords the scientific community to study problems too large to address using conventional computing technology. Use of Grids has resulted in creation of new types of applications and new ways to utilize existing computation-based technology [37]. Grid applications can based on application requirements be segmented into categories such as

- computationally intensive, e.g., interactive simulation efforts such as the SIMRI project [11], and very large-scale simulation and analysis applications such as the Astrophysics Simulation Collaboratory [55].
- data intensive, e.g., experimental data analysis projects such as the European Data Grid [56], and image and sensor analysis applications such as SETI@home [3].
- distributed collaboration efforts, e.g., online instrumentation tools such as ROADnet [45], and remote visualization projects such as the Virtual Observatory [66].

Based on computational requirements and topology of the application, computational Grid applications can broadly be classified as High Performance Computing (HPC), High Throughput Computing (HTC), or hybrid approaches. HPC applications are generally concerned with system peak performance, and measure efficiency in the amount of computation performed on dedicated resource sets within limited time frames. Computations are in HPC applications typically structured to maximize application computational efficiency for a particular problem, e.g., through Message Passing Interface (MPI) [57] frameworks. HTC applications are conversely focused on resource utilization and measure performance in the amount of computation performed on shared resource sets over extended periods of time, e.g., in tasks per month. Computationally, HTC applications are generally composed of large numbers of (small) independent jobs running on non-dedicated resource sets without real-time constraints for result delivery. A number of hybrids between the HPC and HTC

paradigms exist, e.g., the more recently formulated Many Task Computing (MTC) [54] paradigm. MTC applications focus on running large amounts of tasks over short periods of time, are typically communication-intensive but not naturally expressed using synchronized communication patterns like MPI, and measure performance using (application) domain-specific metrics.

Beside obvious computational requirements, Grid applications typically also impose advanced system performance requirements for, e.g.,

- storage capacity: Grid applications potentially process very large data sets, and often do so without predictable access patterns.
- data transfer capabilities: Grid computations are typically brokered and may be performed far from the original location for input data and application software. Efficient data transfer mechanisms are required to relocate data to computational resources, and return results after computation.
- usability: Grid interfaces abstract resource system complexity and use of underlying computational resources to improve system usability and lower learning requirements.
- scalability: Grid application system requirements are likely to vary during application runtime, requiring underlying systems and infrastructure to access and scale computational, storage, and transfer capabilities on demand.
- availability: Grid applications and systems are typically composed through aggregation of computational resources, allowing Grids to exhibit very high levels of system availability despite system capacity varying over time due to resource volatility issues. Consistent levels of system access and quality of service improve the perception of Grid availability and stability.
- collaboration: Grid applications and systems support levels of collaboration ranging from multiple users working on shared data to multiple organizations utilizing shared resources.

System complexity and the great demands and different requirements of current Grid applications have led to the emergence of two major types of Grids; computational Grids and data Grids. Typically, computational Grids focus on providing abstracted views of computational resource access, and address very large computational problems. Data Grids conversely focus on providing virtualization of data storage capabilities and provide non-trivial and scalable qualities of service for very large data sets. A number of additional Grid system subtypes have also emerged, e.g., collaboration Grids, enterprise Grids, and cluster Grids [58]. The work in this thesis has been focused on systems designed for use in computational Grid environments.

From a performance perspective, the construction of Grid systems is facilitated by improvements in computational and network capacity, and motivated by general availability of highly functional and well connected end systems. Increase in network capacity alone has lead to changes in computing geometry and geography [37], and technology advances have today made massive-scale collaborative resource sharing not only feasible, but approaching ubiquitous.

From an application perspective, Grid computing holds promise of more efficient models for collaboration when addressing larger and more complex problems, less steep learning curves (as compared to traditional high-performance computing), increased system utilization rates, and efficient computation support for broader ranges of applications. While Grids of today have achieved much in system utilization, scalability and performance, much work in reducing system complexity and increasing system usability still remain [58].

Chapter 3

Grid Infrastructure

The name Grid computing originated from an analogy in the initial guiding vision of the field; to provide access to the capabilities of computational resources in a way similar to how power grids provide electricity [37], i.e. with transparency in

- resource selection (i.e. which resource to use).
- resource location (i.e. with transparency in resource access).
- resource utilization (i.e. amount of resource capacity used).
- payment models (i.e. pay for resource utilization rather than acquisition).

In this vision, the role of Grid infrastructure becomes similar to that of power production infrastructure: to provide capacity to systems and end-users in cost-efficient, transparent, federated, flexible, and accessible manners. While application and user requirements on Grids vary greatly, and can be argued to be more complex than those of power infrastructure, the analogy is apt in describing a federated infrastructure providing flexible resource utilization models and consistent qualities of service through well-defined interfaces.

To realize a generic computational infrastructure capable of flexible utilization models, it is rational to build on standardized, reusable components. The approach of this work is to identify and isolate well-defined Grid functionality sets, and to design interfaces and architectures for these in manners that allow components to be used as building blocks in construction of interoperable Grid applications and systems [24, 26]. In an analogy to efforts within related fields, the role of generic Grid components could be compared to the role of, e.g., frameworks such as Linear Algebra PACKage (LAPACK) [5] libraries in numerical linear algebra. Similarly, the role of application integration components could be compared to the role of Basic Local Alignment Search (BLAST) [60] toolkits in bioinformatics, and component interfaces to Basic Linear Algebra Subprograms (BLAS) [12] Application Programmer Interfaces (APIs).

From a systems perspective, Grid computing revolves around concepts such as (performance) scalability, virtualization, and transparency [40]. Performance scalability here refers to the ability of a system to dynamically increase the computational (or storage, network, etc.) capacity of the system to meet the requirements of an application on demand. Virtualization here denotes the process of abstracting computational resources, a practice that can be found on all levels of a Grid. For example, Grid applications' use of infrastructure is often abstracted and hidden from end-users, Grid systems and infrastructure typically abstract the use of computational resources from the view of applications, and access to Grid computational resources is typically abstracted by native resource access layers, e.g., batch systems. The term transparency is used to describe that, like access to systems and system components, scalability should be automatic and not require manual efforts or knowledge of underlying systems to realize access to, or increase in, system capacity.

Typically today, performance scalability is achieved in Grid systems through dynamic provisioning of multiple computational resources over a network, virtualization through interface abstraction mechanisms, and transparency through automation of core Grid component tasks (such as resource discovery, resource brokering, file staging, etc.).

To facilitate flexible resource usage models, Grid users and resource allotments are typically organized in Virtual Organizations (VOs) [38]. VOs is a key concept in Grid computing that pertains to virtualization of a system's user base around a set of resource-sharing rules and conditions. The formulation of VOs stems from the dynamical nature of resource sharing where resource availability, sharing conditions, and organizational memberships vary over time. This mechanism allows Grid resource usage allotments to be administrated and provided by decentralized organizations, to whom individual users and projects can apply for memberships and resource usage credits. VOs employ scalable resource allotment mechanisms suitable for cross-ownership domain aggregation of resources, and provide a way to provision resource usage without pre-existing trust relationships between resource owners and individual Grid users.

In summary, a Grid computing infrastructure should provide flexible and secure resource access and utilization through coordinated resource sharing models to dynamic collections of individuals and organizations. Furthermore, resources and users should be organized in Virtual Organizations and systems be devoid of centralized control, scheduling omniscience, and pre-existing trust relationships.

Chapter 4

Grid Job and Resource Management

A core task set of any Grid infrastructure is job and resource management, a term here used to collectively reference a set of processes and issues related to execution of programs on computational resources in Grid environments. This includes, e.g., management, monitoring, and brokering of computational resources; description, submission, and monitoring of jobs; fairshare scheduling [46] and accounting in Virtual Organizations; and various cross-site administrational and security issues.

Grid job and resource management tasks seem intuitive when viewed individually, but quickly become complex when considered as parts of larger systems. A number of component design trade-offs, requirements, and conditions are introduced by core Grid requirements for, e.g., system scalability and transparency, and tend to become oxymoronic when individual component designs are kept strictly task oriented. An approach taken in this work is to primarily regard components as parts of systems, and focus on component interoperability to promote system composition flexibility [26]. The primary focus of the Grid job and resource management contributions here is to abstract system complexity and heterogeneity, and to allow applications to leverage resource capabilities without becoming tightly coupled to particular Grids or Grid middlewares [27].

4.1 Grid Environments

Grid systems are composed through aggregation of multiple cooperating computing systems, and federated Grid environments are realized through (possibly hierarchical) federation of existing Grids.

In the naive model illustrated in Figure 1, regional organizations aggregate dedicated cluster-based resources from local supercomputing centers to form

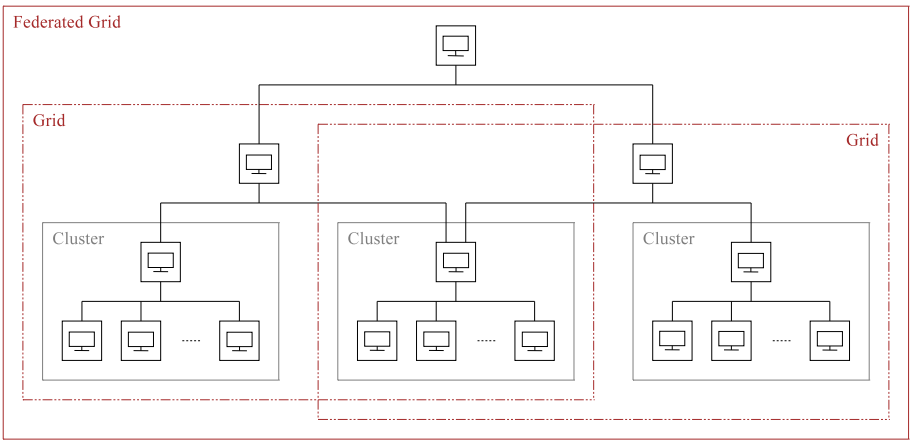


Figure 1: A naive Grid model. Grids aggregate clusters of computational resources, which may be part of multiple Grids. Federated Grid environments are composed from collaborative federation of existing Grids.

computational Grids. Due to the relatively homogeneous nature of today's supercomputers, such Grids typically exhibit low levels of system heterogeneity, and administrators can to a large extent influence system configuration and resource availability.

As also illustrated in Figure 1, international Grids are typically formed from collaborative federation of regional, and other existing Grids. As federated Grids typically aggregate resources from multiple Grids and resource sites, a natural consequence of resource and Grid federation is an increased degree of system heterogeneity. System heterogeneity may be expressed in many ways, e.g., through heterogeneity in hardware and software, resource availability, accessibility, and configuration, as well as in administration policies and utilization pricing. Technical heterogeneity issues are in Grid systems addressed through interface abstraction methods and generic resource description techniques, which allow virtualization of underlying resources and systems.

A core requirement in Grid systems is that resource owners at all levels retain full administrative control over their respective systems. This Grid characteristic, to be devoid of centralized control [33], is a design trait aimed to promote scalability in design and implementation of Grids, and imposes a number of cross-border administrational and security issues.

Security issues naturally arise in federation of computational resources over publicly accessible networks, i.e. the Internet, and are in Grid infrastructures addressed through use of strong cryptographic techniques such as Public Key Infrastructures (PKI) [49]. Grid users are typically organized in VOs, which stipulate rules and conditions for access to Grid resources, and authenticated through established security mechanisms such as PKI certificates [1].

4.2 Grid Resources and Middlewares

A typical HPC Grid resource consists of a high-end computer system equipped with (possibly customized) software such as

- data access and transfer utilities, e.g., GridFTP [18].
- batch systems and scheduling mechanisms, e.g., PBS [10] and Maui [48].
- job and resource monitoring tools, e.g., GridLab Mercury Monitor [8].
- computation frameworks, e.g., BLAST [60].

HTC resources are of more varied nature, CPU-cycle scavenging schemes such as Condor [61] for example typically utilize standard desktop machines, while volunteer computing efforts such as distributed.net [17] may see use of any type of computational resource provided by end-users. HTC Grids often deploy softwares that can be considered part of Grid middlewares on computational resources, e.g., Condor and BOINC [2] clients.

Grids are created through aggregation of computational resources, typically using Grid middlewares to abstract complexity and details of native resource systems such as schedulers and batch systems. Grid middlewares are (typically distributed) systems that act on top of local resource systems, abstracting native system interfaces, and provide interoperability between computational systems. To applications, Grid middlewares offer virtualized access to resource capabilities through abstractive job submission and control interfaces, information systems, and authentication mechanisms.

A number of different Grid middlewares exist, e.g., ARC [19], Globus [42], UNICORE [59], LCG/gLite [13], and vary greatly in design and implementation. In a simplified model, Grid middlewares contain functionality for

- resource discovery, often through specialized information systems.
- job submission, monitoring, and management.
- authentication and authorization of users.

Additionally, middlewares and related systems can incorporate solutions for advanced functionality such as resource brokering [61], accounting [41], and Grid-wide load balancing [13].

While one of the original motivations for construction of Grids where to address resource heterogeneity issues, complexity and size of Grid middlewares have led to a range of middleware interoperability issues, and given rise to the Grid interoperability contradiction [30]; Grid middlewares are not interoperable, and Grid applications are not portable between Grids. The Grid interoperability contradiction results in Grid applications being tightly coupled to Grid middlewares, and a lack of generic tools for Grid job management.

4.3 Resource Management

Grid resources are typically owned, operated, and maintained by local resource owners. Local resource sharing policies override Grid resource policies; computational resources shared in Grid environments according to defined schedules are possibly not available to Grid users outside scheduled hours. Due to this, and hardware and software failures, administrational downtime, etc., Grid resources are generally considered volatile.

In Grid systems, resource volatility is typically abstracted using dynamic service description and discovery techniques, utilizing loosely coupled models [65] for client-resource interaction. Local resource owners publish information about systems and resources in information systems, and Grid clients, e.g., resource brokers and submission engines, discover resources on demand and utilize the best resources currently available during the job submission phase.

Reliable resource monitoring mechanisms are critical to operation in Grid environments. While resource characteristics, e.g., hardware specifications and software installations, can be considered static, factors such as resource availability, load, and queue status are inherently dynamic. To facilitate Grid utilization and resource brokering, resource monitoring systems are used to provide information systems resource availability and status data.

As resource monitoring systems and information systems in Grid environments typically exist in different administrational domains, resource status information need to be disseminated through well-defined, machine-interpretable interfaces. The Web Service Resource Framework (WSRF) [35] specification family addresses Web Service state management issues, and contain interface definitions and notification mechanisms suitable for this task. In Grid environments, information systems potentially contain large quantities of information and can be segmented and hierarchically aggregated to partition resource information into manageable proportions.

4.4 Resource Brokering

A fundamental task in Grid job management is resource brokering; matching of a job to computational resource(s) suitable for job execution. In this model, resource brokers operate on top of Grid middlewares, and rely on information systems and job control systems to enact job executions.

Typically in Grid resource brokering, jobs are represented by job descriptions, which contain machine-readable representations of job characteristics and job execution meta-data. A number of proposed job description formats exist, including middleware-specific solutions such as Globus RSL [34], ARC XRSL [19], and standardization efforts such as JSDL [6]. Job descriptions provide information such as

- program to execute.

- parameters and environmental settings.
- hardware requirements, e.g., CPU, storage, and memory requirements.
- software requirements, e.g., required libraries and licenses.
- file staging information, e.g., data location and access protocols.
- meta-information, e.g., duration estimates and brokering preferences.

Resource brokering is subject to heuristic constraints and optimality criteria such as minimization of cost, maximization of resource computational capacity, minimization of data transfer time, etc., and is typically complicated by factors such as missing or incomplete brokering information, propagation latencies in information systems, and existence of competing scheduling mechanisms [30].

A common federated Grid environment characteristic designed to promote scalability is absence of scheduling omniscience. From this, two fundamental observations can be made. First, no scheduling mechanism can expect to monopolize job scheduling, all schedulers are forced to collaborate and compete with other mechanisms. Second, due to factors such as system latencies, information caching and status polling intervals, all Grid schedulers operate on information which to some extent is obsolete [31]. In these settings, Grid brokers and schedulers need to adapt to their environments and design emphasis should be placed on coexistence [27]. In particular, care should be taken to not reduce total Grid system performance, or performance of competing systems, through inefficient mechanisms in brokering and scheduling processes.

4.5 Job Control

Once resource brokering has been performed, and rendered a suitable computational resource candidate set, jobs can be submitted to resources for execution. For reasons of virtualization and separation of concerns, this is typically done through Grid middleware interfaces rather than directly to native resource interfaces, as resource heterogeneity issues would needlessly complicate clients and applications. Normally, execution of a Grid job on a computational resource adheres to the following schematic.

1. submission: job execution time is allocated at the resource site, i.e. the job is submitted to a resource job execution queue.
2. stage in: job data, including data files, scripts, libraries, and executables required for job execution are transfered to the computational resource as specified by the job description.
3. execution: the job is executed and monitored at the resource.
4. stage out: job data and result files are transfered from the computational resource as specified by the job description.

5. clean up: job data, temporary, and execution files are removed from the computational resource.

Naturally, ability to prematurely abort and externally monitor job executions must be provided by job control systems. In general, most systems of this complexity are built in layers, and Grid middlewares typically provide job control interfaces that abstract native resource system complexity.

As in any distributed system, a number of failures ranging from submission and execution failures to security credential validation and file transfer errors may occur during the job execution process. To facilitate client failure management and error recovery, failure context information must be provided clients. In Grid systems, failure management is complicated by factors such as resource ownership boundaries and resource volatility issues. Care must also be taken to isolate jobs executions, and to ensure that distribution of failure contexts not result in information leakage. Typically, Grids make use of advanced security features that make failure management, administration, and direct access to resource systems complex.

4.6 Job Management

Beyond generic resource brokering and job control capabilities, there exists a functionality set required by advanced high-level Grid applications. For example, efficient mechanisms for monitoring and workflow-based scheduling of jobs can greatly facilitate management of large sets of jobs.

Two types of Grid job monitoring mechanisms exist, pull-based and push-based. In pull models, clients and brokers poll resource status to detect and respond to changes in job and resource status. As jobs and Grid clients typically outnumber available Grid resources, polling-based resource update models scale poorly. As clients and resources exist in different ownership domains, pull models are also sometimes considered intrusive.

In push models, Grid resources, or systems monitoring them, publish status updates for jobs and resources in information systems or directly to interested clients. Push updates typically employ publish-subscribe communication patterns, where interested parties register for updates in advance, e.g., during job submission. In Grid systems, push models provide several performance benefits compared to pull models. Push models improve system scalability through reduced system load and decreased communication volumes, and may sometimes simplify client-side system design as they afford clients to act reactively rather than proactively. This reduced client complexity comes at the cost of increased service-side complexity. As Grid resources are volatile, systems distributed, and most Grids employ unreliable communication channels, push models must often be supplemented with pull model mechanisms. Push notifications can also be extended to notification brokering scenarios, and be incorporated in Message-Oriented Middleware (MOM) [9] or Enterprise Service Bus (ESB) [14]-like notification brokering schemes. The WS-Notification

[43] details interfaces for push model status notifications suitable for Grid job management architectures.

A common advanced Grid application requirement is to, possibly conditionally, run batches of jobs sequentially or in parallel. One way to organize these sets is in Grid workflows [50], where job interdependencies and coordination information are expressed along with job descriptions. In simple versions, workflows can be seen as job descriptions for sets of jobs. In more advanced versions, e.g., the Business Process Execution Language (BPEL) [4], workflows may themselves contain script-like instruction sets for, e.g., conditional execution, looping, and branching of jobs. When using workflows, Grid applications rely on workflow engines, e.g., Taverna [52], Pegasus [16], and Grid infrastructures to automate execution of job sets. An important question here becomes abstraction of level of detail, and balancing of level of detail against level of control for advanced job management systems [23].

Advanced job management systems may also provision functionality for customization of job execution, control, and management. In this case, job management components should provide interfaces for customization that does not require end-users or administrators to replace entire system components, but rather offer flexible configuration and code injection mechanisms [26, 27].

4.7 (Non-Intrusive) Interoperability

A large portion of Grid infrastructure operation builds on automation of Grid functionality tasks. This is achieved through Grid component and system collaboration, and thus require systems participating in Grids to provide machine-interpretable and interoperable system interfaces. Due to Grid heterogeneity issues stemming from Grid and resource federation, properties such as platform, language, and versioning independence become highly desirable. For these reasons, Grid components typically build on open standards and formats, and utilize technologies that facilitate system interoperation, e.g., XML and Web Services [40]. To promote non-intrusive interoperability in Grid system design, many Grid systems are realized as Service-Oriented Architectures [51].

Grid standardization efforts have proposed interfaces for many interoperability systems ranging from job description formats, e.g., JSDL [6], job submission and control interfaces, e.g., OGSA BES [36], to resource discovery, e.g., OGSA RSS [39], and Cloud computing interfaces, but broad consensus on best practices for Grid construction has yet to be reached.

Chapter 5

An Ecosystem of Grid Components

Currently, a number of open research questions regarding Grid and Cloud computing software design are being addressed by the scientific community. A common problem in current efforts is that applications tend to be tightly coupled to specific middlewares or Grids, and lack ability to be generally applicable to computational problems [25]. This work addresses Grid software design methodologies for computational eScience applications that support the majority of current computational approaches, and places focus on infrastructure composition and scalability rather than specific problem sets [24].

The methodology of this work builds on the idea of an ecosystem of Grid infrastructure components [62], which encompasses a view of a software ecosystem where individual components compete and collaborate for survival on an evolutionary basis. Fundamental to this idea is the notion of software niches, areas of functionality defined and populated by software components that interact and provision use of Grid resources to applications and end-users. Here, standardization of interfaces and software components help define niche boundaries, and continuous development of Grid infrastructure components and integration with eScience applications help shape and redefine niches (as well as the ecosystem at large) through competition, innovation, diversity, and evolution.

In this approach, identification and exploration of component and system traits likely to promote software survival in the Grid ecosystem are central, and generally help in identification and formulation of research questions. Softwares designed using this methodology focus on establishment of core functionality, and adapt to, and integrate with, members of neighboring niches rather than attempt to replace them.

Currently, advanced eScience applications and computational infrastructures require software and systems to scale with problem complexity and simultaneously abstract heterogeneity issues introduced by this scalability. For

usability, software also require interoperability and robustness to enable automation of repetitive tasks in computational environments, and flexibility in configuration and deployment to be employed in environments with great variance in usage and deployment requirements. The approach taken in this work is to build on top of Grid middlewares and create layers of flexible software that interoperate non-intrusively with components from different niches in the Grid ecosystem, and allow applications to be decoupled from Grid middlewares.

Chapter 6

Thesis Contributions

Large portions of the work in this thesis focus on Grid job and resource management issues, and address how these can be approached using middleware-independent techniques. Two of the papers outline and discuss approaches to Grid software development, one from a software engineering perspective (II), and one from a system (re)factorization point of view (III). Two of the papers (I and V) investigate and outline a generic architecture for Grid job management capable of adoption in a majority of existing Grid computing environments. Paper IV studies integration issues related to use of the proposed job management architecture, and details an integration architecture building on it.

6.1 Paper I

Paper I [21] investigates software design issues for Grid job management tools. Building on experiences from previous work [28, 29, 31], an architectural model for construction of a middleware-independent Grid job management system is proposed, and the design is detailed from an architectural point of view. In this work, a layered architecture of composable services that each manage a separate part of the Grid job management process is outlined, and design and implementation implications of this architecture are discussed. The architecture separates applications from infrastructure through a customizable set of services, and provides middleware-independence through use of (possible third party) middleware adaption plug-ins.

A Globus Toolkit 4-based [34] prototype implementation of some of the services in the architecture is presented, and the services are integrated with the ARC [19] and Globus [42] middlewares. To demonstrate the feasibility of the approach, preliminary results from prototype testing are presented along with an evaluation of system performance and system use cases.

6.2 Paper II

Paper II [24] analyzes Grid software development practices from a software engineering perspective. An approach to software development for high-level Grid resource management tools is presented, and the approach is illustrated by a discussion of software engineering attributes such as design heuristics, design patterns, and quality attributes for Grid software development.

The notion of an ecosystem of Grid infrastructure components is extended upon, and Grid component coexistence, composability, adoptability, adaptability, and interoperability are discussed in this context. The approach is illustrated by five case studies from recent software development efforts within the GIRD project; the Job Submission Service (JSS) [31], the Grid Job Management Framework (GJMF) [27], the Grid Workflow Execution Engine (GWEE) [22], the SweGrid Accounting System (SGAS) [41], and the Grid-Wide Fairshare Scheduling System (FSGrid) [20].

6.3 Paper III

Paper III [26] investigates Service-Oriented Architecture-based techniques for construction of Grid software, and details a set of service composition techniques for use in Grid infrastructure environments. Transparent service decomposition and dynamic service recomposition techniques are discussed in a Grid software (re)factorization setting, and implications of their use are elaborated upon. A set of architectural design patterns and service development mechanisms for service refactorization, service invocation optimization, customization of service mechanics, dynamic service configuration, and service monitoring are presented in detail, and synergetic effects between the patterns are discussed. Examples of use of the patterns in actual software development efforts are used throughout the paper to illustrate the presented approach.

6.4 Paper IV

Paper IV [25] addresses Grid software integration issues and discusses problems inherent to Grid applications being tightly coupled to Gird middlewares. The paper proposes an architecture for system integration focused on seamless integration of applications and Grid middlewares through a mediating layer handling resource brokering and notification delivery. The proposed architecture is illustrated in a case study where the LUNARC application portal [47] is integrated with the Grid Job Management Framework [27] presented in papers I and V. The proposed integration architecture is evaluated in a performance evaluation and findings from the integration efforts are presented throughout the paper.

6.5 Paper V

Paper V [27] further elaborates on the work of Paper I, and proposes a composable Service-Oriented Architecture-based framework architecture for middleware-independent Grid job management. The proposed architecture is presented in the context of development and deployment in an ecosystem of Grid components, and software requirements and framework composition are discussed in detail. The model of Paper I is extended with additional services for job description translation, system monitoring and logging, as well as a broader integration support functionality range. Furthermore, a proof-of-concept implementation of the entire framework is presented and evaluated in a performance evaluation that illustrates some of the major trade-offs in framework use.

The Grid ecosystem model of Paper II is further developed and discussed in the context of the proposed job management architecture, and the software composition techniques of Paper III are built upon and evaluated in the context of this project. Throughout the paper, a number of software design and implementation findings are presented, and the framework is related to a set of similar software development efforts within adjoining Grid ecosystem niches.

Chapter 7

Future Work

A number of possible future extensions to the work of this thesis have been identified, some of which are currently pursued within the GIRD project. Further development, and documentation of experiences from use of the software development model of Paper II is a continuous effort, and of current special interest is adoption of the model to Cloud computing software development efforts. The model itself is currently utilized in a number of projects under the GIRD multi-project umbrella, and are in the projects of this thesis combined with the techniques of Paper III.

The service composition techniques of Paper III have been further developed in work on Paper V, and are currently under investigation for extension in a code-generation effort within multiple projects. The techniques lend themselves well to software refactorization efforts and prototype implementations are being developed for integration with the Apache Axis2 [7] SOAP [44] engine. Extension of these techniques to Representational State Transfer (REST)-based [32] Resource-Oriented Architectures (ROA) [53] would possibly be a viable alternative to current Web Service Description Language (WSDL)-based [15] code generation. In this case the abstraction of the mechanisms would naturally be placed in API implementations, instead of in generated stub code. Extension of these techniques to a more ubiquitous notification scheme, where the current WSRF-based [35] approach could be extended to a more generic MOM- [9] or ESB-based [14] approach would also be possible. Development of a more generic framework for service development adapted to a larger number of service engines would further such efforts.

The job management framework of Paper I and V is currently being developed into a more mature software product scheduled for use in SweGrid, the Swedish national Grid, and a port of the framework to alternative SOAP stacks is currently under investigation. Interesting research questions related to the architecture of this framework include, e.g., development of data management capabilities, (further) adaption to standardization efforts, investigation of advanced notification brokering capabilities, and inclusion of advanced resource

brokering features such as advance reservation and coallocation of resources, and classadd-based match-making.

Further development and integration of high-level job clients such as workflow engines and Grid portals would be beneficial, as well as further investigation of integration architectures such as that of Paper IV, as these are expected to increase the understanding of application-infrastructure integration issues. Investigation of (minimalistic) implementation approaches for Grid middleware development and simulation are also expected to render a deeper understanding of these issues. Integration with Cloud Computing solutions, and other virtualization-based infrastructure techniques, are also of interest and can be expected to increase the adoptability and flexibility of these techniques.

Bibliography

[1] C. Adams and S. Farrell. Internet X. 509 public key infrastructure certificate management protocols, 1999.

[2] D.P. Anderson. BOINC: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004.

[3] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[4] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. Business process execution language for web services, version 1.1. *Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems*, 2003.

[5] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performancecomputers. In *Proceedings of Supercomputing'90*, pages 2–11, 1990.

[6] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, A. S. McGough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) specification, version 1.0. http://www.ogf.org/documents/GFD.56.pdf, September 2009.

[7] Apache Web Services Project - Axis2. http://ws.apache.org/axis2, September 2009.

[8] Z. Balaton and G. Gombas. Resource and job monitoring in the grid. *Lecture notes in computer science*, pages 404–411, 2003.

[9] G. Banavar, T. D. Chandra, R. E. Strom, and D. C. Sturman. A case for message oriented middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 1–18, London, UK, 1999. Springer-Verlag.

[10] A. Bayucan, R.L. Henderson, C. Lesiak, B. Mann, T. Proett, and D. Tweten. Portable Batch System: External reference specification. Technical report, Technical report, MRJ Technology Solutions, 1999.

[11] H. Benoit-Cattin, G. Collewet, B. Belaroussi, H. Saint-Jalmes, and C. Odet. The SIMRI project: a versatile and interactive MRI simulator. *Journal of Magnetic Resonance*, 173(1):97–115, 2005.

[12] BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/. September 2009.

[13] J. Knobloch (Chair) and L. Robertson (Project Leader). LHC computing Grid technical design report. http://lcg.web.cern.ch/LCG/tdr/, September 2009.

[14] D. Chappell. *Enterprise Service Bus*. O'Reilly Media, Inc., 2004.

[15] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, September 2009.

[16] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, and D.S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[17] distributed.net. http://www.distributed.net/. September 2009.

[18] W. Allcock (editor). GridFTP: Protocol extensions to FTP for the Grid. http://www.ogf.org/documents/GFD.20.pdf, September 2009.

[19] M. Ellert, M. Grønager, A. Konstantinov, B. Kónya, J. Lindemann, I. Livenson, J. L. Nielsen, M. Niinimäki, O. Smirnova, and A. Wäänänen. Advanced resource connector middleware for lightweight computational Grids. *Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications*, 27(2):219–240, 2007.

[20] E. Elmroth and P. Gardfjäll. Design and evaluation of a decentralized system for Grid-wide fairshare scheduling. In H. Stockinger, R. Buyya, and R. Perrott, editors, *e-Science 2005, First International Conference on e-Science and Grid Computing*, pages 221–229. IEEE CS Press, 2005.

[21] E. Elmroth, P. Gardfjäll, A. Norberg, J. Tordsson, and P-O. Östberg. Designing General, Composable, and Middleware-Independent Grid Infrastructure Tools for Multi-Tiered Job Management. In T. Priol and M. Vaneschi, editors, *Towards Next Generation Grids*, pages 175–184. Springer-Verlag, 2007.

[22] E. Elmroth, F. Hernández, and J. Tordsson. A light-weight Grid workflow execution engine enabling client and middleware independence. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science, vol. 4967*, pages 754–761. Springer-Verlag, 2008.

[23] E. Elmroth, F. Hernández, and J. Tordsson. Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment. *Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications*, 2009, to appear.

[24] E. Elmroth, F. Hernández, J. Tordsson, and P-O. Östberg. Designing Service-Based Resource Management Tools for a Healthy Grid Ecosystem. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science, vol. 4967*, pages 259–270. Springer-Verlag, 2008.

[25] E. Elmroth, S. Holmgren, J. Lindemann, S. Toor, and P-O. Östberg. Empowering a Flexible Application Portal with a SOA-based Grid Job Management Framework. In *The 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, to appear, 2009.

[26] E. Elmroth and P-O. Östberg. Dynamic and Transparent Service Compositions Techniques for Service-Oriented Grid Architectures. In S. Gorlatch, P. Fragopoulou, and T. Priol, editors, *Integrated Research in Grid Computing*, pages 323–334. Crete University Press, 2008.

[27] E. Elmroth and P-O. Östberg. A Composable Service-Oriented Architecture for Middleware-Independent and Interoperable Grid Job Management. *UMINF 09.14*, Department of Computing Science, Umeå University, Sweden. Submitted for Journal Publication, 2009.

[28] E. Elmroth and J. Tordsson. An interoperable, standards-based Grid resource broker and job submission service. In H. Stockinger, R. Buyya, and R. Perrott, editors, *e-Science 2005, First International Conference on e-Science and Grid Computing*, pages 212–220. IEEE CS Press, 2005.

[29] E. Elmroth and J. Tordsson. A Grid resource broker supporting advance reservations and benchmark-based resource selection. In J. Dongarra, K. Madsen, and J. Waśniewski, editors, *Applied Parallel Computing - State of the Art in Scientific Computing, Lecture Notes in Computer Science vol. 3732*, pages 1061–1070. Springer-Verlag, 2006.

[30] E. Elmroth and J. Tordsson. Grid resource brokering algorithms enabling advance reservations and resource selection based on performance predictions. *Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications*, 24(6):585–593, 2008.

[31] E. Elmroth and J. Tordsson. A standards-based grid resource brokering service supporting advance reservations, coallocation and cross-grid interoperability. *Concurrency Computat.: Pract. Exper.*, 2009. accepted.

[32] R. T. Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[33] I. Foster. What is the grid? a three point checklist. *GRID today*, 1(6):22–25, 2002.

[34] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In H. Jin, D. Reed, and W. Jiang, editors, *IFIP International Conference on Network and Parallel Computing, LNCS 3779*, pages 2–13. Springer-Verlag, 2005.

[35] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, and S. Weerawarana. Modeling stateful resources with Web services. http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf, September 2009.

[36] I. Foster, A. Grimshaw, P. Lane, W. Lee, M. Morgan, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, and M. Theimer. OGSA© basic execution service version 1.0. http://www.ogf.org/documents/GFD.108.pdf, September 2009.

[37] I. Foster and C. Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2004.

[38] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200, 2001.

[39] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, version 1.5, 2006. http://www.ogf.org/documents/GFD.80.pdf, May 2009.

[40] I. Foster and S. Tuecke. Describing the elephant: The different faces of IT as service. *ACM Queue*, 3(6):26–34, 2005.

[41] P. Gardfjäll, E. Elmroth, L. Johnsson, O. Mulmo, and T. Sandholm. Scalable Grid-wide capacity allocation with the SweGrid Accounting System (SGAS). *Concurrency Computat.: Pract. Exper.*, 20(18):2089–2122, 2008.

[42] Globus. http://www.globus.org. September 2009.

[43] S. Graham, D. Hull, and B. Murray. Web Services Base Notification 1.3 (WS-BaseNotification). http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf, September 2009.

[44] M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, H. Frystyk Nielsen, A. Karmarkar, and Y. Lafon. SOAP version 1.2 part 1: Messaging framework. http://www.w3.org/TR/soap12-part1/, September 2009.

[45] T. Hansen, S. Tilak, S. Foley, K. Lindquist, F. Vernon, A. Rajasekar, and J. Orcutt. ROADNet: A network of SensorNets. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 579–587, 2006.

[46] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[47] J. Lindemann and G. Sandberg. An extendable GRID application portal. In *European Grid Conference (EGC)*. Springer Verlag, 2005.

[48] Maui Cluster Scheduler. http://www.clusterresources.com/products/maui/, September 2009.

[49] U. Maurer. Modelling a public-key infrastructure. *Lecture Notes in Computer Science*, 1146:325–350, 1996.

[50] F. Neubauer, A. Hoheisel, and J. Geiler. Workflow-based Grid applications. *Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications*, 22(1-2):6–15, 2006.

[51] OASIS Open. Reference Model for Service Oriented Architecture 1.0. http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf, September 2009.

[52] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[53] H. Overdick. The resource-oriented architecture. In *2007 IEEE Congress on Services (Services 2007)*, pages 340–347, 2007.

[54] I. Raicu, I.T. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS) 2008.*, pages 1–11, 2008.

[55] M. Russell, G. Allen, G. Daues, I. Foster, E. Seidel, J. Novotny, J. Shalf, and G. von Laszewski. The astrophysics simulation collaboratory: A science portal enabling community software development. *Cluster Computing*, 5(3):297–304, 2002.

[56] B. Segal, L. Robertson, F. Gagliardi, and F. Carminati. Grid computing: The European Data Grid Project. In *Nuclear Science Symposium Conference Record, 2000 IEEE*, volume 1, page 2/1, 2000.

[57] M. Snir, S.W. Otto, D.W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The complete reference*. MIT Press Cambridge, MA, USA, 1995.

[58] H. Stockinger. Defining the grid: a snapshot on the current view. *The Journal of Supercomputing*, 42(1):3–17, 2007.

[59] A. Streit, D. Erwin, Th. Lippert, D. Mallmann, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and Ph. Wieder. UNICORE - from project results to production grids. In L. Grandinetti, editor, *Grid Computing: The New Frontiers of High Performance Processing, Advances in Parallel Computing 14*, pages 357–376. Elsevier, 2005.

[60] T.A. Tatusova and T.L. Madden. BLAST 2 Sequences, a new tool for comparing protein and nucleotide sequences. *FEMS microbiology letters*, 174(2):247–250, 1999.

[61] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency Computat. Pract. Exper.*, 17(2–4):323–356, 2005.

[62] The Globus Project. An "ecosystem" of Grid components. http://www.globus.org/grid_software/ecology.php, September 2009.

[63] The Grid Infrastructure Research & Development (GIRD) project. Umeå University, Sweden. http://www.gird.se, September 2009.

[64] J. Treadwell. Open grid services architecture glossary of terms. In *Global Grid Forum, Lemont, Illinois, USA, GFD-I*, volume 44, pages 2–2, 2005.

[65] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. *Lecture Notes in Computer Science*, pages 49–64, 1997.

[66] R. Williams. Grids and the virtual observatory. *Grid Computing: Making the Global Infrastructure a Reality*, pages 837–858, 2003.

I

Paper I

Designing General, Composable, and Middleware-Independent Grid Infrastructure Tools for Multi-Tiered Job Management*

Erik Elmroth, Peter Gardfjäll, Arvid Norberg,
Johan Tordsson, and Per-Olov Östberg

Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden
{elmroth, peterg, arvid, tordsson, p-o}@cs.umu.se
http://www.gird.se

Abstract: We propose a multi-tiered architecture for middleware-independent Grid job management. The architecture consists of a number of services for well-defined tasks in the job management process, offering complete user-level isolation of service capabilities, multiple layers of abstraction, control, and fault tolerance. The middleware abstraction layer comprises components for targeted job submission, job control and resource discovery. The brokered job submission layer offers a Grid view on resources, including functionality for resource brokering and submission of jobs to selected resources. The reliable job submission layer includes components for fault tolerant execution of individual jobs and groups of independent jobs, respectively. The architecture is proposed as a composable set of tools rather than a monolithic solution, allowing users to select the individual components of interest. The prototype presented is implemented using the Globus Toolkit 4, integrated with the Globus Toolkit 4 and NorduGrid/ARC middlewares and based on existing and emerging Grid standards. A performance evaluation reveals that the overhead for resource discovery, brokering, middleware-specific format conversions, job monitoring, fault tolerance, and management of individual and groups of jobs is sufficiently small to motivate the use of the framework.

Key words: Grid job management infrastructure, standards-based architecture, fault tolerance, middleware-independence, Grid ecosystem.

* By permission of Springer Verlag

DESIGNING GENERAL, COMPOSABLE, AND MIDDLEWARE-INDEPENDENT GRID INFRASTRUCTURE TOOLS FOR MULTI-TIERED JOB MANAGEMENT*

Erik Elmroth, Peter Gardfjäll, Arvid Norberg,
Johan Tordsson, and Per-Olov Östberg
Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden
{elmroth, peterg, arvid, tordsson, p-o}@cs.umu.se
http://www.gird.se

Abstract We propose a multi-tiered architecture for middleware-independent Grid job management. The architecture consists of a number of services for well-defined tasks in the job management process, offering complete user-level isolation of service capabilities, multiple layers of abstraction, control, and fault tolerance. The middleware abstraction layer comprises components for targeted job submission, job control and resource discovery. The brokered job submission layer offers a Grid view on resources, including functionality for resource brokering and submission of jobs to selected resources. The reliable job submission layer includes components for fault tolerant execution of individual jobs and groups of independent jobs, respectively. The architecture is proposed as a composable set of tools rather than a monolithic solution, allowing users to select the individual components of interest. The prototype presented is implemented using the Globus Toolkit 4, integrated with the Globus Toolkit 4 and NorduGrid/ARC middlewares and based on existing and emerging Grid standards. A performance evaluation reveals that the overhead for resource discovery, brokering, middleware-specific format conversions, job monitoring, fault tolerance, and management of individual and groups of jobs is sufficiently small to motivate the use of the framework.

Keywords: Grid job management infrastructure, standards-based architecture, fault tolerance, middleware-independence, Grid ecosystem.

*Financial support has been received from The Swedish Research Council (VR) under contract number 621-2005-3667. This research was conducted using the resources of the High Performance Computing Center North (HPC2N).

1. Introduction

We investigate designs for a standards-based, multi-tier job management framework that facilitates application development in heterogeneous Grid environments. The work is driven by the need for job management tools that:

- offer multiple levels of functionality abstraction,
- offer multiple levels of job control and fault tolerance,
- are independent of, and easily integrated with, Grid middlewares,
- can be used on a component-wise basis and at the same time offer a complete framework for more advanced functionality,

An overall objective of this work is to provide understanding of how to best develop such tools. Among architectural aspects of interest are, e.g., to what extent job management functionalities should be separated into individual components or combined into larger, more feature-rich components, taking into account both functionality and performance. As an integral part of the project, we also evaluate and contribute to current Grid standardization efforts for, e.g., data formats, interfaces and architectures. The evaluation of our approach will in the long term lead to the establishment of a set of general design recommendations.

Features of our prototype software include user-level isolation of service capabilities, a wide range of job management functionalities, such as basic submission, monitoring, and control of individual jobs; resource brokering; autonomous processing; and atomic management of sets of jobs. All services are designed to be middleware-independent with middleware integration performed by plug-ins in lower-level components. This enables both easy integration with different middlewares and transparent cross-middleware job submission and control.

The design and implementation of the framework rely on emerging Grid and Web service standards [3],[9],[2] and build on our own experiences from developing resource brokers and job submission services [6],[7],[8], Grid scheduling support systems [5], and the SweGrid Accounting System (SGAS) [10]. The framework is based on WSRF and implemented using the Globus Toolkit 4.

2. A Model for Multi-Tiered Job Submission Architectures

In order to provide a highly flexible and customizable architecture, a basic design principle is to develop several small components, each designed to perform a single, well-defined task. Moreover, dependencies between components are kept to a minimum, and are well-defined in order to facilitate the use of alternative components. These principles are adopted with the overall idea that a

specific middleware, or a specific user, should be able to make use of a subset of the components without having to adopt an entire, monolithic system [11].

We propose to organize the various components according to the following layered architecture.

Middleware Abstraction Layer. Similar to the hardware abstraction layer of an operating system, the middleware abstraction layer provides the functionality of a set of middlewares while encapsulating the details of these. This construct allows other layers to access resources running different middlewares without any knowledge of their actual implementation details.

Brokered Job Submission Layer. The brokered job submission layer offers fundamental capabilities such as resource discovery, resource selection and job submission, but without any fault tolerance mechanisms.

Reliable Job Submission Layer. The reliable job submission layer provides a fault tolerant, reliable job submission. In this layer, individual jobs or groups of jobs are automatically processed according to a customizable protocol, which by default includes repeated submission and other failure handling mechanisms.

Advanced Job Submission & Application Layers. Above the three previously mentioned layers, we foresee both an *advanced job submission layer*, comprising, e.g., workflow engines, and an *application layer*, comprising , e.g., Grid applications, portals, problem solving environments and workflow clients.

3. The Grid Job Management Framework (GJMF)

Here follows a brief introduction to the GJMF, where the individual services and their respective roles in the framework are described.

The GJMF offers a set of services which combined constitute a multi-tiered job submission, control and management architecture. A mapping of the GJMF architecture to the proposed layered architecture is provided in Figure 1.

All services in the GJMF offer a user-level isolation of the service capabilities; a separate service component is instantiated for each user and only the owner of a service component is allowed to access the service capabilities. This means that the whole architecture supports a decentralized job management policy, and strives to optimize the performance for the individual user.

The services in the GJMF also utilize a local call structure, using local Java calls whenever possible for service-to-service interaction. This optimization is only possible when the interacting services are hosted in the same container.

The GJMF supports a dynamic one-to-many relationship model, where a higher-level service can switch between lower-level service instances to improve fault tolerance and performance.

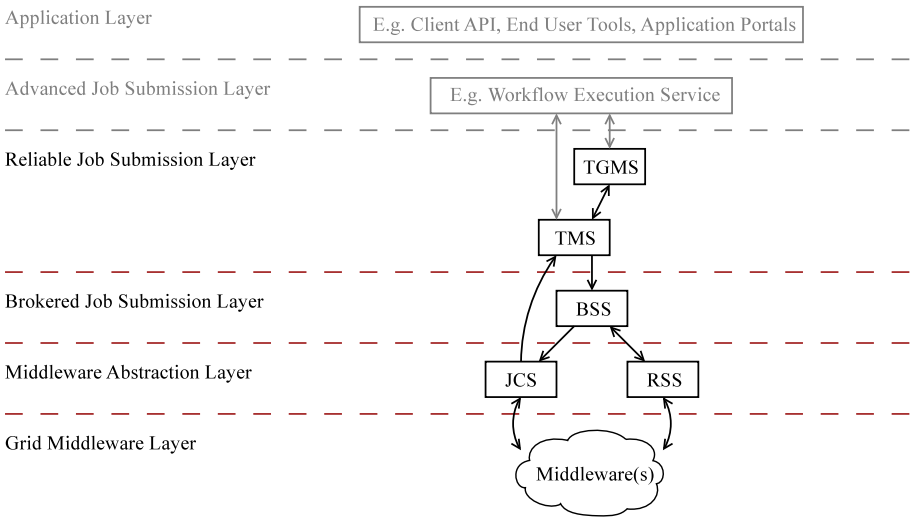


Figure 1. GJMF components mapped to their respective architectural layers.

As a note on terminology, there are two different types of job specifications used in the GJMF: abstract *task* specifications and concrete *job* specifications. Both are specified in JSDL [3], but vary in content. A job specification includes a reference to a computational resource to process the job, and therefore contains all information required to submit the job. A task specification contains all information required except a computational resource reference. The act of brokering, the matching of a job specification to a computational resource, thus transforms a task to a job.

Job Control Service (JCS). The JCS provides a functionality abstraction of the underlying middleware(s) and offers a platform- and middleware-independent job submission and control interface. The JCS operates on jobs and can submit, query, stop and remove jobs. The JCS also contains customization points for adding support for new middlewares and exposes information about jobs it controls through WSRF resource properties, which either can be explicitly queried or monitored for asynchronous notifications. Note that this functionality is offered regardless of underlying middleware, i.e., if a middleware does not support event callbacks the JCS explicitly retrieves the information required to provide the notifications. Currently, the JCS supports the GT4 and the ARC middlewares.

Resource Selection Service (RSS). The RSS is a resource selection service based on the OGSA Execution Management Services (OGSA EMS) [9]. The OGSA EMS specify a resource selection architecture consisting of two services, the Candidate Set Generator (CSG) and the Execution Planning Service (EPS).

The purpose of the CSG is to generate a candidate set, containing machines where the job *can* execute, whereas the EPS determines where the job *should* execute. Upon invocation, the EPS contacts the CSG for a list of candidate machines, reorders the list according to a previously known or explicitly provided set of rules and returns an *execution plan* to the caller.

The current OGSA EMS specification is incomplete, e.g., the interface of the CSG is yet to be determined. Due to this, the CSG and the EPS are in our implementation combined into one service - the RSS. The candidate set generation is implemented by dynamical discovery of available resources using a Grid information service, e.g., GT4 WS-MDS, and filtering of the identified resources against the requirements in the job description. The RSS contains a caching mechanism for Grid information, which alleviates the frequency of information service queries.

Brokering & Submission Service (BSS). The BSS provides a functionality abstraction for brokered task submission. It receives a task (i.e., an abstract job specification) as input and retrieves an execution plan (a prioritized list of jobs) from the RSS. Next, the BSS uses a JCS to submit the job to the most suitable resource found in the execution plan. This process is repeated for each resource in the execution plan until a job submission has succeeded or the resource list has been exhausted. A client submitting a task to the BSS receives an EPR to a job WS-Resource in the JCS as a result. All further interaction with the job, e.g., status queries and job control is thus performed directly against the JCS.

Task Management Service (TMS). The TMS provides a high-level service for automated processing of individual tasks, i.e., a user submits a task to the TMS which repeatedly sends the task to a known BSS until a resulting job is successfully executed or a maximum number of attempts have been made. Internally, the TMS contains a per-user job pool from which jobs are selected for sequential submission. The TMS job pool is of a configurable, limited size and acts as a task submission throttle. It is designed to limit both the memory requirements for the TMS and the flow of job submissions to the JCS. The job submission flow is also regulated via a congestion detection mechanism, where the TMS implements an incremental back-off behavior to limit BSS load in situations where the RSS is unable to locate any appropriate computational resources for the task. The TMS tracks job progress via the JCS and manages a state machine for each job, allowing it to handle failed jobs in an efficient manner. The TMS also contains customization points where the default behaviors for task selection, failure handling and state monitoring can be altered via Java plug-ins.

Task Group Management Service (TGMS). Like the TMS for individual tasks, the TGMS provides an automated, reliable submission solution for groups of tasks. The TGMS relies on the TMS for individual task submission and offers a convenient way to submit groups of independent tasks. Internally, the TGMS contains two levels of queues for each user. All task groups that contain unprocessed tasks are placed in a task group queue. Each task group queue, in turn, contains its own task queue. Tasks are selected for submission in two steps: first an active task group is selected, then a task from this task group is selected for submission. By default, tasks are resubmitted until they have reached a terminal state (i.e., succeeded or failed). A task group reaches a terminal state once all its tasks are processed. A task group can also be suspended, either explicitly by the user or implicitly by the service when it is no longer meaningful to continue to process the task group, e.g., when associated user credentials have expired. A suspended task group must be explicitly resumed to become active. The TGMS contains customization points for changing the default behaviors for task selection, failure handling and state monitoring.

Client API. The Client API is an integral part of the GJMF; it provides utility libraries and interfaces for creating tasks and task groups, translating job descriptions, customizing service behaviors, delegating credentials and contains service-level APIs for accessing all components in the GJMF. The purpose of the GJMF Client API is to provide easy-to-use programmable (Java) access to all parts of the GJMF.

For further information regarding the GJMF, including design documents and technical documentation of the services, see [12].

4. Performance Evaluation

We evaluate the performance of the TGMS and the TMS by investigating the total cost imposed by the GJMF services compared to the total cost of using the native job submission mechanism of a Grid middleware, GT4 WS-GRAM (without performing resource discovery, brokering, fault recovery etc.).

In the reference tests with WS-GRAM, a client sequentially submits a set of jobs using the WS-GRAM Java API, delaying the submission of a job until the previous one has been successfully submitted. All jobs run the trivial /bin/true command and are executed on the Grid resources using the POSIX Fork mechanism. The jobs in a test are distributed evenly among the Grid resources using a round-robin mechanism. The WS-GRAM tests do not include any WS-MDS interaction. No job input or output files are transferred and no credentials are delegated to the submitted jobs. In each test, the total wall clock time is recorded. Tests are performed with selected numbers of jobs, ranging from 1 to 750.

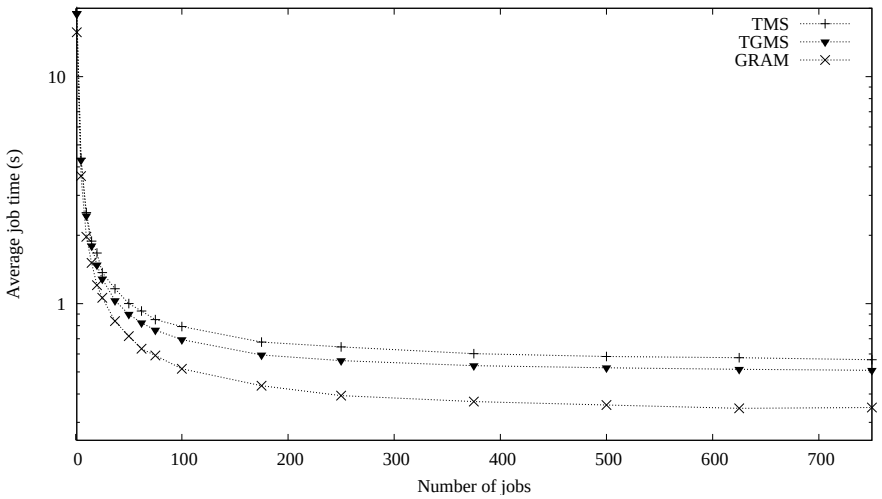


Figure 2. GRAM and GJMF job processing performance.

The configuration of the GJMF tests is the same as for the WS-GRAM tests, with the following additions. For the TGMS tests, user credentials are delegated from the client to the service for each task group (each test). Delegation is also performed only once per test in the TMS case, as all jobs in a TMS test reuse the same delegated credentials. For both the TGMS and the TMS tests, the BSS performs resource discovery using the GT4 WS-MDS Grid information system and caches retrieved information for 60 seconds. In the TMS and TGMS tests, all services are co-located in the same container, to enable the use of local Java calls between the services, instead of (more costly) Web service invocations.

Test Environment. The test environment includes four identical 2 GHz AMD Opteron CPU, 2 GB RAM machines, interconnected with a 100 Mbps Ethernet network, and running Ubuntu Linux 2.6 and Globus Toolkit 4.0.3.

In all tests, one machine runs the GJMF (or the WS-GRAM client) and the other three act as WS-GRAM/GT4 resources. For the GJMF tests, the RSS retrieves WS-MDS information from one of the three resources, which aggregates information about the other two.

Analysis. Figure 2 illustrates the average time required to submit and execute a job for different number of jobs in the test. As seen in the figure, the TGMS offers a more efficient way to submit multiple tasks than the TMS. This is due to the fact that the TMS client performs one Web service invocation per task whereas the TGMS client only makes a single, albeit large, call to the TGMS. The TGMS client requires between 13 (1 task) and 16.6 seconds (750 tasks) to delegate credentials, invoke the Web service and get a reply. For the TMS,

the initial Web service call takes roughly 13 seconds (as it is associated with dynamic class-loading, initialization and delegation of credentials), additional calls average between 0.4 and 0.6 seconds. For the GRAM client, the initial Web service invocation takes roughly 12 seconds. The additional TMS Web service calls quickly become the dominating factor as the number of jobs are increased. When using Web service calls between the TGMS and the TMS this factor is canceled out. Conversely, when co-located with the TMS and using local Java calls, the TGMS only suffers a negligible overhead penalty for using the TMS for task submission. In a test with 750 jobs, the average job time is roughly 0.35 seconds for WS-GRAM, and approximately 0.51 and 0.57 seconds for the TGMS and TMS, respectively.

As the WS-GRAM client and the JCS use the same GT4 client libraries, the difference between the WS-GRAM performance and that of the other services can be used as a direct measure of the GJMF overhead.

In the test cases considered, the time required to submit a job (or a task) can be divided into three parts.

1 The initialization time for GT4 Java clients. This includes time for class loading and run-time environment initialization. This time may vary with the system setup but is considered to be constant for all three test cases.
2 The time required to delegate credentials. This only applies to the GJMF tests, not the test of WS-GRAM. Even though delegated credentials are shared between jobs, the TMS is still slightly slower than the TGMS in terms of credential delegation. The TMS has to retrieve the delegated credential for each task, whereas the TGMS only retrieves the delegated credential once per test.
3 The Web service invocation time. This factor grows with the size of the messages exchanged and affects the TGMS, as a description of each individual task is included in the TGMS input message. The invocation time is constant for the TMS and WS-GRAM tests, as these services exchange fixed size messages.

Summary. When co-hosted in the same container, the GJMF services allots an overhead of roughly 0.2 seconds per task for large task groups (containing 750 tasks or more). The main part of this overhead is associated with Java class loading, delegation of credentials and initial Web service invocation. These factors result in larger average overheads for smaller task groups. For task groups containing 5 tasks, the average overhead per task is less than 1 second, and less than 0.5 seconds for 15 tasks. It should also be noted that, as jobs are submitted sequentially but executed in parallel, the submission time (including the GJMF overhead), is masked by the job execution time. Therefore, when using real world applications with longer job durations than those in the tests, the impact of the GJMF overhead is reduced.

5. Related Work

We have identified a number of contributions that relate to this project in different ways. For example, the Gridbus [16] middleware includes a layered architecture for platform-independent Grid job management; the GridWay Metascheduler [13] offers reliable and autonomous execution of jobs; the GridLab Grid Application Toolkit [1] provides a set of services to simplify Grid application development; GridSAM [15] offers a Web service-based job submission pipeline which provides middleware abstraction and uses JSDL job descriptions; P-GRADE [14] provides reliable, fault-tolerant parallel program execution on the grid; and GEMLCA [4] offers a layered architecture for running legacy applications through grid services. These contributions all include features which partially overlap the functionality available in the GJMF. However, our work distinguishes itself from these contributions by, in the same software, providing i) a composable service-based solution, ii) multiple levels of abstraction, iii) middleware-interoperability while building on emerging Grid service standards.

6. Concluding Remarks

We propose a multi-tiered architecture for building general Grid infrastructure components and demonstrate the feasibility of the concept by implementing a prototype job management framework. The GJMF provides a standards-based, fault-tolerant job management environment where users may use parts of, or the entire framework, depending on their individual requirements. Furthermore, we demonstrate that the overhead incurred by using the framework is sufficiently small (approaching 0.2 seconds per job for larger groups of jobs) to motivate the practical use of such an architecture. Initial tests demonstrate that by proper methods, including reuse of delegated credentials, caching of Grid information and local Java invocations of co-located services, it is possible to implement an efficient service-based multi-tier framework for job management. Considering the extra functionality offered and the small additional overhead imposed, the GJMF framework is an attractive alternative to a pure WS-GRAM client for the submission and management of large numbers of jobs.

Acknowledgments

We are grateful to the anonymous referees for constructive comments that have contributed to the clarity of this paper.

References

[1] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor. Enabling

applications on the Grid - a GridLab overview. *Int. J. High Perf. Comput. Appl.*, 17(4), 2003.

[2] S. Andreozzi, S. Burke, L. Field, S. Fisher, B. Kónya, M. Mambelli, J. M. Schopf, M. Viljoen, and A. Wilson. Glue schema specification version 1.2 draft 7. http://glueschema.forge.cnaf.infn.it/uploads/Spec/GLUEInfoModel_1_2_final.pdf, March 2007.

[3] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, A. S. McGough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) specification, version 1.0. http://www.ogf.org/documents/GFD.56.pdf, March 2007.

[4] T. Delaittre, T. Kiss, A. Goyeneche, G. Terstyanszky, S.Winter, and P. Kacsuk. GEMLCA: Running legacy code applications as Grid services. *Journal of Grid Computing*, 3(1 – 2):75 – 90, June 2005. ISSN: 1570-7873.

[5] E. Elmroth and P. Gardfjäll. Design and evaluation of a decentralized system for Grid-wide fairshare scheduling. In H. Stockinger, R. Buyya, and R. Perrott, editors, *e-Science 2005, First International Conference on e-Science and Grid Computing*, pages 221–229. IEEE CS Press, 2005.

[6] E. Elmroth and J. Tordsson. An interoperable, standards-based Grid resource broker and job submission service. In H. Stockinger, R. Buyya, and R. Perrott, editors, *e-Science 2005, First International Conference on e-Science and Grid Computing*, pages 212–220. IEEE CS Press, 2005.

[7] E. Elmroth and J. Tordsson. A standards-based Grid resource brokering service supporting advance reservations, coallocation and cross-Grid interoperability. Submitted to *Concurrency and Computation: Practice and Experience*, December 2006.

[8] E. Elmroth and J. Tordsson. Grid resource brokering algorithms enabling advance reservations and resource selection based on performance predictions. *Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications*, 2007, to appear.

[9] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, version 1.5. http://www.ogf.org/documents/GFD.80.pdf, March 2007.

[10] P. Gardfjäll, E. Elmroth, L. Johnsson, O. Mulmo, and T. Sandholm. Scalable Grid-wide capacity allocation with the SweGrid Accounting System (SGAS). Submitted to *Concurrency and Computation: Practice and Experience*, October 2006.

[11] Globus. An "Ecosystem" of Grid Components. http://www.globus.org/grid_software/ecology.php. March 2007.

[12] Grid Infrastructure Research & Development (GIRD). http://www.gird.se. March 2007.

[13] E. Huedo, R.S. Montero, and I.M. Llorente. A framework for adaptive execution on Grids. *Software - Practice and Experience*, 34(7):631–651, 2004.

[14] P. Kacsuk, G. Dózsa, J. Kovács, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombás. P-GRADE: a Grid programming environment. *Journal of Grid Computing*, 1(2):171 – 197, 2003.

[15] W. Lee, A. S. McGough, and J. Darlington. Performance evaluation of the GridSAM job submission and monitoring system. In *UK e-Science All Hands Meeting*, Nottingham, UK, 2005.

[16] S. Venugopal, R. Buyya, and L. Winton. A Grid service broker for scheduling e-Science applications on global data Grids. *Concurrency Computat. Pract. Exper.*, 18(6):685–699, May 2006.

II

Paper II

Designing Service-Based Resource Management Tools for a Healthy Grid Ecosystem*

Erik Elmroth, Francisco Hernández, Johan Tordsson, and Per-Olov Östberg

Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden
{elmroth, hernandf, tordsson, p-o}@cs.umu.se
http://www.gird.se

Abstract: We present an approach for development of Grid resource management tools, where we put into practice internationally established high-level views of future Grid architectures. The approach addresses fundamental Grid challenges and strives towards a future vision of the Grid where capabilities are made available as independent and dynamically assembled utilities, enabling run-time changes in the structure, behavior, and location of software. The presentation is made in terms of design heuristics, design patterns, and quality attributes, and is centered around the key concepts of co-existence, composability, adoptability, adaptability, changeability, and interoperability. The practical realization of the approach is illustrated by five case studies (recently developed Grid tools) high-lighting the most distinct aspects of these key concepts for each tool. The approach contributes to a healthy Grid ecosystem that promotes a natural selection of "surviving" components through competition, innovation, evolution, and diversity. In conclusion, this environment facilitates the use and composition of components on a per-component basis.

* By permission of Springer Verlag

Designing Service-Based Resource Management Tools for a Healthy Grid Ecosystem*

Erik Elmroth, Francisco Hernández, Johan Tordsson, and Per-Olov Östberg

Dept. of Computing Science and HPC2N
Umeå University, SE-901 87 Umeå, Sweden
{elmroth, hernandf, tordsson, p-o}@cs.umu.se

Abstract. We present an approach for development of Grid resource management tools, where we put into practice internationally established high-level views of future Grid architectures. The approach addresses fundamental Grid challenges and strives towards a future vision of the Grid where capabilities are made available as independent and dynamically assembled utilities, enabling run-time changes in the structure, behavior, and location of software. The presentation is made in terms of design heuristics, design patterns, and quality attributes, and is centered around the key concepts of co-existence, composability, adoptability, adaptability, changeability, and interoperability. The practical realization of the approach is illustrated by five case studies (recently developed Grid tools) high-lighting the most distinct aspects of these key concepts for each tool. The approach contributes to a healthy Grid ecosystem that promotes a natural selection of "surviving" components through competition, innovation, evolution, and diversity. In conclusion, this environment facilitates the use and composition of components on a per-component basis.

1 Introduction

In recent years, the vision of the Grid as the general-purpose, service-oriented infrastructure for provisioning of computing, data, and information capabilities has started to materialize in the convergence of Grid and Web services technologies. Ultimately, we envision a Grid with open and standardized interfaces and protocols, where independent Grids can interoperate, virtual organizations co-exist, and capabilities be made available as independent utilities.

However, there is still a fundamental gap between the technology used in major production Grids and recent technology developed by the Grid research community. While current research directions focus on user-centric and service-oriented infrastructure design for scenarios with millions of self-organizing nodes, current production Grids are often more monolithic systems with stronger inter-component dependencies.

* This research was conducted using the resources of the High Performance Computing Center North (HPC2N). Financial support has been provided by The Swedish Research Council (VR) under contract 621-2005-3667.

We present an approach to Grid infrastructure component development, where internationally established high-level views of future Grid architectures are put into practice. Our approach addresses the future vision of the Grid, while enabling easy integration into current production Grids. We illustrate the feasibility of our approach by presenting five case studies.

The outline of the rest of the paper is as follows. Section 2 gives further background information, including our vision of the Grid, a characterization of competitive factors for Grid software, and a brief review of internationally established conceptual views of future Grid architectures. Section 3 presents our approach to Grid infrastructure development, which complies with these views. The realization of this approach for specific components is illustrated in Section 4, with a brief presentation of five tools recently developed within the Grid Infrastructure Research & Development (GIRD) project [26]. These are Grid tools or toolkits for resource brokering [9–11], job management [7], workflow execution [8], accounting [16, 24], and Grid-wide fairshare scheduling [6].

2 Background and Motivation

Our approach to Grid infrastructure development is driven by the need and opportunity for a general-purpose infrastructure. This infrastructure should facilitate flexible and transparent access to distributed resources, dynamic composition of applications, management of complex processes and workflows, and operation across geographical and organizational boundaries. Our vision is that of a large evolving system, realized as a Service-Oriented Architecture (SOA) that enables provisioning of computing, data, and information capabilities as utility-like services serving business, academia, and individuals. From this point of departure, we elaborate on fundamental challenges that need to be addressed to realize this vision.

2.1 Facts of life in Grid environments

The operational context of a Grid environment is harsh, with heterogeneity in resource hardware, software, ownerships, and policies. The Grid is distributed and decentralized by nature, and any single point of control is impossible not only for scalability reasons but also since resources are owned by different organizations. Furthermore, as resource availability varies, resources may at any time join or leave the Grid. Information about the set of currently available resources and their status will always to some extent be incomplete or outdated.

Actors have different incentives to join the Grid, resulting in asymmetric resource sharing relationships. Trust is also asymmetric, which in scenarios with cross trust-domain orchestration of multiple resources that interact beyond the client-server model, gives rise to complex security challenges.

Demand for resources typically exceed supply, with contention for resources between users as a consequence. The Grid user community at large is disparate

in requirements and knowledge, necessitating the development of wide ranges of user interfaces and access mechanisms. All these complicating factors add up to an environment where errors are rule rather than exception.

2.2 A General-purpose Grid ecosystem

Recently, a number of organizations have expressed views on how to realize a single and fully open architecture for the future Grid. To a large extent, these expressions conform to a single view of a highly dynamic service-oriented infrastructure for general-purpose use.

One such view proposes the model of a healthy ecosystem of Grid components [25], where components occupy niches in the ecosystem and are designed for component-by-component selection by developers, administrators, and end-users. Components are developed by the Grid community at large and offer sensible functionality, available for easy integration in high-level tools or other software. In the long run, competition, innovation, evolution, and diversity lead to natural selection of "surviving" components, whereas other components eventually fade out or evolve into different niches.

European organizations, such as the Next Generation Grids expert group [12] and NESSI [23], have focused on a common architectural view for Grid infrastructure, possibly with a more emphasized business focus compared to previous efforts. Among their recommendations is a strong focus on SOAs where services can be dynamically assembled, thus enabling run-time changes in the structure, behavior, and location of software. The view of services as utilities includes directly and immediately usable services with established functionality, performance, and dependability. This vision goes beyond that of a prescribed layered architecture by proposing a multi-dimensional mesh of concepts, applying the same mechanisms along each dimension across the traditional layers.

In common for these views are, for example, a focus on composable components rather than monolithic Grid-wide systems, as well as a general-purpose infrastructure rather than application- or community-specific systems. Examples of usage range from business and academic applications to individual's use of the Grid. These visions also address some common issues in current production Grid infrastructures, such as interoperability and portability problems between different Grids, as well as limited software reuse. Before detailing our approach to Grid software design, which complies with the views presented above, we elaborate on key factors for software success in the Grid ecosystem.

2.3 Competitive factors for software in the Grid ecosystem

In addition to component-specific functional requirements, which obviously differ for different types of components, we identify a set of general quality attributes (also known as non-functional requirements) that successful software components should comply with. The success metrics considered here are the amount of users and the sustainability of software.

In order to attract the largest possible user community, usability aspects such as availability, ease of installation, understandability, and quality of documentation and support are important. With the dynamic and changing nature of Grid environments, flexibility and the ability to adapt and evolve is vital for the survival of a software component. Competitive factors for survival include changeability, adaptability, portability, interoperability, and integrability. These factors, along with mechanisms used to improve software quality with respect to them, are further discussed in Section 3. Other criteria, relating to sustainability, include the track record of both components and developers as well as the general reputation of the latter in the user community.

Quality attributes such as efficiency (with emphasis on scalability), reliability, and security also affect the software success rate in the Grid ecosystem. These attributes are however not further discussed herein.

3 Grid Ecosystem Software Development

In this section we present our approach to building software well-adjusted to the Grid ecosystem. The presentation is structured into five groups of software design heuristics, design patterns, and quality attributes that are central to our approach. All definitions are adapted to the Grid ecosystem environment, but are derived from, and conform to, the ISO/IEC 9126-1 standard [20].

3.1 Co-existence – Grid ecosystem awareness

Co-existence is defined as the ability of software to co-exist with other independent softwares in a shared resource environment. The behavior of a component well adjusted to the Grid ecosystem is characterized by non-intrusiveness, respect for niche boundaries, replaceability, and avoidance of resource overconsumption.

When developing new Grid components, we identify the purpose and boundaries of the corresponding niches in order to ensure the components' place and role in the ecosystem. By stressing non-intrusiveness in the design, we strive to ensure that new components do not alter, hinder, or in any other way affect the function of other components in the system. While the introduction of new software into an established ecosystem may, through fair competition, reshape, create, or eliminate niches, it is still important for the software to be able to cooperate and interact with neighboring components.

By the principle of decentralization, it is crucial to avoid making assumptions of omniscient nature and not to rely on global information or control in the Grid. By designing components for a user-centric view of systems, resources, component capabilities, and interfaces, we emphasize decentralization and facilitate component co-existence and usability.

3.2 Composability – software reuse in the Grid ecosystem

Composability is defined as the capability of software to be used both as individual components and as building blocks in other systems. As systems may

themselves be part of larger systems, or make use of other systems' components, composability becomes a measure of usefulness at different levels of system design. Below, we present some design heuristics that we make use of in order to improve software composability.

By designing components and component interactions in terms of interfaces rather than functionality, we promote the creation of components with well-defined responsibilities and provision for module encapsulation and interface abstraction. We strive to develop simple, single-purpose components achieving a distinct separation of concerns and a clear view of service architectures. Implementation of such components is faster and less error-prone than more complex designs. Autonomous components with minimized external dependencies make composed systems more fault tolerant as their distributed failure models become simpler.

Key to designing composable software is to provision for software reuse rather than reinvention. Our approach, leading to generic and composable tools well adjusted to the Grid ecosystem, encourages a model of software reuse where users of components take what they need and leave the rest. Being decentralized and distributed by nature, SOAs have several properties that facilitate the development of composable software.

3.3 Adoptability – Grid ecosystem component usability

Adoptability is a broad concept enveloping aspects such as end-user usability, ease of integration, ease of installation and administration, level of portability, and software maintainability. These are key factors for determining deployment rate and niche impact of a software.

As high software usability can both reduce end-user training time and increase productivity, it has significant impact on the adoptability of software. We strive for ease of system installation, administration, and integration (e.g., with other tools or Grid middlewares), and hence reduce the overhead imposed by using the software as stand-alone components, end-user tools, or building blocks in other systems. Key adoptability factors include quality of documentation and client APIs, as well as the degree of openness, complexity, transparency and intrusiveness of the system.

Moreover, high portability and ease of migration can be deciding factors for system adoptability.

3.4 Adaptability and Changeability – surviving evolution

Adaptability, the ability to adapt to new or different environments, can be a key factor for improving system sustainability. *Changeability*, the ability for software to be changed to provide modified behavior and meet new requirements, greatly affects system adaptability.

By providing mechanisms to modify component behavior via configuration modules, we strive to simplify component integration and provide flexibility in,

and ease of, customization and deployment. Furthermore, we find that the use of policy plug-in modules which can be provided and dynamically updated by third parties are efficient for making systems adaptable to changes in operational contexts. By separating policy from mechanism, we facilitate for developers to use system components in other ways than originally anticipated and software reuse can thus be increased.

3.5 Interoperability – interaction within the Grid ecosystem

Interoperability is the ability of software to interact with other systems. Our approach includes three different techniques for making our components available, making them able to access other Grid resources, and making other resources able to access our components, respectively. Integration of our components typically only requires the use of one or two of these techniques.

Whenever feasible, we leverage established and emerging Web and Grid services standards for interfaces, data formats, and architectures. Generally, we formulate integration points as interfaces expressing required functionality rather than reflecting internal component architecture. Our components are normally made available as Grid services, following these general principles.

For our components to access resources running different middlewares, we combine the use of customization points and design patterns such as Adapter and Chain of Responsibility [15]. Whenever possible, we strive to embed the customization points in our components, simplifying component integration with one or more middlewares.

In order to make existing Grid softwares able to access our components, we strive to make external integration points as few, small, and well-defined as possible, as these modifications need to be applied to external softwares.

4 Case Studies

We illustrate our approach to software development by brief presentations of five tools or toolkits recently developed in the GIRD project [26]. The presentations describe the overall tool functionality and high-light the most significant characteristics related to the topics discussed in Section 3.

All tools are built to operate in a decentralized Grid environment with no single point of control. They are furthermore designed to be non-intrusive and can coexist with alternative mechanisms. To enhance adoptability of the tools, user guides, administrator manuals, developer APIs, and component source code are made available online [26]. As these adoptability measures are common for all projects, the adoptability characteristics are left out of the individual project presentations.

The use of SOAs and Web services naturally fulfills many of the composability requirements outlined in Section 3. The Web service toolkit used is the Globus Toolkit 4 (GT4) Java WS Core, which provides an implementation of the Web Services Resource Framework (WSRF). Notably, the fact that our tools are made

available as GT4-based Web services should not be interpreted as been built primarily for use in GT4-based Grids. On the contrary, their design is focused on generality and ease of middleware integration.

4.1 Job Submission Service (JSS)

The JSS is a feature-rich, standards-based service for cross-middleware job submission, providing support, e.g., for advance reservations and co-allocation. The service implements a decentralized brokering policy, striving to optimize the job performance for individual users by minimizing the response time for each submitted job. In order to do this, the broker makes an a priori estimation of the whole, or parts of, the Total Time to Delivery (TTD) for all resources of interest before making the resource selection [9–11].

Co-existence: The non-intrusive decentralized resource broker handles each job isolated from the jobs of other users. It can provide quality of service to end-users despite the existence of competing job submission tools.

Composability: The JSS is composed of several modules, each performing a well-defined task in the job submission process, e.g., resource discovery, reservation negotiation, resource selection, and data transfer.

Changeability and adaptability: Users of the JSS can specify additional information in job request messages to customize and fine-tune the resource selection process. Developers can replace the resource brokering algorithms with alternative implementations.

Interoperability: The architecture of the JSS is based on (emerging) standards such as JSDL, WSRF, WS-Agreement, and GLUE. It also includes customization points, enabling the use of non-standard job description formats, Grid information systems, and job submission mechanisms. The latter two can be interfaced despite differences in data formats and protocols. By these mechanisms, the JSS can transparently submit jobs to and from GT4, NorduGrid/ARC, and LCG/gLite.

4.2 Grid Job Management Framework (GJMF)

The GJMF [7] is a framework for efficient and reliable processing of Grid jobs. It offers transparent submission, control, and management of jobs and groups of jobs on different middlewares.

Co-existence: The user-centric GJMF design provides a view of exclusive access to each service and enforces a user-level isolation which prohibits access to other users' information. All services in the framework assume shared access to Grid resources. The resource brokering is performed without use of global information, and includes back-off behaviors for Grid congestion control on all levels of job submission.

Composability: Orchestration of services with coherent interfaces provides transparent access to all capabilities offered by the framework. The functionality for job group management, job management, brokering, Grid information system access, job control, and log access are separated into autonomous services.

Changeability and adaptability: Configurable policy plug-ins in multiple locations allow customization of congestion control, failure handling, progress monitoring, service interaction, and job (group) prioritizing mechanisms. Dynamic service orchestration and fault tolerance is provided by each service being capable of using multiple service instances. For example, the job management service is capable of using several services for brokering and job submission, automatically switching to alternatives upon failures.

Interoperability: The use of standardized interfaces such as JSDL as job description format, OGSA BES for job execution, and OGSA RSS for resource selection improves interoperability and replaceability.

4.3 Grid Workflow Execution Engine (GWEE)

The GWEE [8] is a light-weight and generic workflow execution engine that facilitates the development of application-oriented end-user workflow tools. The engine is light-weight in that it focuses only on workflow execution and the corresponding state management. This project builds on experiences gained while developing the Grid Automation and Generative Environment (GAUGE) [19, 17].

Co-existence: The engine operates in the narrow niche of workflow execution. Instead of attempting to replace other workflow tools, the GWEE provides a means for accessing advanced capabilities offered by multiple Grid middlewares. The engine can process multiple workflows concurrently without them interfering with each other. Furthermore, the engine can be shared among multiple users, but only the creator of a workflow instance can monitor and control that workflow.

Composability: The main responsibilities of the engine, managing task dependencies, processing tasks on Grid resources, and managing workflow state, are performed by separate modules.

Adaptability and Changeability: Workflow clients can monitor executing workflows both by synchronous status requests and by asynchronous notifications. Different granularities of notifications are provided to support specific client requirements – from a single message upon workflow completion to detailed updates for each task state change.

Interoperability: The GWEE is made highly interoperable with different middlewares and workflow clients through the use of two types of plug-ins. Currently, it provides middleware plug-ins for execution of computational tasks in GT4 and in the GJMF, as well as GridFTP file transfers. It also provides plug-ins for transforming workflow languages into its native language, as currently has been done for the Karajan language. The Chain of Responsibility design pattern allows concurrent usage of multiple implementations of a particular plug-in.

4.4 SweGrid Accounting System (SGAS)

SGAS allocates Grid capacity between user groups by coordinated enforcement of Grid-wide usage limits [24, 16]. It employs a credit-based allocation model

where Grid capacity is granted to projects via Grid-wide quota allowances. The Grid resources collectively enforce these allowances in a soft, real-time manner. The main SGAS components are a Bank, a logging service (LUTS), and a quota-aware authorization tool (JARM), the latter to be integrated on each Grid resource.

Co-existence: SGAS is built as stand-alone Grid services with minimal dependencies on other software. Normal usage is not only non-intrusive to other software but also to usage policies, as resource owners retain ultimate control over local resource policies, such as strictness of quota enforcement.

Composability: There is a distinct separation of concerns between the Bank and the LUTS, for managing usage quotas and logging usage data, respectively. They can each be used independently.

Changeability and adaptability: The Bank can be used to account for any type of resource consumption and with any price-setting mechanism, as it is independent of the mapping to the abstract "Grid credit" unit used. The Bank can also be changed from managing pre-allocations to accumulating costs for later billing. The JARM provides customization points for calculating usage costs based on different pricing models. The tuning of the quota enforcement strictness is facilitated by a dedicated customization point.

Interoperability: The JARM has plug-in points for middleware-specific adapter code, facilitating integration with different middleware platforms, scheduling systems, and data formats. The middleware integration is done via a SOAP message interceptor in GT4 GRAM and via an authorization plug-in script in the NorduGrid/ARC GridManager. The LUTS data is stored in the OGF Usage Record format.

4.5 Grid-Wide Fairshare Scheduling System (FSGrid)

FSGrid is a Grid-wide fairshare scheduling system that provides three-party QoS support (user, resource-owner, VO-authority) for enforcement of locally and globally scoped share policies [6]. The system allows local resource capacity as well as global Grid capacity to be logically divided among different groups of users. The policy model is hierarchical and sub-policy definition can be delegated so that, e.g., a VO can partition its share among its projects, which in turn can divide their shares among users.

Co-existence: The main objective of FSGrid is to facilitate for distributed resources to collaboratively schedule jobs for Grid-wide fairness. FSGrid is non-intrusive in the sense that resource owners retain ultimate control of how to perform the scheduling on their local resources.

Composability: FSGrid includes two stand-alone components with clearly separated concerns for maintaining a policy tree and to log usage data, respectively. In fact, the logging component in current use is the LUTS originally developed for SGAS, illustrating the potential for reuse of that component.

Changeability and adaptability: A customizable policy engine is used to calculate priority factors based on a runtime policy tree with information about

resource pre-allocations and previous usage. The priority calculation can be customized, e.g., in terms of length, granularity, and rate of aging of usage history. The administration of the policy tree is flexible as sub-policy definition can be delegated to, e.g., VOs and projects.

Interoperability: Besides the integration of the LUTS (see Section 4.4), FSGrid includes a single external point of integration, as a fair-share priority factor call-out to FSGrid has to be integrated in the local scheduler on each resource.

5 Related Work

Despite the large amount of Grid related projects to date, just a few of these have shared their experiences regarding software design and development approaches. Some of these projects have focused on software architecture. In a survey by Filkenstein et al. [13], existing data-Grids are compared in terms of their architectures, functional requirements, and quality attributes. Cakic et al. [2] describe a Grid architectural style and a light-weight methodology for constructing Grids. Their work is based on a set of general functional requirements and quality attributes that derives an architectural style that includes information, control, and execution. Mattmann et al. [22] analyze software engineering challenges for large-scale scientific applications, and propose a general reference architecture that can be instantiated and adapted for specific application domains. We agree on the benefits obtained with a general architecture for Grid components to be instantiated for specific projects, however, our focus is on the inner workings of the components making up the architecture.

The idea of software that evolves due to unforeseen changes in the environment also appears in the literature. In the work by Smith et al. [3], the way software is modified over time is compared with Darwinian evolution. In this work, the authors discuss the best-of-breed approach, where an organization collects and assembles the most suitable software component from each niche. The authors also construct a taxonomy of the "species" of enterprise software. A main difference between this work and our contribution is that our work focuses on software design criteria.

Other high-level visions of Grid computing include that of interacting autonomous software agents [14]. One of the characteristics of this vision is that software engineering techniques employed for software agents can be reused with little or no effort if the agents encompasses the service's vision [21]. A different view on agent-based software development for the Grid is that of evolution based on competition between resource brokering agents [4]. These projects differ from our contribution as our tools have a stricter focus on functionality (being well-adjusted to their respective niches).

Finally, it is also important to notice that there are a number of tools that simplify the development of Grid software. These tools facilitate, for example, implementation [18], unit testing [5], and automatic integration [1].

6 Concluding Remarks

We explore the concept of the Grid ecosystem, with well-defined niches of functionality and natural selection (based on competition, innovation, evolution, and diversity) of software components within the respective niches. The Grid ecosystem facilitates the use and composition of components on a per-component basis. We discuss fundamental requirements for software to be well-adjusted to this environment and propose an approach to software development that complies with these requirements. The feasibility of our approach is demonstrated by five case studies. Future directions for this work include further exploration of processes and practices for development of Grid software.

7 Acknowledgements

We acknowledge Magnus Eriksson for valuable feedback on software engineering standardization matters.

References

1. M-E. Bégin, G. Diez-Andino, A. Di Meglio, E. Ferro, E. Ronchieri, M. Selmi, and M. Zurek. Build, configuration, integration and testing tools for large software projects: ETICS. In N. Guelfi and D. Buchs, editors, *Rapid Integration of Software Engineering Techniques,* LNCS 4401, pp. 81–97. Springer-Verlag, 2007.
2. J. Cakic and R. F. Paige. Origins of the Grid architectural style. In *Engineering of Complex Computer Systems. 11th IEEE Int. Conference, IECCS 2006*, pp. 227–235. IEEE CS Press, 2006.
3. J. Smith David, W. E. McCarthy, and B. S. Sommer. Agility – the key to survival of the fittest in the software market. *Commun. ACM*, 46(5):65–69, 2003.
4. C. Dimou and P. A. Mitkas. An agent-based metacomputing ecosystem. http://issel.ee.auth.gr/ktree/Documents/Root Folder/ISSEL/Publications/Biogrid An Agent-based Metacomputing Ecosystem.pdf, visited October 2007.
5. A. Duarte, W. Cirne, F. Brasileiro, and P. Machado. GridUnit: software testing on the Grid. In K.M. Anderson, editor, *Software Engineering. 28th Int. Conference, ICSE 2006*, pp. 779–782. ACM Press, 2006.
6. E. Elmroth and P. Gardfjäll. Design and evaluation of a decentralized system for Grid-wide fairshare scheduling. In H. Stockinger et al., editors, *First International Conference on e-Science and Grid Computing*, pp. 221–229. IEEE CS Press, 2005.
7. E. Elmroth, P. Gardfjäll, A. Norberg, J. Tordsson, and P-O. Östberg. Designing general, composable, and middleware-independent Grid infrastructure tools for multi-tiered job management. In T. Priol and M. Vaneschi, editors, *Towards Next Generation Grids*, pp. 175–184. Springer-Verlag, 2007.
8. E. Elmroth, F. Hernández, and J. Tordsson. A light-weight Grid workflow execution engine enabling client and middleware independence. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics. 7th Int. Conference, PPAM 2007*. Lecture notes in Computer Science, Springer Verlag, 2007 (to appear).
9. E. Elmroth and J. Tordsson. An interoperable, standards-based Grid resource broker and job submission service. In H. Stockinger et al., editors, *First International Conference on e-Science and Grid Computing*, pp. 212–220. IEEE CS Press, 2005.

10. E. Elmroth and J. Tordsson. A standards-based Grid resource brokering service supporting advance reservations, coallocation and cross-Grid interoperability. *Submitted to Concurrency and Computation: Practice and Experience*, 2006.
11. E. Elmroth and J. Tordsson. A Grid resource brokering algorithms enabling advance reservations and resource selection based on performance predictions. *Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications*, 2008, to appear.
12. Expert Group on Next Generation Grids 3 (NGG3). Future for European Grids: Grids and service oriented knowledge utilities. Vision and research directions 2010 and beyond, 2006. ftp://ftp.cordis.lu/pub/ist/docs/grids/ngg3_eg_final.pdf, visited October 2007.
13. A. Finkelstein, C. Gryce, and J. Lewis-Bowen. Relating requirements and architectures: a study of data-grids. *J. Grid Computing*, 2(3):207–222, 2004.
14. I. Foster, N. R. Jennings, and C. Kesselman. Brain meets brawn: why Grid and agents need each other. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, pp. 8–15. IEEE CS Press, 2004.
15. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
16. P. Gardfjäll, E. Elmroth, L. Johnsson, O. Mulmo, and T. Sandholm. Scalable Grid-wide capacity allocation with the SweGrid Accounting System (SGAS). *Concurrency and Computation: Practice and Experience*, (accepted) 2007.
17. Z. Guan, F. Hernández, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy, and Y. Liu. Grid-Flow: a Grid-enabled scientific workflow system with a petri-net-based interface. *Concurrency Computat.: Pract. Exper.*, 18(10):1115–1140, 2006.
18. S. Hastings, S. Oster, S. Langella, D. Ervin, T. Kurc, and J. Saltz. Introduce: an open source toolkit for rapid development of strongly typed Grid services. *J. Grid Computing*, 5(4):407–427, 2007.
19. F. Hernández, P. Bangalore, J. Gray, Z. Guan, and K. Reilly. GAUGE: Grid Automation and Generative Environment. *Concurrency Computat.: Pract. Exper.*, 18(10):1293–1316, 2006.
20. ISO/IEC. Software engineering - Product quality - Part 1: Quality model. International standard ISO/IEC 9126-1. 2001.
21. P. Leong, C. Miao, and B-S. Lee. Agent oriented software engineering for Grid computing. In *Cluster Computing and the Grid. 6th IEEE Int. Symposium, CCGRID 2006*. IEEE CS Press, 2006.
22. C. A. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes. A software architecture-based framework for highly distributed and data intensive scientific applications. In K.M. Anderson, editor, *Software Engineering. 28th Int. Conference, ICSE 2006*, pp. 721–730. ACM Press, 2006.
23. Networked European Software and Services Initiative (NESSI). http://www.nessi-europe.com, visited October 2007.
24. T. Sandholm, P. Gardfjäll, E. Elmroth, L. Johnsson, and O.Mulmo. A service-oriented approach to enforce Grid resource allocations. *International Journal of Cooperative Information Systems*, 15(3):439–459, 2006.
25. The Globus Project. An "ecosystem" of Grid components. http://www.globus.org/grid_software/ecology.php, visited October 2007.
26. The Grid Infrastructure Research & Development (GIRD) project. Umeå University, Sweden. http://www.gird.se, visited October 2007.

III

Paper III

Dynamic and Transparent Service Compositions Techniques for Service-Oriented Grid Architectures*

Erik Elmroth and Per-Olov Östberg

Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden
{elmroth, p-o}@cs.umu.se
http://www.gird.se

Abstract: With the introduction of the Service-Oriented Architecture design paradigm, service composition has become a central methodology for developing Grid software. We present an approach to Grid software development consisting of architectural design patterns for service de-composition and service re-composition. The patterns presented can each be used individually, but provide synergistic effects when combined as described in a unified framework. Software design patterns are employed to provide structure in design for service-based software development. Service APIs and immutable data wrappers are used to simplify service client development and isolate service clients from details of underlying service engine architectures. The use of local call structures greatly reduces inter-service communication overhead for co-located services, and service API factories are used to make local calls transparent to service client developers. Light-weight and dynamically replaceable plug-ins provide structure for decision support and integration points. A dynamic configuration scheme provides coordination of service efforts and synchronization of service interactions in a user-centric manner. When using local calls and dynamic configuration for creating networks of cooperating services, the need for generic service monitoring solutions becomes apparent and is addressed by service monitoring interfaces. We present these techniques along with their intended use in the context of software development for service-oriented Grid architectures.

Key words: Grid software development, Service-Oriented Architecture, Web Service composition, Design patterns, Grid ecosystem.

* By permission of Crete University Press

DYNAMIC AND TRANSPARENT SERVICE COMPOSITION TECHNIQUES FOR SERVICE-ORIENTED GRID ARCHITECTURES*

Erik Elmroth and Per-Olov Östberg
Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden
{elmroth, p-o}@cs.umu.se
http://www.gird.se

Abstract With the introduction of the Service-Oriented Architecture design paradigm, service composition has become a central methodology for developing Grid software. We present an approach to Grid software development consisting of architectural design patterns for service de-composition and service re-composition. The patterns presented can each be used individually, but provide synergistic effects when combined as described in a unified framework. Software design patterns are employed to provide structure in design for service-based software development. Service APIs and immutable data wrappers are used to simplify service client development and isolate service clients from details of underlying service engine architectures. The use of local call structures greatly reduces inter-service communication overhead for co-located services, and service API factories are used to make local calls transparent to service client developers. Light-weight and dynamically replaceable plug-ins provide structure for decision support and integration points. A dynamic configuration scheme provides coordination of service efforts and synchronization of service interactions in a user-centric manner. When using local calls and dynamic configuration for creating networks of cooperating services, the need for generic service monitoring solutions becomes apparent and is addressed by service monitoring interfaces. We present these techniques along with their intended use in the context of software development for service-oriented Grid architectures.

Keywords: Grid software development, Service-Oriented Architecture, Web Service composition, Design patterns, Grid ecosystem.

*Financial support has been received from The Swedish Research Council (VR) under contract number 621-2005-3667. This research was conducted using the resources of the High Performance Computing Center North (HPC2N).

1. Introduction

With the introduction of service-oriented computing and the increased popularity of the Service-Oriented Architecture (SOA) design paradigm, service composition has become a key methodology for building distributed, service-based applications. In this work we outline the foundational concepts of our SOA development methodology, introducing and describing a number of techniques targeting the development of robust, scalable, and flexible Grid software. We investigate development methodologies such as design patterns, call optimizations, plug-in structures, and techniques for dynamic service configuration. When combined, these techniques make up the foundation of an approach for composable Web Services that are to be used in Grid SOA environments. The techniques are here presented in Grid Web Service development scenarios.

The outline of the paper is the following: A motivation and overview of our work is presented in Section 2. A more detailed introduction to the concept and aspects of service composition is given in Section 3, after which we present architectural design patterns used to address these concepts in Section 4. Finally, a brief survey of related work is presented in Section 5, followed by conclusions in Section 6 and acknowledgements.

2. Motivation and Overview

The work presented here has grown out of a need for flexible development techniques for the creation of efficient and composable Web Services. Current Grid systems employ more and more SOA-based software where scalability is a key requirement on all levels of system design, including in the development process. Service composition techniques, which employ services as building blocks in applications through the use of service aggregators, often create systems that impose substantial overhead in terms of memory requirements and execution time. Although Web Services are distributed by definition, utilizing them dynamically is often a process with lack of flexibility and transparency. The complexity of SOAP message processing alone can present impracticalities to SOA developers, as a single Web Service that exchanges large or frequent messages may in itself negatively impact the performance of other, co-located, services.

In our approach, we address these issues in two ways; by providing *flexible and transparent structures for dynamic reconfiguration of (networks of) services*, and by outlining *development patterns for optimization of interaction between co-located services and service components*. More specifically, we provide a set of architectural software design patterns for service APIs, local call structures, flexible plug-in and configuration architectures, and service monitoring facilities. Combined, these techniques make up a framework that serves to reduce the temporal and spatial system footprints (time of ex-

ecution and memory requirements, respectively) of co-located services, and provide for a software development model where dynamic service composition is made transparent to service client developers. The techniques presented are completely orthogonal to approaches using the Business Process Execution Language for Web Services (BPEL4WS) [9], Web Service Choreography Interface (WSCI) and similar techniques for service composition, and the resulting Web Services can be used in a range of service orchestration and choreography scenarios.

The approach presented here has emerged from work on the Grid Job Management Framework (GJMF) [3], a software developed in the Grid Infrastructure Research & Development (GIRD) [14] project. As a key part of the GIRD project, we investigate software development methodologies for the Grid ecosystem [13], an ecosystem of niched software components where component survival follows from evolution and natural selection [5], and a Grid built on such components. We primarily develop software in Java using the Globus Toolkit 4 (GT4) Java WS Core [7], which contains an implementation of the Web Services Resource Framework (WSRF).

3. Service Composition Techniques

Two approaches to service composition are service orchestration and service choreography. As the needs and practices in Grid and Web Service software development vary, clear definitions of the terms are yet to be fully agreed upon. In Peltz [11], service orchestration and choreography are described as approaches to create business processes from composite Web Services. Furthermore, service orchestration is detailed to be concerned with the message-level interactions of (composite and constituent) Web Services, describing business logic and goals to be realized, and representing the control flow of a single party in the message exchange. Service choreography is defined in terms of public message exchanges between multiple parties, to be more collaborative by nature, and taking a system-wide perspective of the interaction, allowing involved parties to describe their respective service interactions themselves.

Our approach to service composition is primarily concerned with transparency and scalability in dynamic service usage. We investigate techniques for developing Web Services in a dynamic and efficient manner, Web Services that can be transparently de-composed and dynamically re-composed.

3.1 Transparent Service De-Composition

At system level, Web Services are defined in terms of their interfaces without making any assumptions about the internal workings of the service functionality. In SOA design, focus is on service interactions rather than service design, and a service set providing required functionality is assumed to exist.

In the development of individual services, the structured software development approach is often hindered by the practical limitations of Web Services. By recursively subdividing the functionality of a composite Web Service, a process here referred to as service de-composition, it is often possible to identify functionality that can be reused by other services if exposed as Web Services. However, response times and memory requirements of Web Services often make it impractical to expose core functionality in this manner.

We address this issue with a framework for call optimizations, which allows software components to simultaneously and transparently function as both Web Services and local Java objects in co-located services. Small, single purpose components are easier to develop and maintain, less error-prone, and often better matched to standardized functionality [5]. By mediating the technical limitations imposed by Web Services, the use of these techniques provides a programming model that offers transparency in the use of services in distributed object-oriented modelling. As these techniques are optimizations of calls between co-located services, they are completely orthogonal to, and can be used in conjunction with, service composition techniques such as BPEL4WS, WS-AtomicTransaction and WS-Coordination.

A recent example of the application of these techniques is the construction of a workflow execution engine. A workflow engine typically contains functionality for, e.g., workflow state coordination, task submission, job monitoring, and log maintenance. By de-composing the engine functionality into a set of cooperating services rather than a large, monolithic structure, reusable software components are created and can be exposed as Web Services. The use of the proposed call optimization framework makes the de-composition process transparent to developers, provides improved fault tolerance though the use of multiple service providers (for, e.g., job submission), and preserves the performance of a single software component (an example from [3] and [4]).

3.2 Dynamic Service Re-Composition

Given a mechanism for service de-composition, a natural next step is to identify mechanisms to facilitate dynamic and transparent reconfiguration of Web Services during runtime, here referred to as service re-composition. In most service orchestration and choreography scenarios, this can be achieved using late service binding and dynamic discovery of services. As in the case of service de-composition, natural inefficiencies in these techniques may discourage developers from using them to their full potential.

We employ a scheme for dynamic configuration of services into networks of smaller, constituent services. Once again, this is a lower-level optimization of the service interactions that does not compete with traditional service orchestration techniques, but can rather co-exist with them. The scheme (out-

lined in Section 4.5) consists of services keeping local copies of configuration modules that may at any time be updated by external means. All services consult their respective configuration modules when making decisions about what plug-ins to load, which services to interact with, etc. Once a transaction with another service has been initialized, information about this process is maintained separately. The benefits of using this scheme include increased flexibility in development and deployment; access to transparent mechanisms for redundancy, fault tolerance and load balancing; and ease of administration.

A practical example of the application of this technique is the internal workings of the GJMF [3]. All services in the framework are configured using the dynamic configuration technique described, allowing services to reshape the network of services that collectively make up the higher-level functionality of the framework. Note that this technique is completely transparent to service orchestration and choreography approaches as it operates on a lower level. In fact, in a service orchestration scenario it is expected that the configuration data would be provided the service by the orchestration mechanism itself.

4. Architectural Design Patterns

The techniques presented here are intended to be used as architectural design patterns to facilitate the development of scalable and composable Web Services. Though they may be used individually, the techniques have proven to provide synergistic effects when combined, both in development and deployment.

4.1 Software Design Patterns

In architecture design, we extensively employ the use of established software design patterns [8] for the creation of efficient and reusable software components with small system footprints. The Flyweight, Builder, and Immutable patterns are used to create lean and efficient data structures. Patterns such as the Singleton, Factory Method, and Observer are deployed in a variety of scenarios to create dynamic and composable software components. To enable components to dynamically update and replace functionality, we use the Strategy, Abstract Factory, Model-View-Controller, and Chain of Responsibility patterns. The Facade, Mediator, Proxy, Command, Broker, Memento, and Adapter patterns are used to facilitate, organize, abstract, and virtualize component interaction.

4.2 Immutable Wrappers & Service APIs

In this section, we present patterns used for data representation and service APIs. The techniques presented combine design patterns and design heuristics, and are aimed to simplify service client development and facilitate the techniques presented in the following sections.

Passive data objects such as job and workflow descriptions are rarely modified once created. A useful pattern for the representation of passive data objects is to construct immutable data wrapper classes that provide abstraction of the data interface. Embedding data validation in wrappers also simplifies data handling, and is considered good practice in defensive programming. Typically, in Web Service development, data representations are specified in service descriptions and stub types are generated from WSDL. The use of wrappers around stub types provides the additional benefit of encapsulating service engine-specific stub behavior and incompatibility issues between service engines. The practice of assigning unique identifiers, e.g., in the form of Universally Unique Identifiers (UUID), to data instances facilitates the use of persistence models such as Java object serialization and GT4 resource persistence, and provides services and clients with synchronized data identifiers. By creating a service-specific data translation component, it is possible to help service instances to translate stubs to wrappers, and vice versa. The use of immutable wrappers and a designated translation component is illustrated in Figure 1. In the figure, software components are illustrated as boxes, component interactions as solid arrows, and dynamically discovered and resolved interactions as arrows with dotted lines. Note that the service client APIs and back-end make use of immutable data wrappers and are isolated from the stubs by the stub type translator.

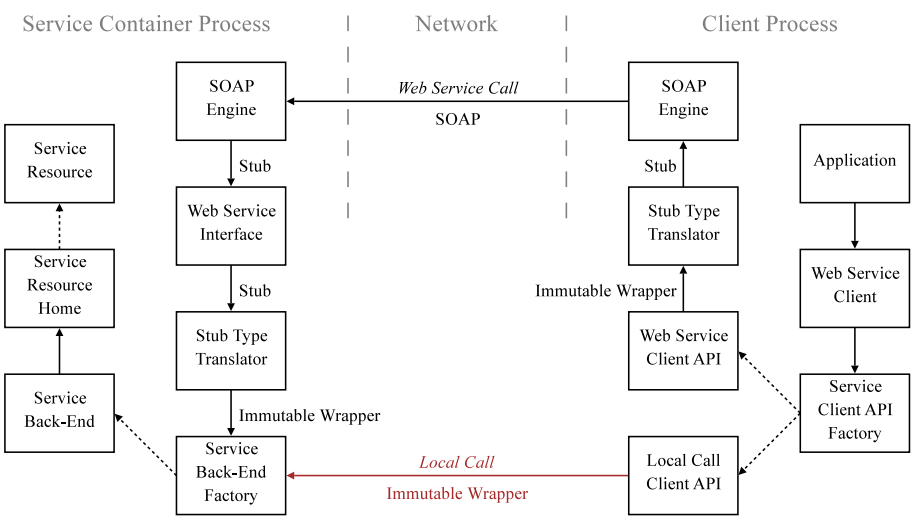


Figure 1. Illustration of local call optimizations for co-located services; dynamic resolution of service client APIs, back-ends and resources; and the use of immutable data wrappers.

In the interest of software usability for developers, it is recommended to provide client APIs with each Web Service. This practice allows developers

with limited experience of Web Service development to use SOAs transparently, and offers reference implementations detailing service use. In service APIs, a programming language interface, rather than a concrete implementation, should be used to abstract the service interface. The API interface should furthermore make strict use of wrapped data types in order to isolate it from changes in underlying architectures, e.g., Web Service engine replacement.

4.3 Local Call Structures

The use of local call structures facilitates the development of components that can be used both as generic objects and stand-alone Web Services. As illustrated in Figure 1, we propose a structure where Web Service implementations are divided into separate components for service data, interface, and implementation. Here, the service data are modeled as WSRF resources, which are dynamically resolved through the resource home using unique resource identifiers. The service interface contains the actual Web Service interfaces, and handles call semantics, stub type translation, and parameter validation issues. The service implementation back-end houses the service logic. It is dynamically resolved using a service back-end factory that instantiates a unique service implementation for each user, providing complete user-level isolation of service capabilities and resources.

Separating the service interface from the service implementation makes it possible for service clients that are co-located with the service (i.e., other services running in the same service container) to directly access the service logic. As illustrated in Figure 1, local calls bypass resource consuming data translations, credentials delegations, and Web Service invocations. For service notification invocations, the process is mediated through a notification dispatcher that dynamically resolves service resources and provides optional notification filtering and translation. Note that this scheme allows the GT4 resource persistence mechanisms to function unhindered, and remains compatible with the WSRF and WS-Notification specifications.

The resolution of the service back-end, and the local call logic, are encapsulated and made transparent to developers through the use of service client API classes. A service API factory provides appropriate service API implementations based on inspection of the service URLs, e.g., comparing IP address and port number to the local service containers configuration to determine if a local call can be made and wrapping the use of multiple (stateless) service instances into a single, logical service client interface. The service API factory makes this process transparent to the developer, which provides a set of service URLs to retrieve a service client interface.

The use of local calls efficiently optimizes communication between co-located services, but the main benefit of the technique is that it allows for

transparent de-composition of service functionality into networks of services. This provides for a more flexible development model for services that can be dynamically re-composed with a minimum of overhead, a requirement for service networks that rely on state update notifications for service coordination.

4.4 Policy Advisor and Mechanism Provider Plug-Ins

For situations where modules are to be dynamically provided and reused within components, but not between them, we make use of dynamic plug-in structures. Made up by a combination of programming language interfaces and designated configuration points, plug-in modules are dynamically located and loaded, and are considered volatile in the sense that they are intended to be short-lived and dynamically replaceable.

Functionality provided by plug-ins can be divided into two major categories: policy advisors and mechanism providers. A policy advisor implementation is intended to function in a strict advisory capacity for scenarios where policy logic is too complex to be expressed in direct configuration. The typical role of a policy advisor is to provide decisions when asked specific questions (formulated by the plug-in interface). This type of plug-in is useful for decision support in, e.g., failure handling or job prioritization. Mechanism providers are typically used for interface abstraction and integration point exposure. These types of modules are used to provide, e.g., vendor-specific database accessors or alternative brokering algorithms for job submitters.

Plug-in implementations should be light-weight, refrain from causing side-effects, have short response times, be thread-safe, and use minimal amounts of memory. Services using plug-ins should acquire the modules dynamically for each use, and rely strictly on the plug-in interface for functionality. As plug-ins can be provided by third party developers, and dynamically provided over networks, the use of code signing techniques to maintain service integrity is advisable. Grid security solutions that deploy Public-Key Infrastructures (PKI) for associating X.509 certificates with users can also be used to provide key pairs for code signing. When services provide user-centric views of service functionality, per-user configuration of service mechanism is trivial to realize.

4.5 Dynamic Service Configuration

Configuration data for Web Services are typically expressed in XML and loaded from local configuration files. Semantic Web Services provide configuration metadata to facilitate a higher degree of automation in, primarily, service composition and choreography. Similar to this approach, we employ a simplistic architecture for dynamic configuration built on the interchange of configuration data between services, and customized configuration modules to be used within services. This approach allows services to be expressed as net-

works of services, and to dynamically adapt to changes in executional context in a way that can be utilized by semantic service aggregators.

Central to our configuration approach is a dynamically replaceable configuration module. Each service maintains a configuration module factory that instantiates configuration modules when needed. The manner in which data contained in the configuration modules are acquired is encapsulated in the factory and can alternate between, e.g., polling of configuration files, triggering in databases, querying of Grid Monitoring and Discovery Services, and notifications from dedicated configuration services.

Providing configuration data through dedicated configuration services allows for transparent configuration of multiple services, where each service requests configuration data based on current user identity and service location. Dedicated configuration services can monitor resource availability and perform, e.g., load balancing through dynamic reconfiguration of networks of cooperating services. In terms of administrational overhead, this technique can alleviate the managerial burden of administrating services as it provides a single point of configuration for multiple service containers. As the local call structures of Section 4.3 provide an automatic and transparent optimization of calls between co-located services, the configuration service may attempt to optimize inter-service usage by favoring cooperation between co-located services.

In this scheme, services should never maintain direct references to configuration modules, but rather rely on them as temporary factories for configuration data. Interpretation of configuration data, type conversions, and data validation are examples of tasks to be performed by configuration modules. The use of caching techniques for configuration modules, and the synchronization and acquisition of raw configuration data should be encapsulated in configuration module factories. As seen in Section 4.4, configuration data may also be supplied in the form of plug-ins, in which case the configuration module is responsible for the location and dynamic construction of these plug-ins. When providing sensitive data, the personalization techniques of Section 4.3 can be used to provide user-level isolation of service configuration.

4.6 Service Monitoring

The dynamic configuration solutions of Section 4.5 facilitate the deployment of composite Web Services as networks of services. For reasons of system transparency, it is equally important to make parts of this configuration available to service clients, e.g., as WSRF resource properties. Consider a client submitting workflows to a workflow execution service, which schedules and submits a Grid job for each workflow task. In the interest of system openness, the client should be provided means to trace job execution, e.g., from workflow down to computational resource level. By publishing job End-Point References (EPR),

or log service URLs, the service empowers clients with the ability to monitor and trace job execution.

As mentioned in Section 4.2, data entities are provided unique identifiers prior to Web Service submission. Using these identifiers as resource keys for corresponding WSRF resources in Web Services allow clients with knowledge of identifiers (and service URL) to create resource EPRs when needed. Stateful services expose interfaces for listing resources contained in the services. For efficiency, the information returned by these interfaces are limited to lists of data identifiers (UUIDs). To improve usability and ease of development for service clients, boiler-plate solutions for tools to monitor service content are provided with each service. Although not further explored here, it should be noted that these monitoring interfaces, as well as the wrappers and service APIs of Section 4.2, are well suited for use in web portals and directly usable in the JavaService Pages (JSP) environment.

5. Related Work

There exists numerous valuable contributions on how to design for service composition and orchestration within both the fields of Grid computing and service orientation. For reasons of brevity, however, this section only references a selected number of related publications that directly touch upon the concepts presented in our software development approach.

The authors of [6] provide a grouping of service composition strategies. Our approach, containing late service bindings and semi-automatic service interaction planning, falls into the semi-dynamic service composition strategies category of this model. Brief surveys of service orchestration and choreography techniques are given in [10] and [11], and an approach for developing pattern-based service coordination is presented in [15]. Our work focuses on design heuristics and patterns for dynamic and transparent service composition in Grid contexts, and is considered orthogonal to all these techniques. The authors of [2] investigate a framework for service composition using Higher Order Components. Here, component Web Service interface generation is automated, and services are dynamically configured and deployed. We consider this a different technique pursuing a similar goal, i.e., dynamic service composition.

The Globus Toolkit [7] and the Apache Axis Web Service engine both contain utilities for local call optimizations. The Axis engine provides an in-memory call mechanism, and the Globus Toolkit provides a configurable local invocation utility that performs dynamically resolved Java calls to methods in co-located services. These approaches provide a higher level of transparency in service development, whereas our approach focuses on transparency for service client developers. In terms of performance, direct Java calls are naturally faster than in-memory Web Service invocations, and the GT4 approach suffers additional

overhead for the dynamic invocation of methods compared to our approach. Additionally, GT4 does not currently support local invocations for notifications.

Recent approaches to Grid job monitoring are presented in [1] and [12], and are here included to illustrate service monitoring functionality in dynamically composable service networks. We strive to provide dynamic monitoring and traceability mechanisms that are usable in external service monitoring tools, rather than providing stand-alone service monitoring solutions.

6. Conclusions

We present an approach to Grid software development consisting of a number of architectural design patterns. These patterns, as presented in Section 4, provide a framework addressing service de- and re-composition. The patterns presented can each be used individually, but provide synergistic effects when combined into a framework. E.g., the unique identifiers of the immutable wrappers that are used in service client APIs can also be used as resource keys for service resources, providing a simple mechanism for client-service data synchronization. Additional examples of synergistic effects are the cooperative use of local call structures, dynamic configuration, plug-ins, and service monitoring techniques: Local call structures reduce service footprints to a level where services are usable for the creation of transparent service networks. As service APIs and service API factories make the use of local calls transparent, service client developers are given an automated mechanism for optimization of service interaction. Employing dynamic configuration techniques to exploit the transparency of local calls then further increases flexibility in service interaction and administration of multiple services. Plug-ins can in turn be used to represent policy decisions, i.e., configuration semantics too complex to be represented in direct configuration, to provide alternative mechanisms, and expose integration points in services. Parts of service configuration can be exposed through monitoring interfaces to provide system transparency and monitorability, and services can employ replaceable plug-ins to utilize customized monitoring mechanisms.

The patterns described provide individually useful mechanisms for system architecture, and are orthogonal in design to each other and related technologies. Combined, they provide a framework for building lean and efficient Web Services that can be used transparently in cooperative networks of services.

Acknowledgments

We are grateful to Johan Tordsson and the anonymous referees for providing valuable feedback on, and improving the quality of, this work.

References

[1] A. N. Duarte, P. Nyczyk, A. Retico, and D. Vicinanza. Global Grid monitoring: the EGEE/WLCG case. In *GMW '07: Proceedings of the 2007 workshop on Grid monitoring*, pages 9–16, New York, NY, USA, 2007. ACM.

[2] J. Dünnweber, S. Gorlatch, F. Baude, V. Legrand, and N. Parlavantzas. Towards automatic creation of Web Services for Grid component composition. In V. Getov, editor, *Proceedings of the Workshop on Grid Systems, Tools and Environments, 12 October 2005, Sophia Antipolis, France*, December 2006.

[3] E. Elmroth, P. Gardfjäll, A. Norberg, J. Tordsson, and P-O. Östberg. Designing general, composable, and middleware-independent Grid infrastructure tools for multi-tiered job management. In T. Priol and M. Vaneschi, editors, *Towards Next Generation Grids*, pages 175–184. Springer-Verlag, 2007.

[4] E. Elmroth, F. Hernández, and J. Tordsson. A light-weight Grid workflow execution engine enabling client and middleware independence. In R. Wyrzykowski et.al, editors, *Parallel Processing and Applied Mathematics. 7th Int. Conference, PPAM 2007*. Lecture Notes in Computer Science, Springer Verlag, 2007 (to appear).

[5] E. Elmroth, F. Hernández, J. Tordsson, and P-O. Östberg. Designing service-based resource management tools for a healthy Grid ecosystem. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics. 7th Int. Conference, PPAM 2007*. Lecture Notes in Computer Science, Springer-Verlag, 2007 (to appear).

[6] M. Fluegge, I. J. G. Santos, N. P. Tizzo, and E. R. M. Madeira. Challenges and techniques on the road to dynamically compose Web Services. In *ICWE '06: Proceedings of the 6th international conference on Web engineering*, pages 40–47, New York, NY, USA, 2006. ACM.

[7] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In H. Jin et al., editors, *IFIP International Conference on Network and Parallel Computing,* Lecture Notes in Computer Science 3779, pages 2–13. Springer-Verlag, 2005.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] IBM. Business Process Execution Language for Web Services, version 1.1. http://www.ibm.com/developerworks/library/specification/ws-bpel/, visited February 2008.

[10] N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, 08(6):51–59, 2004.

[11] C. Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, 2003.

[12] M. Ruda, A. Křenek, M. Mulač, J. Pospíšil, and Z. Šustr. A uniform job monitoring service in multiple job universes. In *GMW '07: Proceedings of the 2007 workshop on Grid monitoring*, pages 17–22, New York, NY, USA, 2007. ACM.

[13] The Globus Project. An "ecosystem" of Grid components. http://www.globus.org/grid_software/ecology.php, visited February 2008.

[14] The Grid Infrastructure Research & Development (GIRD) project. Umeå University, Sweden. http://www.gird.se, visited February 2008.

[15] C. Zirpins, W. Lamersdorf, and T. Baier. Flexible coordination of service interaction patterns. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 49–56, New York, NY, USA, 2004. ACM.

IV

Paper IV

Empowering a Flexible Application Portal with a SOA-based Grid Job Management Framework*

Erik Elmroth[1], Sverker Holmgren[2], Jonas Lindemann[3], Salman Toor[2], and Per-Olov Östberg[1]

[1] *Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden*
{elmroth, p-o}@cs.umu.se
http://www.gird.se

[2] *Dept. Information Technology, Uppsala University, Box 256, SE-751 05 Uppsala, Sweden*
{sverker.holmgren, salman.toor}@it.uu.se http://www.uu.se

[3] *LUNARC, Lund University, Box 117, SE-221 00, Sweden*
jonas.lindemann@lunarc.lu.se http://www.lu.se

Abstract: The complexity of simultaneously providing customized user interfaces and transparent Grid access has led to a situation where current Grid portals tend to either be tightly coupled to specific middlewares or only provide generic user interfaces. In this work, we build upon the methodology of the Grid Job Management Framework and propose a flexible and robust 3-tier integration architecture that decouples application interface customization from Grid job management. Furthermore, we illustrate the approach with a proof of concept integration of the Lunarc Application Portal, which here serves as both a framework for the creation of application-oriented user interfaces and a Grid portal, and the Grid Job Management Framework, a framework for transparent access to multiple Grid middlewares. The loosely coupled architecture facilitates creation of sophisticated user interfaces customized to enduser applications while preserving the middleware-independence of the job management framework. The integration architecture is presented together with brief introductions to the integrated systems, and a system evaluation is provided to demonstrate the flexibility of the architecture.

* By permission of Springer Verlag

Empowering a Flexible Application Portal with a SOA-based Grid Job Management Framework

Erik Elmroth[1], Sverker Holmgren[2], Jonas Lindemann[3], Salman Toor[2], and Per-Olov Östberg[1]

[1] Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden,
{elmroth, p-o}@cs.umu.se http://www.gird.se
[2] Dept. Information Technology, Uppsala University,
Box 256, SE-751 05 Uppsala, Sweden
{sverker.holmgren, salman.toor}@it.uu.se http://www.uu.se
[3] LUNARC, Lund University, Box 117, SE-221 00, Sweden
jonas.lindemann@lunarc.lu.se http://www.lu.se

Abstract. The complexity of simultaneously providing customized user interfaces and transparent Grid access has led to a situation where current Grid portals tend to either be tightly coupled to specific middlewares or only provide generic user interfaces. In this work, we build upon the methodology of the Grid Job Management Framework and propose a flexible and robust 3-tier integration architecture that decouples application interface customization from Grid job management. Furthermore, we illustrate the approach with a proof of concept integration of the Lunarc Application Portal, which here serves as both a framework for the creation of application-oriented user interfaces and a Grid portal, and the Grid Job Management Framework, a framework for transparent access to multiple Grid middlewares. The loosely coupled architecture facilitates creation of sophisticated user interfaces customized to end-user applications while preserving the middleware-independence of the job management framework. The integration architecture is presented together with brief introductions to the integrated systems, and a system evaluation is provided to demonstrate the flexibility of the architecture.

1 Introduction

The task of constructing complete, robust, and high-performing systems that simultaneously provide sophisticated user interfaces and transparent access to computational resources is inherently complex. The range of Grid middlewares available today combined with the amount of applications (potentially) running on Grids introduces additional complexity. Thus, current portal-oriented efforts towards this goal [8, 11, 13] typically yield solutions that provide application-oriented interfaces tightly coupled to specific Grid middlewares, or Grid middleware solutions accessible only through generic user interfaces.

In this work we explore an architectural design pattern for development of advanced end-user applications capable of middleware-agnostic Grid use. We extend the methodology of [6] to development of flexible Grid portals that combine

application-oriented user interfaces with transparent Grid access. A 3-tier integration architecture that abstracts Grid functionality and isolates user interfaces from job management is proposed, and the approach is illustrated by an integration of the Lunarc Application Portal, an application-oriented Grid portal, and the Grid Job Management Framework, a middleware-independent Grid job management system designed for this purpose. The integration of these systems provides a flexible architecture where user interfaces can be adapted to specific applications and abstracted beyond the details of the underlying middleware.

1.1 The Lunarc Application Portal

The Lunarc Application Portal (LAP) is a web-based portal for submitting jobs to Grid resources [16–18]. The portal is implemented in Python using the Webware for Python [21] application server, and utilizes (in the original design) ARC/arcLib [5] for submitting and controlling jobs. Webware is a lightweight application server providing multi-user session handling, servlets, and page rendering. Although Webware provides a built-in web server, most applications use the Apache web server and a special extension module, mod_webkit2, to forward HTTP requests to the Webware application server. The recommended way of running LAP is through an SSL-enabled Apache web server.

The LAP can be viewed both as a web portal and as a Python-based framework for implementation of customized user interfaces for Grid-enabled applications. The core implementation includes a set of modules that provide management of users and job definitions, security, middleware integration, and user interface rendering. The portal also provides a set of servlets for non-application oriented tasks such as job definition creation, job monitoring, and job control.

Support for new applications in LAP is offered through use of customization points and plug-ins. An LAP plug-in is comprised of a user interface generation servlet, a task class that defines job attributes, methods for generating job descriptions, and a set of bootstrap files required for Grid job submission.

In order to simplify the process of implementing user interfaces, LAP provides an object-oriented user interface module, Web.Ui, that renders HTML for web user interfaces and handles form submissions. LAP also provides functionality for automatic generation of xRSL [5] job descriptions.

1.2 The Grid Job Management Framework

Developed in the Grid Infrastructure Research & Development (GIRD) [19] project, the Grid Job Management Framework (GJMF) [6] is a toolkit for job management in Grid environments. The design of the framework is a product of research on service composition techniques [9] and exploration of software design principles for a healthy Grid ecosystem [7]. The framework is implemented as a Service-Oriented Architecture (SOA), using Java and the Globus Toolkit [10].

The GJMF is comprised of a hierarchical set of replaceable Web Services that combined provide an infrastructure for virtualization of Grid middleware functionality and automatization of the repetitive tasks of job management.

The granularity of job management in the GJMF ranges from management of individual jobs to automatic and fault-tolerant processing of sets of abstract task groups. The GJMF provides middleware virtualization by principle of abstraction, and presents a common interface to Grid middleware functionality to developers and end-users without regard to details of the underlying middleware. Functionality in the framework not supported by the underlying middleware, e.g., job state notifications in ARC, is emulated by the framework and presented to applications and end-users as native resources of the middleware. The GJMF also provides numerous structures for customization of the job management process. This customization ranges from individual configuration of the framework services to plug-in structures where, e.g., brokering algorithms, monitoring interfaces, failure handling, and job prioritization modules can be provided and installed by third party developers. See [6] for details.

A full Java client API for the framework is provided and allows developers with limited experience of Web Service development to utilize the framework. This API, as demonstrated in this work, facilitates integration of the GJMF with other systems, e.g., application portals and Grid applications. All features of the framework are accessible through both the Web Service interfaces and the Java client API. The GJMF utilizes JSDL [2] for job descriptions, and provides a translation service for transformations to other job description formats.

2 Integration Architecture

In the proposed architecture, we employ a loosely coupled model where customized modules in portals (the LAP) dynamically discover and access computational resources via a Grid access layer (the GJMF services). The LAP and GJMF are assigned the following responsibilities in the integration architecture.

- Application management: It is the responsibility of the LAP to provide application configuration parameters, gather job submission parameters, create application file repositories, acquire user credentials, authenticate users in the Grid environment, and to render user interfaces. For example, the LAP provides application requirement metadata in job descriptions to indicate and detail the use of Matlab in applications.
- Grid job management: The GJMF is responsible for all matters pertaining to Grid job management. This includes functionality for resource brokering, job submission, job monitoring, job control, and to provide robust handling of failures in job submission and execution. For example, the GJMF uses the previously mentioned application requirement metadata to broker Matlab jobs to Matlab-equipped hosts.

As illustrated in Figure 1, the original 2-tier architecture of the LAP has been extended into a classical 3-tier architecture [4] where the GJMF services are accessed through bridge modules and the GJMF client API. As can be seen in the illustration, the GJMF job management coexists non-intrusively with the legacy ARC/arcLib-based job management modules of the original LAP.

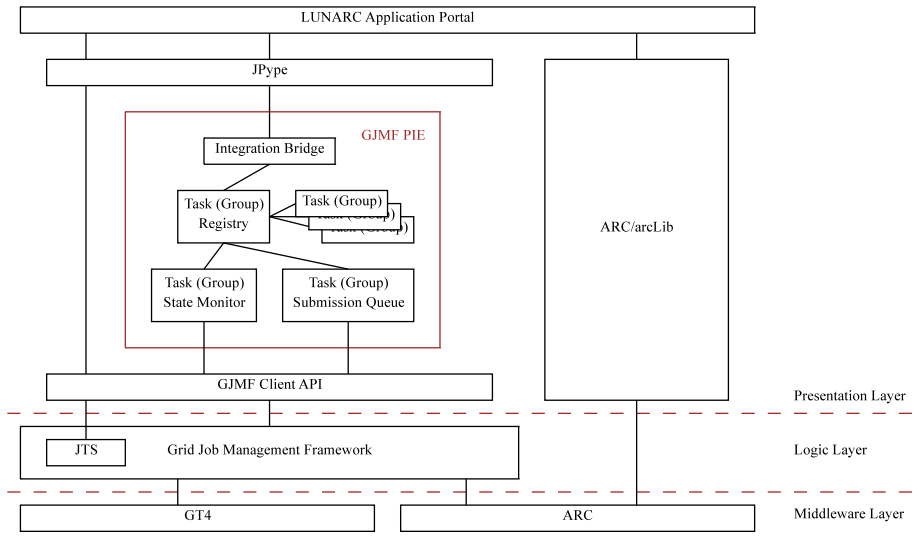


Fig. 1. Overview of the LAP-GJMF integration architecture. Integration components are presented without detailing the internal workings of the LAP or the GJMF.

The integration of the LAP and the GJMF has resulted in the development of the GJMF Portal Integration Extensions (PIE), a customization of the xRSL translation capabilities of the GJMF JSDL Translation Service (JTS), as well as the inclusion of a number of Java-Python bridge components in the LAP.

In the GJMF, it is the purview of the JTS to provide translations between job description formats and to ensure that job semantics are preserved in the process. In the case of the LAP-GJMF integration, there are two types of translations performed: an xRSL to JSDL translation is performed in the LAP upon job creation, and a translation from JSDL to the actual job description format used by the middleware (xRSL for ARC, and RSL [10] for GT4) is performed internally in the GJMF during the final stages of job submission. In the LAP-GJMF integration, the JTS uses job description annotations to provide semantically correct translations of application support parameters (e.g., preserving process environment information) for LAP applications. Naturally, the JTS also contains customization points for extending the translation capabilities to support other formats or alternative translation semantics.

The PIE is a Java-based software component consisting of an integration bridge, a task (group) registry, a submission queue, and a state monitor. These components provide an LAP interface, manage tasks and task groups, handle background GJMF submissions, and monitor GJMF state updates respectively. The PIE effectively wraps use of the GJMF client API and provides functionality for job submission, job control, notification-based state monitoring, and job brokering to the portal. PIE objects are deployed in authenticated sessions in the LAP, run inside the LAP process space, and help enforce user-level isolation

of job information in the portal (each user session is provided a unique PIE instance). In the LAP, bridge modules for job submission, job control, portal status updates, and job monitoring that interface with the PIE have been added. As the bridge modules are native to the LAP (i.e., developed in Python), JPype (version 0.5.3) has been employed to bridge the Java-Python barrier. JPype is a library that allows Python applications to access Java class libraries within the Python process space, connecting the virtual machines on native code level.

As also illustrated in Figure 1, the flexibility of the integration architecture allows existing legacy applications supported by the LAP to continue to function unaltered, including applications who have not yet been adapted to the new environment. This is achieved by the portal maintaining a concurrent legacy job management setup, which utilizes the ARC/arcLib [5] for job management.

When investigating the scalability and flexibility of the architecture, it should be noted that just as a single LAP can make use of multiple GJMFs, multiple LAPs can make use of the same GJMF. Similarly, just as a single GJMF can make use of multiple middleware installations concurrently, so can multiple GJMFs utilize the same middleware installation. Furthermore, as demonstrated by the test configurations of Section 3, the LAP, the GJMF, and the middleware(s) can all be hosted locally or distributed to dedicated servers over networks. The components of the architecture are designed to function non-intrusively [7] for seamless integration in production Grid environments.

3 System Evaluation

To evaluate and demonstrate the flexibility of the proposed architecture, a number of tests have been performed using a range of test configurations and a set of Grid applications that are in current production use.

Software Installations.
The test suites in the system evaluation have been run on nodes deploying different configurations of (at least) three software installations.

- LAP node: A front-end deploying an installation of the upgraded LAP. This node houses the PIE and the bridge components of the integration architecture as well as a fully functional legacy installation of the original LAP architecture. For file staging, the LAP node also deploys a GridFTP server.
- GJMF node: A node deploying a full installation of the GJMF. All GJMF services are run in the same GT4 ws-core 4.0.5 service container to enable inter-service local call optimizations [6, 9], and all communication is protected using GT4 Secure Conversation encryption.
- Middleware node(s): A (set of) middleware back-ends, running either the GT4 or the ARC middleware. The middleware node(s) also deploy middleware-specific GridFTP file staging solutions.

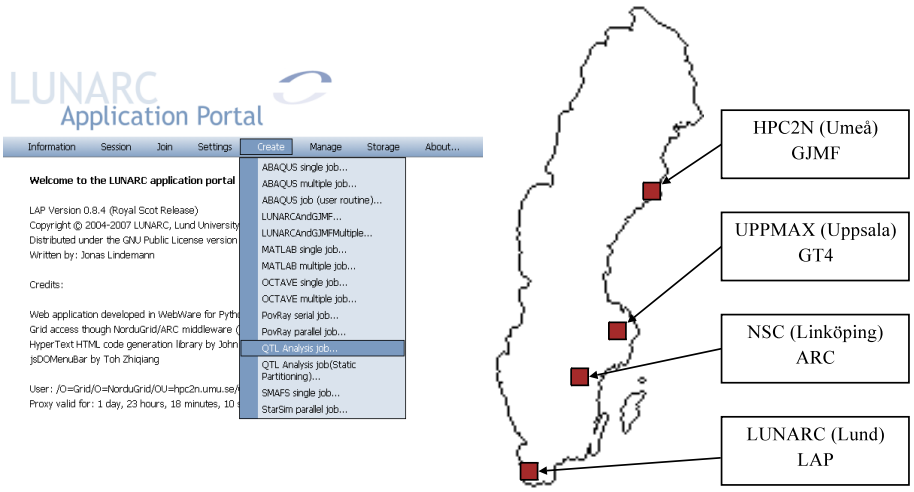


Fig. 2. The LAP and the production environment deployment configuration.

Deployment Configurations.
To illustrate the robustness and flexibility of the proposed architecture, the system is demonstrated in three deployment configurations. NorduGrid certificates are used for authentication of actors and security contexts in all tests.

- Local environment: All three software components are installed on the same machine, and each software component executes in a dedicated process.
- Distributed environment: Each software component is installed on a dedicated machine. All machines are located on the same network.
- Generic production environment: As illustrated in Figure 2, each software component is installed on geographically distributed production machines. The LAP node is located at LUNARC (Lund, Sweden), the GJMF node at HPC2N (Umeå, Sweden), GT4 middleware node(s) at UPPMAX (Uppsala, Sweden), and ARC middleware node(s) at NSC (Linköping, Sweden).

Test Applications.
Two applications for which the LAP is in production use today have been used to gauge the usability of the portal in a production environment.

- Matlab application: The LAP contains a bootstrapping module for initializing and executing Matlab code on Grid resources without use of the Matlab Compiler. This type of application requires a Matlab installation on the computational resource, executes a single job, and performs bidirectional file staging. The Matlab application module is here tested using an implementation of finite element code simulating stresses in straddling beams.
- Bioinformatics application (QTL mapping): This application searches for locations in the genome of an organism affecting a quantitative trait like body weight, crop yield, etc. The search is performed by solving a demanding

multidimensional global optimization problem, which is parallelized into a set of independent jobs. This type of application performs bidirectional file staging but does not require specific execution environment support libraries.

Usage Scenarios.
In the LAP, the main usage scenarios involve two user roles: the application expert and the end-user. Support for new applications is added to the LAP through the creation of application-specific plug-in modules that perform automatic creation of job environments, configuration of application workflows, and generation of application user interfaces. Creation and configuration of applications is the responsibility of the application expert (or system administrator). The process of creating, submitting, and managing jobs is in the LAP performed by the portal end-user, and includes four conceptual stages.

1. The portal end-user creates a job by instantiating a pre-configured application workflow, and supplies the job with required application parameters in the LAP. This results in the generation of an xRSL job description, which is later translated to a JSDL job description using the GJMF JTS.
2. The end-user submits the job from the LAP, an action resulting in the submission of a GJMF task (for single jobs) or a GJMF task group (for multiple jobs) to the PIE. The PIE places the task or task group in a background submission queue, and eventually submits it to the GJMF.
3. The GJMF processes the task or task group, submitting and resubmitting it to middlewares until the process has resulted in a successful job completion. Portal end-users can monitor task or task group progress in the LAP.
4. Once a task or task group has been processed by the GJMF, the end-user accesses resulting data files in the LAP, and removes the job from the LAP. All file stagings are performed by middlewares as GridFTP transfers between the LAP server and the computational resource used for job execution. The GJMF conveys file staging information, but is not actively involved in any file staging scenarios. The file staging semantics of the proposed architecture differ from the original architecture of the LAP, where end-users manually fetched job results from computational resources using HTTP requests. File transfer status is considered part of job execution status and a failed file staging attempt (in either direction) will result in a failed job. A multi-job task failure will not affect the status of other multi-job tasks.

4 Performance Observations

We briefly discuss the proposed architecture's impact on interface response times, job management overhead, and job execution makespans.

– Interface response times. Compared to the original 2-tier architecture of the LAP, the proposed integration architecture improves upon the system's user interface responsiveness, providing instantaneous response to user actions. This is due to the background submission queues of the PIE and the new

architecture's use of the GJMF notification-based state monitoring, which improves scalability through a reduced need for middleware state polling.
- Job management overhead. The overhead associated with job management tasks performed by the GJMF (e.g., resource discovery, task brokering) sum to an average of less than one second per job and has previously been documented in [6]. As the GJMF job management overhead is masked by job execution times, the system-wide impact of this overhead is negligible.
- Job execution makespan. The job execution makespan is made up by factors such as batch system queue times, middleware overhead, job execution time, and file staging times. These factors are independent of the proposed integration architecture and therefor out of the scope of this discussion.

The integration architecture has proven stable and provides enhanced functionality and middleware independence with comparable or improved performance.

5 Related Work

There exist a number of projects that implements web interfaces for Grid resources, such as Gridsphere [13], GridBlocks [11], and the P-GRADE portal [15]. The user interfaces of these portals are often designed as workflow editors and applications are viewed as building blocks in larger contexts. This differs from our work as the LAP focuses on creation of customized user interfaces for specific applications, and provides pre-configured workflows for target applications.

There also exist a number of projects related to the GJMF approach to job management, e.g., the Gridbus [20] middleware that employs a layered architecture and platform-independent approach to Grid job management; the GridWay Metascheduler [14] that provides reliable and autonomous execution of Grid jobs; the GridLab Grid Application Toolkit [1] that provides a service-oriented toolkit for Grid application development; GridSAM [12] that uses JSDL job descriptions and offers middleware-abstracted job submission through Web Services; and P-GRADE [15], which provides fault-tolerant Grid execution of parallel programs.

Related to the integration architecture, a kin project is the GEMLCA [3] integration with P-GRADE [15], where the layered architecture of GEMLCA is employed to run legacy applications as Grid services and P-GRADE provides interfaces for building execution environment, application monitoring, and results management. In comparison with other projects, the aim of the proposed architecture is to illustrate how to exploit the already available components of the LAP and the GJMF using the simplest possible integration model.

6 Conclusions

We have investigated integration techniques for user-friendly, robust, scalable, and flexible Grid portal architectures, proposed a layered approach to system integration, and demonstrated this in the integration of two existing systems; the LAP and the GJMF. The proposed integration architecture improves upon the

original LAP architecture in terms of scalability, support for multiple middlewares, performance, response times, and deployment flexibility. The user-friendly interfaces of the LAP abstract the use of the GJMF, allowing existing portal installations to be transparently upgraded to use the new integration architecture.

Use of the GJMF's automated brokering and job (re)submission capabilities improves the system's fault-tolerance and robustness, and introduces transparent middleware independence. As the multiple job submission mode of the LAP makes use of the GJMF Task Group Management Service (TGMS), the need for manual synchronization of jobs is eliminated and allows end-users to treat multiple jobs as a single management unit. Similarly, use of customized JTS job description translations facilitates middleware independence and automated job result retrieval. The middleware independence introduced by the GJMF allows for integration of new middlewares, facilitates transitions to new environments, and increases the expected lifetime of the LAP and LAP applications. Conversely, use of the LAP's ability to easily create customized application user interfaces empowers the GJMF with application support and usability features.

The proposed integration architecture is lightweight and non-intrusive, supports a representative range of Grid applications, and does not impede use of the original architecture's functionality in any way. In fact, the two versions are completely orthogonal in implementation and can co-exist in the same deployment environment. Use of the GJMF for job management in the portal contributes additional functionality in terms of resource brokering, failure handling, loose coupling of resources, and middleware independence. The PIE improves portal response times and scalability in state monitoring and job submission.

The GJMF-empowered LAP is currently available in a prototype version for SweGrid, supporting bioinformatics, computational chemistry, and astronomy applications. The Matlab extensions of the original architecture have been preserved and Matlab-based applications function unaltered in the new architecture.

7 Acknowledgments

This work has in part been supported by the Swedish Research Council (VR) under contract 621-2005-3667. For use of their resources, we acknowledge HPC2N, Umeå University, LUNARC, Lund University, NSC, Linköping University, and UPPMAX, Uppsala University. We also thank Daniel Henriksson, Johan Tordsson, and the anonymous referees for valuable feedback.

References

1. G. Allen, K. Davis, K. Dolkas, N. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor. Enabling Applications on the Grid - A GridLab Overview. *International Journal of High Performance Computing Applications*, 2003.
2. A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, A. S. McGough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) specification, version 1.0. http://www.ogf.org/documents/GFD.56.pdf, March 2007.

3. T. Delaittre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, and P. Kacsuk. GEMLCA: Running legacy code applications as Grid services. *Journal of Grid Computing*, 3(1 - 2):75 – 90, June 2005. ISSN: 1570-7873.
4. E. Eckerson. Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications. *Open Information Systems*, 10(1):3–22, 1995.
5. M. Ellert, M. Grønager, A. Konstantinov, B. Kónya, J. Lindemann, I. Livenson, J. L. Nielsen, M. Niinimäki, O. Smirnova, and A. Wäänänen. Advanced Resource Connector middleware for lightweight computational Grids. *Future Generation Computer Systems*, 27:219–240, 2007.
6. E. Elmroth, P. Gardfjäll, A. Norberg, J. Tordsson, and P-O. Östberg. Designing general, composable, and middleware-independent Grid infrastructure tools for multi-tiered job management. In T. Priol and M. Vaneschi, editors, *Towards Next Generation Grids*, pages 175–184. Springer-Verlag, 2007.
7. E. Elmroth, F. Hernández, J. Tordsson, and P-O. Östberg. Designing service-based resource management tools for a healthy Grid ecosystem. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics. 7th Int. Conference, PPAM 2007*, volume 4967, pages 259–270. Lecture Notes in Computer Science, Springer-Verlag, 2008.
8. E. Elmroth, M. Nylén, and R. Oscarsson. A User-Centric Cluster and Grid Computing Portal. *International Journal of Computational Science and Engineering*, 3(5), 2007 (to appear).
9. E. Elmroth and P-O. Östberg. Dynamic and Transparent Service Compositions Techniques for Service-Oriented Grid Architectures. In S. Gorlatch, P. Fragopoulou, and T. Priol, editors, *Integrated Research in Grid Computing*, pages 323–334. Crete University Press, 2008.
10. I. Foster. Globus toolkit version 4: Software for service-oriented systems. In H. Jin et al., editors, *IFIP International Conference on Network and Parallel Computing,* LNCS 3779, pages 2–13. Springer-Verlag, 2005.
11. GridBlocks. http://gridblocks.hip.fi, visited December 2008.
12. GridSAM. http://gridsam.sourceforge.net, visited December 2008.
13. Gridsphere Portal Framework. http://www.gridsphere.org/gridsphere/gridsphere, visited December 2008.
14. E. Huedo, R. S. Montero, and I. M. Llorente. A framework for adaptive execution on Grids. *Software - Practice and Experience*, 34(7):631–651, 2004.
15. P. Kacsuk and G. Sipos. Multi-grid and multi-user workflows in the P-GRADE Grid portal. *Journal of Grid Computing*, 3(3-4):221–238, 2006.
16. P. Linde and J. Lindemann. ELT Science Case Evaluation Using An HPC Portal. In *Astronomical Data Analysis Software and Systems XVII*, 2007.
17. J. Lindemann and G. Sandberg. An extendable GRID application portal. In *European Grid Conference (EGC)*. Springer Verlag, 2005.
18. J. Lindemann and G. Sandberg. A Lightweight Application Portal for the Grid. In *Nordic Seminar on Computational Mechanics NSCM 19*, 2006.
19. The Grid Infrastructure Research & Development (GIRD) project. Umeå University, Sweden. http://www.gird.se, visited December 2008.
20. S. Venugopal, R. Buyya, and L. Winton. A Grid service broker for scheduling e-science applications on global data Grids. *Concurrency and Computation: Practice & Experience*, 18(6):685–699, May 2006.
21. Webware, Python Web Application Toolkit. http://www.webwareforpython.org, visited December 2008.

V

Paper V

A Composable Service-Oriented Architecture for Middleware-Independent and Interoperable Grid Job Management

Erik Elmroth and Per-Olov Östberg

Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden
{elmroth, p-o}@cs.umu.se
http://www.gird.se

Abstract: We propose a composable, loosely coupled Service-Oriented Architecture for middleware-independent Grid job management. The architecture is designed for use in federated Grid environments and aims to decouple Grid applications from Grid middlewares and other infrastructure components. The notion of an ecosystem of Grid infrastructure components is extended, and Grid job management software design is discussed in this context. Nonintrusive integration models and abstraction of Grid middleware functionality through hierarchical aggregation of autonomous Grid job management services are emphasized, and service composition techniques facilitating this process are explored. Earlier efforts in Service-Oriented Architecture design are extended upon, and implications of these are discussed throughout the paper. A proof-of-concept implementation of the proposed architecture is presented along with a technical evaluation of the performance of the prototype, and a details of architecture implementation are discussed along with trade-offs introduced by the service composition techniques used.

Key words: Grid job management, service composition, federated Grids, middleware-independence, Grid ecosystem

A Composable Service-Oriented Architecture for Middleware-Independent and Interoperable Grid Job Management

Erik Elmroth and Per-Olov Östberg

Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden

Abstract

We propose a composable, loosely coupled Service-Oriented Architecture for middleware-independent Grid job management. The architecture is designed for use in federated Grid environments and aims to decouple Grid applications from Grid middlewares and other infrastructure components. The notion of an ecosystem of Grid infrastructure components is extended, and Grid job management software design is discussed in this context. Non-intrusive integration models and abstraction of Grid middleware functionality through hierarchical aggregation of autonomous Grid job management services are emphasized, and service composition techniques facilitating this process are explored. Earlier efforts in Service-Oriented Architecture design are extended upon, and implications of these are discussed throughout the paper. A proof-of-concept implementation of the proposed architecture is presented along with a technical evaluation of the performance of the prototype, and a details of architecture implementation are discussed along with trade-offs introduced by the service composition techniques used.

Key words: Grid job management, service composition, federated Grids, middleware-independence, Grid ecosystem

Email address: {elmroth, p-o}@cs.umu.se [http://www.gird.se] (Erik Elmroth and Per-Olov Östberg)

1. Introduction

The increasingly common use of federated Grids put new requirements on software for job and resource management. Common examples of federated Grids include hierarchical Grids, where large-scale international collaborations make use of parts of national Grids, and multiple national Grids allow cross-Grid utilization in order to more efficiently handle variations in resource demand. Requirements placed on software for federated Grids even further emphasize many of the key requirements typically put on software for individual Grids. As federated Grids are often more short-lived, less monolithic, and more heterogeneous in Grid middleware, there is an even stronger need for tools to provide key functionality with great flexibility in deployment and configuration. The need for software that coexist and non-intrusively integrate with other middleware components is vital, scalability requirements even more emphasized as system size increases, and centralized solutions even less feasible due to factors such as increased heterogeneity in Grid access, and policy enforcement being performed locally on resource sites.

In these settings, Grid job management tools should focus on maintaining non-intrusive integration models, provide functionality on top of available Grid interfaces, and abstract complexity of underlying Grid infrastructure components when possible. Resource brokering should be performed assuming the use of multiple concurrent job submission systems and without attempts of maintaining global state information for Grid jobs or resources. Resource contention issues and failure handling should be implemented using adaptive approaches and robustness provided through redundancy of capability rather than prediction of possible failure causes.

While there are many viable approaches to Grid job management in use today, there exists a need for robust Grid job management tools that are able to function across Grid boundaries, integrate non-intrusively, and provide abstractions of Grid middleware functionality that decouple applications from specific Grid middlewares. Generic Grid applications decoupled from Grid middlewares are more likely to be able to migrate to new Grids, be reused in new projects, and adapted to new problems. To further a looser coupling between applications and Grid resources, tools need to provide flexibility in utilization and deployment without sacrificing scalability, performance, or middleware and platform independence.

The research question of how to best design Grid infrastructure software is currently open-ended and addressed in a number of different ways. The

approach taken in this work consists of identification of a set of desirable traits likely to promote software sustainability in a Grid ecosystem (as described in Section 2), and exploration of software design and development methodologies which result in composable software components that inhabit and define niches in an ecosystem of Grid infrastructure components.

For software aimed for collaborative Grid environments, usage scenarios tend to include a number of complex factors such as heterogeneous user bases organized in virtual organizations, varying deployment requirements, resource heterogeneity and contention issues, unpredictable failure models, and ever-changing user requirements. In such settings, robustness of tools are prioritized, and utility is measured in terms of scalability, flexibility in usage and deployment, level of functionality and heterogeneity abstraction, complexity of administration, ability to automate repetitive administrative tasks, degree of coupling between components, and level of integration intrusion.

In this work, we extend the software design methodologies of [22], [20], and [18], and propose a composable Service-Oriented Architecture (SOA) for Grid job management constituted by layers of loosely coupled, composable, and replaceable Web Services. The architectural model of the framework is built upon the principle of abstraction; functionality is stratified into layers of autonomous services that incrementally provide additional features to end-users by utilizing and abstracting complexity of underlying services. This enables software developers to build aggregated systems where individual parts of the architecture can be deployed as stand-alone components, minimizes the formal knowledge between components to provide a loosely coupled model of component interaction, and allows great flexibility in system configuration. Application developers can choose what parts of the framework to make use of based on current application needs, and system administrators can reconfigure framework deployment dynamically. The architecture also promotes application and middleware interoperability by providing an abstracted interface to Grid middleware functionality, and supports concurrent use of multiple middlewares. The services of the framework implement well defined interfaces and provide customization points for dynamic alteration of component and system behavior. By use of configurable customization points in the middleware abstraction services, additional middleware support can easily be provided by third parties.

The architecture is illustrated by a proof-of-concept implementation called the *Grid Job Management Framework (GJMF)*, which is presented along with an evaluation of system performance. The framework is designed to provide

transparent Grid access to applications through a set of abstractive interfaces that can be combined with (optional) advanced customizability features. Applications built on top of the framework are provided transparent Grid middleware functionality that allows Grid resource utilization without coupling applications to specific middlewares. Applications and infrastructure tools are also able to reuse components of the framework for generic Grid operation on an individual basis, promoting a functionality-based model of software reuse and increasing component and system sustainability. Throughout the paper, intended system behavior and implications of system design and architecture are discussed alongside documentation of experiences from system design and development.

The structure of the remainder of the paper is as follows: In Section 2 an introduction to the concept of an ecosystem of Grid components is given along with a brief description of software requirements for infrastructure components inhabiting such a system. After this, an architecture model for a layered Grid job management framework is proposed in Section 3, followed by a detailed presentation of the individual services of the proof-of-concept implementation of the framework in Section 4. An architecture discussion then ensues in Section 5, followed by a performance evaluation illustrating some of the framework trade-offs in Section 6. Related and future work are presented in sections 7 and 8, respectively, and the paper is concluded in Section 9.

2. The Grid Ecosystem and Software Requirements

An ecosystem can be defined as a system formed by the interaction of a community of organisms with their shared environment. Central to the ecosystem concept is that organisms interact with all elements in their surroundings, and that ecosystem niches are formed from specialization of interactions within the ecosystem. In an ecosystem of Grid components [64], [20], niches are defined by functionality required and provided by software components, end-users, and other Grid actors; and Grid infrastructures are constituted by systems composed of components selected from the ecosystem. Here, software compete on evolutionary bases for ecosystem niches, where natural selection tend to preserve components better at adapting to altered conditions over time. Adaptability is hence defined in terms of integrability, interoperability, adoptability, efficiency, and flexibility. For software to be successful in the Grid ecosystem, individual software components should be

composable, replaceable, able to integrate non-intrusively with other components, support established niche actors, e.g., Grid middlewares and applications, and promote adoptability through ease of use and minimization of administrational complexity.

Currently however, the majority of Grid resources available are accessible only through a specific Grid middleware deployed on the site of the resource. This, combined with the complexity and interoperability issues of today's Grid middlewares, leads to the Grid interoperability contradiction [24], and tend to result in a degree of tight coupling between Grid applications and Grid middlewares. To isolate Grid end-users and applications from details of the underlying middleware and create a more loosely coupled model of Grid resource use, a Grid job management tool should be designed to operate on top of middlewares, abstract middleware functionality and offer a middleware-agnostic interface to Grid job management. From an ecosystem point-of-view, this type of Grid middleware functionality abstraction helps to define and decouple an autonomous job management niche.

Furthermore, to promote interchangeability, components should build upon standardization efforts, e.g., support de facto standard approaches for virtual organization-based authentication and accounting solutions, function independent of platform, language, and middleware requirements, and provide transparent and easy-to-use Grid resource access models that support use of federated Grid resources. This reduces ecosystem component development complexity, mitigates learning requirements for Grid end-users, and promotes interoperability and adoption of Grid utilization in new user groups.

Like in any evolution-based system, adaptability and efficiency are key to software sustainability in the Grid ecosystem as they promote adoption and use of software components. By creating systems composed of small, well-defined, and replaceable components, functionality can be aggregated into flexible applications, resulting in increased survivability for both components and applications [22]. This idea to create composed and loosely coupled applications from autonomous networked components lies at the heart of Service-Oriented Architecture (SOA) [53] methodology.

In the framework, components are realized as autonomous Web Services that contain multiple customization points where third party plug-ins can be used to alter or augment both system and component behavior. To promote deployment flexibility, the framework composition, as well as individual component customization setup, can also be dynamically altered via service configurations as described in [22]. Additionally, individual components of

the framework can be used as stand-alone services, or in other composed architectures, while concurrently serving as part of the framework in the same deployment environment.

3. Framework Architecture

The practice of developing and deploying infrastructure components as dynamically configured SOAs facilitates development of flexible and robust applications that aggregate component functionality and are capable of dynamic reconfiguration [22]. This approach also provides a model for distributed software reuse, both on component and code level, and facilitates integration software development with a minimum of intrusion into existing systems [21]. Providing small, single-purpose components reduces component complexity and facilitates adaptation to standardization efforts [22].

The architectural model used has previously been briefly introduced in [18], and various aspects of the software development model have been discussed in [22], [20], and [21]. The software development model used in this work is a product of work in the Grid Infrastructure Research & Development (GIRD) multiproject [65] and is documented in [22] and [20]. The models favor architectures built on principles of flexibility, robustness, and adaptability; and aims to produce software well adjusted for use in the Grid ecosystem [64].

3.1. Architecture Layers

As illustrated in Figure 1, the framework architecture is divided into six layers of functionality, where each layer builds upon one or more lower layers and provides aggregated functionality to service clients. The layers range (bottom-up) from a Grid middleware layer to an application layer with four job management layers in between. For each layer a core functionality set has been identified and implemented as autonomous services in the proof-of-concept prototype (illustrated in the figure).

Grid Middleware Layer. In the architecture model, the Grid middleware layer houses all software components concerned with abstraction of native job management capabilities. This typically constitutes traditional Grid middlewares abstracting batch systems, e.g., the Globus middleware (GT4) [35] abstracting the Portable Batch System (PBS) [42], standardized job dispatchment services, e.g., the OGSA BES [28], and desktop Grid approaches such as

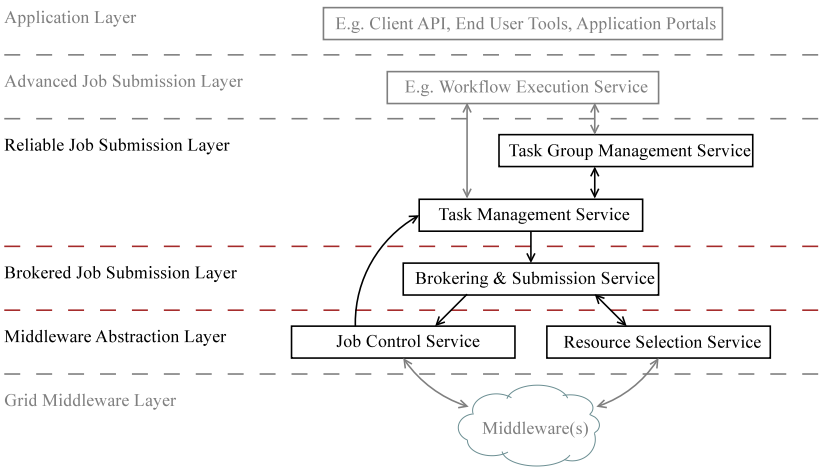


Figure 1: The proposed framework architecture. Services organized in hierarchical layers of functionality. Within the framework services communicate hierarchically, service clients are not restricted to this invocation pattern.

the Berkeley Open Infrastructure for Network Computing (BOINC) [7] and Condor [62] abstracting use of CPU cycle scavenging and volunteer computing resources. Components in the Grid middleware layer are not considered part of, or provided by, the framework but are essential in providing native job submission, control, and monitoring capabilities to the framework.

Middleware Abstraction Layer. The purpose of the middleware abstraction layer is to abstract the details of Grid middleware components and provide a unified Grid middleware interface to higher-layer components. All framework components housed in other layers are insulated from details of native and Grid job submission, monitoring, and control by the services in the middleware abstraction layer. Hence, integration of the framework with additional (or new versions of) Grid middlewares should ideally only concern components in this layer.

Currently, the middleware abstraction layer contains services for targeted job submission and control, information system interfaces, and services concerned with translation of job descriptions. Components in the middleware abstraction layer are expected to abstract middleware complexity and provide well-defined interfaces and customization support for integration of new

Grid middlewares. For middlewares lacking required functionality, e.g., middlewares with limited job monitoring capabilities, components in the middleware abstraction layer are expected to implement required system functionality to provide a unified job control interface.

Brokered Job Submission Layer. Placed atop of the middleware abstraction layer, the brokered job submission layer provides aggregated functionality for indirect, or brokered, job submission. The services of this layer improves upon the targeted job submission capabilities of the middleware abstraction layer by providing automated matching of jobs to computational resources. Job submission performed by services in this layer relies on the targeted job submission capabilities and the information system interfaces of the middleware abstraction layer, and provides a best effort type of failure handling by identifying a set of suitable computational resources for a job and (sequentially) attempting to submit the job to each of these until the job is accepted by a resource. Services in the brokered job submission layer do not provide job monitoring capabilities, as job submission here is expected to result in monitorable jobs in middleware abstraction layer services.

Reliable Job Submission Layer. Intended as the robust job submission abstraction of the architecture, services of the reliable job submission layer provide fault-tolerant and autonomous job submission and management capabilities. The term reliable job submission refers to the ability of these services to autonomously handle different types of errors in the job submission and execution processes through resubmission of jobs according to predefined failover policies. Services in the reliable job submission layer rely on services of the brokered job submission layer for brokering and job submission, and services of the middleware abstraction layer for job monitoring and control. Functionality for failure handling, e.g., for Grid congestion and job execution failures, is aggregated, and management of sets of independent jobs is provided. Services of the reliable job submission layer also provide monitoring capabilities for jobs and sets of jobs through job management contexts created for all resources submitted here.

Advanced Job Submission Layer. The advanced job submission layer is in the architecture of the framework aimed towards more advanced mechanisms for job management, e.g., workflow tools, Grid application components, and portal interfaces that by functionality requirements are coupled to individual components of the framework. The services of the advanced job submission

layer are intended to utilize the services of the reliable job submission layer, and function as integration bridges and customized service interfaces to the framework. Services in the advanced job submission layer are expected to provide their own job management and monitoring contexts as they are intended to aggregate the functionality of the other layers of the framework. A number of functionality sets for advanced job management have been identified and are under consideration (see Section 8) for development in the prototype implementation of the framework, e.g., management of data and sets of interdependent jobs.

Application Layer. Residing at the top of the hierarchical structure of the framework, the application layer houses Grid applications, computational portals, and other types of external service clients. As in the case of the Grid middleware Layer, softwares in the application layer are not necessarily considered part of the architecture of the framework, but are likely to impact the design of software in the architecture through design, construction, and feature requirements. Typically, service clients not integrated with the framework services are considered part of the application layer.

4. The Grid Job Management Framework (GJMF)

Implemented as a prototype of the proposed architecture of Section 3, the Grid Job Management Framework (GJMF) is a Java-based toolkit for submission, monitoring, and control of Grid jobs designed as a hierarchical SOA of cooperating Web Services. Framework composition can be altered dynamically and controlled through service configuration and via customization points in services. The Grid-enabled Web Services of the GJMF have been implemented and are typically deployed using the Globus Toolkit [26], are compatible with established Grid security models, and conform to the use of a number of Web Service and Grid standards, e.g., the Web Service Description Language (WSDL) [14], SOAP [40], the Web Service Resource Framework (WSRF) [27], and the Job Submission Description Language (JSDL) [9]. The GJMF also conforms to the design of the Open Grid Service Architecture (OGSA) [29] and builds on the design of the OGSA Basic Execution Service (OGSA BES) [28], and the OGSA Resource Selection Services (OGSA RSS) [31].

The services in the framework interact by passing messages using either request-response (for, e.g., job submissions) or publish-subscribe (for, e.g.,

state update notifications) communication patterns. The information routed through the framework travels vertically in Figure 1, and typically consists of job descriptions passed downwards in task and job submissions, and status update notifications propagated upwards in service state coordination messages. All services maintain state representations as WS-Resources [38], and expose these through service interfaces and WS-ResourceProperties [37], allowing clients to inspect state both explicitly and through subscription to WS-BaseNotifications [36].

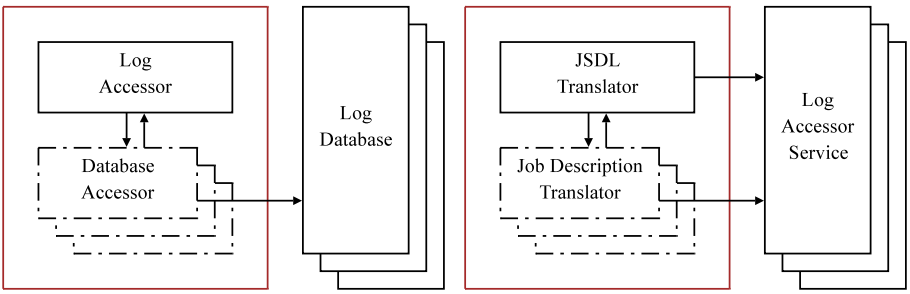
4.1. Job Definitions

To facilitate the model of offering aggregated functionality through services organized in hierarchical layers, the GJMF defines three types of job definitions.

- A *job* is a concrete job description, containing all information required to execute a program on a (specified) computational resource. Jobs are in the GJMF processed by the Job Control Service and correspond to unique executions of programs on computational resources. Jobs typically consist of a JSDL file specifying an executable program, program parameters, computational resource references, file staging information, and optional JSDL annotations containing custom job processing hints.

- A *task* is an abstract (often incomplete) job description that typically requires additional information, e.g., computational resource references or specific job submission parameters, to become submitable to Grid middlewares. This required information is typically provided by task to resource matching (brokering). Tasks are in the GJMF processed by the Task Management Service. Note that by the GJMF definition, a job is a task subtype. This allows jobs to be submitted as tasks in the GJMF, in which case the additional brokering information is utilized in the brokering and job submission process.

- A *task group* is a set of independent tasks and jobs that can be executed in any order. Task groups distinguish themselves from jobs and tasks by having shared execution contexts for all tasks in a task group. Thus, the processing result of a task group is determined by the combined processing results of the task group's tasks and jobs. Task groups are in the GJMF processed by the Task Group Management Service.



(a) The Log Accessor Service. (b) The JSDL Translation Service.

Figure 2: Internal structure of the Log Accessor Service and the JSDL Translation Service. Customization points are illustrated using dotted lines. The services are constructed around a set of customizable core components that provide database access and job description translation semantics respectively.

4.2. Components

As illustrated in Figure 1, the core of the GJMF is made up by five job management services. Part of the framework but not illustrated in the figure are also two auxiliary services, a job description translation and a log access service, as well as two core libraries, a service development utility library and the GJMF client Application Programming Interface (API). All services in the GJMF make use of these libraries, and all service interaction within the framework is routed through the service client APIs, allowing service communication optimizations to be ubiquitous and completely transparent to services, service clients, and end-users. Each service is capable of using multiple instances of other services, and supports a model of user-level isolation where unique service instances (back-ends) are created for each service user. Worker threads and contexts within individual services are shared among service back-ends and competition for resources between service instances occur as if services were deployed in separate service containers.

4.2.1. Log Accessor Service (LAS)

In a distributed architecture managing multiple synchronized states, ability to track state development and review processing progression is highly desirable. The Log Accessor Service (LAS) is a service that provides database-like interfaces to job, task, and task group logs generated by the GJMF. As

the name suggests, the LAS is designed to provide convenient log access to services, end-users, and clients. Within the GJMF, the LAS is used to record state transitions as well as job submission and processing information. The LAS is typically expected to have monodirectional data transfers, e.g., the GJMF services use the LAS to store data, and service clients use it to inspect details of task processing.

The internal structure of the LAS is illustrated in Figure 2a. The log accessor component offers a service interface and abstracts use of database-specific accessor plug-ins. The LAS maintains internal storage queues and resource serialization mechanisms to minimize overhead for use of the service and provide an asynchronized communication model for log storage. As also illustrated in Figure 2a, database support is provided the service through the use of customizable database accessor plug-in modules. These accessors can be provided by third parties to provide the LAS access to custom database formats currently not supported. Boiler-plate solutions for accessor plug-ins supporting Structured Query Language (SQL) [43] and Java Database Connectivity (JDBC) [61] are provided to facilitate development of custom plug-ins. Currently, the LAS supports use of MySQL [52], PostgreSQL [56], and Apache Derby [63], and accessor plug-ins for these systems are provided. Unlike the other services of the GJMF, use of the LAS is optional and not required for any other part of the GJMF to function. The LAS can be configured to use specific database accessors, and these accessors can also be configured through the LAS configuration.

4.2.2. JSDL Translation Service (JTS)

In the GJMF, the JSDL Translation Service (JTS) is used to provide job description translations to service clients and services. In terms of service to service communication, the JTS is typically used by the Job Control Service to provide translations of JSDL to formats used by Grid middlewares in native job submission. When used by service clients, the JTS can both provide translations from proprietary job description formats to JSDL, and translations from JSDL to Grid middleware formats (where the latter typically would be used to verify that job description semantics are preserved in translation).

As illustrated in Figure 2b, the JTS employs a modularized architecture where translation semantics are provided by plug-in modules, and support for new language translations can be added by third parties without modification of the framework. The JSDL translator component provides a service

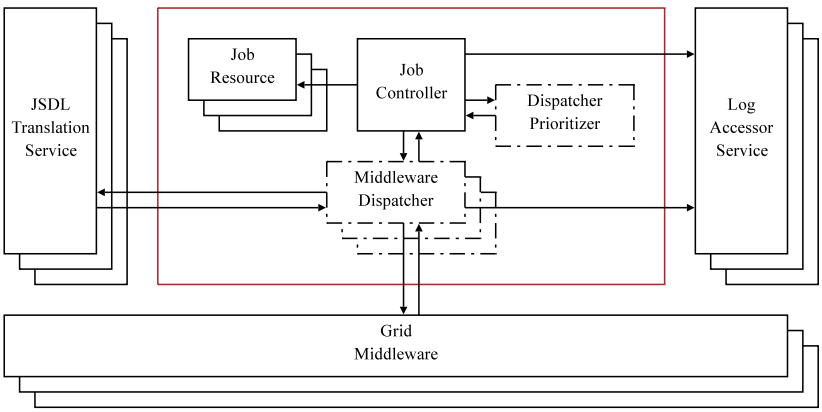


Figure 3: Internal structure of the Job Control Service. Customization points are illustrated using dotted lines.

interface and abstracts the use of job description translator plug-ins. Both the JSDL translator and the translator plug-ins make optional use of LASs for log storage. Currently, the JTS supports translation between JSDL [9] and Globus Toolkit 4 Resource Specification Language (GT4 RSL) [26], NorduGrid Extended Resource Specification Language (XRSL) [17], and a custom dialect of XRSL presented in [21]. Translations of job descriptions are made based on the context of the job description representation created. Typically this means that job descriptions to be translated are parsed record by record for information required to create new representations of corresponding semantics. Type-specific data representations are translated based on the semantics of the enacting middleware, e.g., Uniform Resource Locators (URLs) are reformatted and supplied suitable protocol tags to match middleware transfer mechanism preferences. The JTS can be configured to use a specific set of translation modules, which can be configured through the JTS configuration.

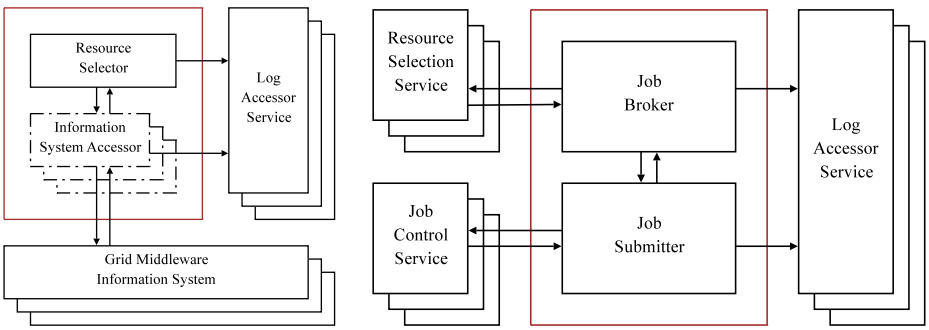
4.2.3. Job Control Service (JCS)

Being one of the two fundamental middleware abstraction services of the GJMF, the purpose of the Job Control Service (JCS) is to provide a uniform and middleware-independent interface for job submission and control. The JCS defines a set of generic job functionality, as well as a job state model (illustrated in Figure 8c), that provide a fundamental view of job management

that other services in the GJMF build upon. Within the GJMF, the JCS is used by the Brokering & Submission Service as a job submission interface, and by the Task Management Service as a job monitoring and control interface, but the service may also be used directly by service clients as a targeted Grid job submission and control tool.

The internal structure of the JCS is illustrated in Figure 3. The job controller component provides a service interface and coordinates execution of jobs. Job resources are used to maintain job state and are exposed as inspectable WS-ResourceProperties to service clients. The job controller abstracts the use of middleware-specific job dispatcher and dispatcher prioritizer plug-ins, and both the job controller and the middleware dispatchers utilize LASs for log storage. Middleware support in the JCS is provided through customizable and configurable plug-in modules that allow third parties to develop and deploy support for proprietary job management solutions. Middleware dispatchers abstract use of Grid middlewares and employ the JTS and the LAS for job description translation and log storage respectively. The JCS currently provides middleware support for the NorduGrid ARC [17], GT4 [35] middlewares, and Condor [62]. For test and service client development purposes, the JCS also provides a simulation environment where jobs are simulated rather than submitted and executed. This utility allows JCS clients to encounter exotic job behaviors on demand via discrete-event simulation of job state transitions.

The JCS can be configured to use a specific set of middleware dispatchers, a middleware dispatcher prioritizer, a state monitor, a set of JTSs, and an optional set of LASs. For custom job processing, the functionality of the JCS may also be altered by providing processing hints to the JCS through annotations in the JSDL job description. These annotations can affect, e.g., middleware dispatcher prioritization, or provide job submission parameters such as queue system information for ARC submissions (an example from [21]) or GT4 Globus Resource Allocation Manager (WS-GRAM) parameters for Condor-G [32] submissions. As these types of processing hints are completely orthogonal to standard service behavior, i.e. does not affect processing of other jobs or other service functionality, they can be used to temporarily alter service behavior for a specific job without alteration of framework composition or configuration.



(a) The Resource Selection Service.

(b) The Brokering and Submission Service.

Figure 4: Internal structures of the Resource Selection Service and the Brokering and Submission Service. Customization points are illustrated using dotted lines.

4.2.4. Resource Selection Service (RSS)

The fundamental task of matching a job to a suitable computational resource on a Grid is referred to as job or resource brokering. Built on the OGSA RSS [31] model, the GJMF Resource Selection Service (RSS) provides a service interface for performing job to resource matching in Grid environments. Within the GJMF, the RSS is used by the Brokering & Submission Service as an execution planning and brokering tool, but the service may also be used by service clients for job to resource matching directly.

The internal structure of the RSS is illustrated in Figure 4a. The resource selector component provides a service interface, coordinates brokering of tasks to computational resources, abstracts the use of middleware-specific information system accessor plug-ins, and utilizes LASs for log storage. Information system accessors abstract the use of middleware information systems, provide translations of middleware-specific record formats to an internal RSS format, and make use of LASs for log storage. The RSS internally maintains mechanisms for retrieval of resource information from information systems, caching of resource information, information system monitoring, and a customization mechanism that allows third parties to develop plug-ins to support new information sources, e.g., new Grid middleware information systems.

The RSS can be configured to retrieve information from a range of information systems, currently including the ARC and GT4 Grid middleware

information systems, as well as a simulated information system configurable through the RSS configuration intended for service development purposes. The RSS also provides boiler-plate solutions for data access and type conversion to facilitate implementation of custom information accessors.

4.2.5. Brokering & Submission Service (BSS)

The Brokering & Submission Service (BSS) provides the GJMF and service clients with an interface for best-effort brokered job submission. The definition of best effort job submission used here is that no measures for correction of, or compensation for, failed job submissions or executions are taken. Once brokered, the BSS attempts to sequentially submit jobs to each suitable computational resource identified (as ranked by the RSS) until a resource has accepted the job or the list of resources has been exhausted. Beyond this behavior, failures are considered permanent.

Within the GJMF, the BSS is used by the Task Management Service for task submissions, but the BSS may also be used directly by service clients as a best effort job submission tool for brokered submission of abstract (incomplete) job descriptions. The BSS does not maintain a context for submitted jobs, service clients that wish to inspect job state are referred to a JCS instance hosting the job upon successful job submission. Note that while job submission failures are reported directly to service clients, errors in job executions are by the BSS assumed to be reported by the enacting JCS or detected and handled by service clients.

The internal structure of the BSS is illustrated in Figure 4b. The job broker component provides a service interface and interacts with RSSs to retrieve execution plans for tasks. The job submitter component is used by the job broker and interfaces with JCSs to submit jobs. Both components make use of LASs for log storage. The BSS relies on the RSS and JCS for job to resource matching and job control respectively, and is capable of using multiple instances of each service to provide redundancy in job brokering and submission. Note that jobs, i.e., tasks with a concrete job description including a resource specification, are not relayed to the RSS for resource brokering but directly submitted to resources via the JCS. The BSS can be configured to use a set of RSSs, a set of JCSs, and an optional set of LASs.

4.2.6. Task Management Service (TMS)

Being the primary mechanism for reliable submission of individual jobs in the GJMF, the Task Management Service (TMS) provides an interface for au-

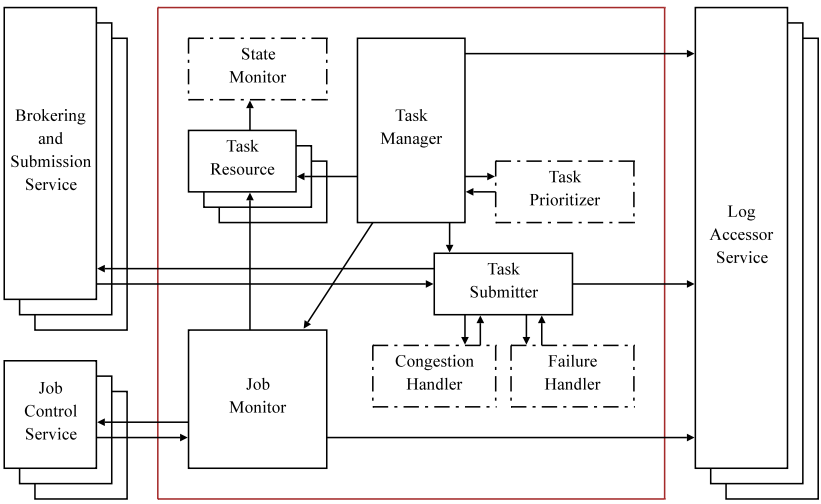


Figure 5: Internal structure of the Task Management Service. Customization points are illustrated using dotted lines.

tomated and fault-tolerant task management and defines a task state model (illustrated in Figure 8b). The TMS maintains inspectable state contexts for tasks and employs a model of event-driven state management powered by the JCS state mechanisms. To provide failover capabilities, tasks submitted through the TMS are repeatedly submitted and monitored by the TMS until resulting in a successful job execution, or a configurable amount of attempts have been made (in which case the task fails). Within the GJMF, the TMS is used by the Task Group Management Service for management of individual tasks.

The internal structure of the TMS is illustrated in Figure 5. The task manager provides a service interface, coordinates task processing using a task prioritizer plug-in, and uses LASs for log storage. The task submitter utilizes BSSs for task submission, employs congestion and failure handler plug-ins for task resubmission decision support, and stores state through LASs. Task state is maintained and exposed through WS-ResourceProperties by task resources. A state monitor plug-in can be employed to provide customizable access to task state. The job monitor utilizes JCSs for job monitoring and control of jobs, updates task resources, and stores state through LASs.

The internal mechanisms of the TMS can be customized via configuration

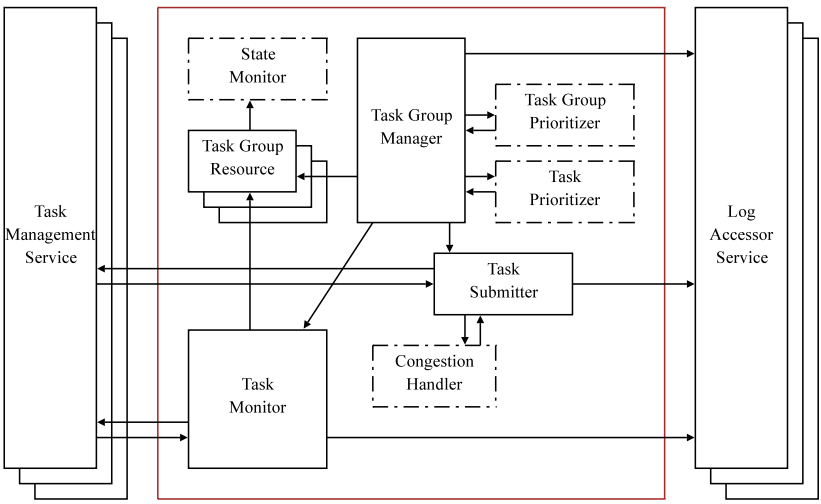


Figure 6: Internal structure of the Task Group Management Service. Customization points are illustrated using dotted lines.

and a set of plug-in modules that control task prioritization, congestion handling, failure handling, and state monitoring. To enforce user-level isolation and fair competition in multi-user scenarios, the TMS maintains separate job queues for each user. The TMS relies on the BSS for submission of tasks to Grid resources, and can be configured to use customized congestion and failure handlers to control task resubmission behaviors, and a customized task prioritizer to influence task processing order. The TMS can also be configured to use a state transition monitor for event-driven state monitoring, a set of BSSs, and an optional set of LASs.

4.2.7. Task Group Management Service (TGMS)

Similar to the TMS for individual jobs, the Task Group Management Service (TGMS) exposes an interface for management of groups of (mutually independent) jobs and tasks, and defines a task group state model (illustrated in Figure 8a). The TGMS provides a convenient way to manage sets of tasks as a single entity, and is intended to be used by service clients and more complex task management systems, e.g., workflow engines such as [19] or parameter sweep applications. The TGMS is currently not used by other services in the GJMF.

The internal structure of the TGMS is illustrated in Figure 6. The task group manager provides a service interface, coordinates task and task group processing using task and task group prioritizer plug-ins, and stores state through LASs. Task group state is maintained and exposed as WS-ResourceProperties by task group resources, which can also be accessed by state monitor plug-ins. The task submitter submits jobs to TMSs, uses a congestion handler plug-in for resubmission decision support, and stores logs through LASs. The task monitor utilizes TMSs for task monitoring, updates task group resources, and stores logs through LASs.

The TGMS maintains state contexts for task groups, employs user-exclusive submission queues for both task groups and tasks, provides customizable plug-in modules for task group and task prioritization, state management, and congestion handling. As the TGMS relies on the TMS for task submission and management, the TGMS does not contain a failure handler for job submission or execution failures. Task execution failures in the TMS are by the TGMS considered permanent, no error recovery or failover actions are taken by the TGMS. Task submission failures, i.e. failures in TMS task submission, are considered temporary and result in the TGMS rescheduling task submissions indefinitely until successful.

The TGMS also provides a mechanism for suspension of (processing of) task groups, a mechanism designed to adapt to scenarios where user credentials expire or large task groups need to be paused. Once suspended, task groups need to be explicitly resumed to be processed by the TGMS. Tasks in a suspended task group that have already been submitted to a TMS will be processed if possible, but no new task submissions will be made until (processing of) the task group has been resumed. The TGMS can be configured to use a congestion handler to customize back-off behaviors in Grid congestion situations; task group and task prioritizers to customize processing order of task groups, tasks, and jobs; a state transition monitor for event-driven state monitoring, a set of TMSs, and an optional set of LASs.

4.2.8. The GJMF Common Library

The GJMF common library is a service development utility library that encapsulates functionality common to all services of the GJMF. The library facilitates service development by providing a common type set, a service development model, and boiler-plate solutions for, e.g., local call optimizations, service stubs, credentials delegation, security contexts, worker threads, state management, service client APIs, dynamic configuration, and resource

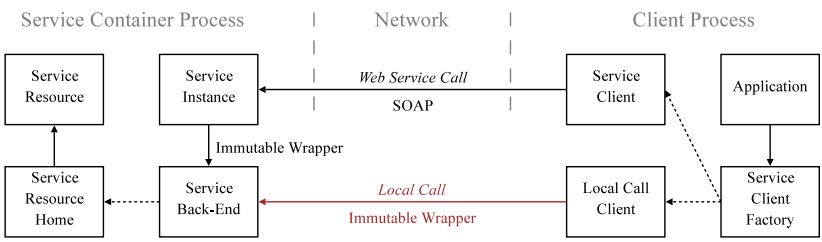


Figure 7: The GJMF service structure. The GJMF common library provides boiler-plate solutions for service instantiation, service back-end implementation, resource management, and client APIs. The GJMF client API abstracts use of the service invocation optimizations through use of service client factories. Dynamic invocation patterns illustrated using dotted lines.

serialization.

The GJMF common library provides a simple framework for service development that defines a service structure used by all services in the GJMF. The service structure is illustrated in Figure 7, and details separation of service interface implementation from service back-end implementations, and service clients from service client factories. Service client factories are exposed to applications and dynamically instantiate service client implementations based on type of service invocation to be used. Service clients marshal data and perform service invocations, in the case of regular service clients through Web Service SOAP messages and through direct service back-end invocations using immutable wrapper types for local call optimization clients. Service interface implementations marshal SOAP data through stubs into immutable wrapper types and invoke corresponding methods in service back-ends. Service back-end implementations are responsible for maintaining state in service resources, which are accessed through service resource homes. The service structure of the GJMF common library has previously been discussed in [22].

The service development framework, in concert with the GJMF client API, handles common tasks such as data type marshalling, service instantiation, notification subscription management, and notification delivery. The framework also encapsulates a local call optimization mechanism that allows service components to be exposed as local objects to other services codeployed in the same service container, which allows co-hosted services to make

marshalled in-process Java calls directly between service clients and service back-ends. This optimization mechanism, which is discussed in Section 5.1, evaluated in Section 6.3.5, and also addressed in [22], is hidden by the service structure of the common library and made completely transparent to service clients through the client API. As described in [22], the common library provides a set of basic and immutable types for use in the GJMF client API as well as a type marshalling mechanism that abstracts the use of stub types in the GJMF.

The primary purpose of the GJMF common library is to facilitate service development by providing standardized solutions to common tasks in service development. While end-users and GJMF service clients typically never interact directly with the common library, most of the functionality is accessible to service developers for use outside the GJMF context.

The GJMF common library includes four parts:

- `Clients` - contains boilerplate solutions for service clients and service client factories. These service client abstractions hide the use of local call optimizations within the GJMF, provide transparent factory mechanisms for creation of client instances, and perform client-side marshalling of data types.

- `Interfaces` - contains definitions of all service interfaces for the GJMF, including base interfaces that service interfaces are derived from. These interfaces are used in the GJMF client APIs and abstract all service to service interaction in the GJMF.

- `Types` - contains all type definitions used in GJMF service interfaces, including WSDL stub type to immutable wrapper translation mechanisms for marshalling of Web Service invocations and notifications. These type definitions encapsulate all state and log information for the GJMF, and provides boilerplate solutions for state management.

- `Utilities` - contains utility functions and mechanisms for the GJMF services such as boilerplate solutions for service implementations, tools for management and delegation of credentials, service configuration solutions, and service resource management mechanisms.

All parts of the common library are used cooperatively to reduce the length of service development cycles and produce robust service implementations. The `Clients` and `Interfaces` modules are used for producing service

client APIs, the `Utilities` modules to produce service back-end implementations, and the `Interfaces` and `Types` modules to define service interaction protocols. One example of the flexibility of the service interaction model is the use of JSDL documents to convey both job specification data and processing hints, e.g., middleware submission parameters and queue information markers. To facilitate this model, and simplify use of the framework, all data exchanged with services in the GJMF have dedicated immutable wrapper types defined. All GJMF service interfaces have also been specified as Java interfaces, operating exclusively on these wrapper types. The common library provides all services with a configuration mechanism, providing service back-ends with dynamic access to configuration data from configuration files.

4.2.9. The GJMF Client Application Programming Interface

The GJMF client Application Programming Interface (API) is a set of Java classes abstracting the use of the GJMF Web Services for Java programmers. Mimicking the interface of the GJMF services, the client API is designed to provide intuitive use of the framework to developers with limited experience of Web Service and SOA development. As illustrated in Figure 7, and discussed in Section 4.2.8, the GJMF client API transparently handles local call optimizations, state notification management, and service instance management [22]. All GJMF functionality provided to service clients and end-users are accessible through both the GJMF services and the GJMF client API.

5. Architecture Discussion

To meet the flexibility and adaptability requirements discussed in Section 2, we build upon and extend the software development model previously presented in [22]. Key approaches in this model include use of Service-Oriented Architectures (SOAs) [53], design patterns, refactorization methods, and techniques to improve software adaptability such as dynamic configuration techniques and provisioning of software customization points. All software is developed in Java using common open source tools such as Eclipse, Apache Ant, and Apache Axis. The Globus Toolkit [26] is employed as a development environment for the production of Grid-enabled Web Services compatible with established Grid security models.

5.1. Invocation Patterns

The services of the GJMF support two basic modes of service invocation; sequential (regular) service invocation and batch invocation. In batch invocations, a set of service requests are bundled and sent to the service in a single service invocation. The batch invocation mode allows service clients to, e.g., submit a set of tasks to the TMS in a single request, significantly reducing service invocation makespan. Batch invocations conserve network bandwidth and reduce service invocation memory footprints on the server side. To simplify service invocation semantics, sets of requests sent using batch invocation modes are processed as transactions by the services in the GJMF. That is, if, e.g., a job submission in a batch request fails, other job submissions in the batch are canceled and rolled back if processed.

When service clients are codeployed with the GJMF services, i.e. residing inside services deployed in the same service container as the GJMF, service invocations are by default routed through the GJMF local call optimization framework. GJMF local call optimization mechanisms observe that services hosted in the same container share the same process space, and thus operate in the same Java Virtual Machine (JVM), and bypass service request serializations to allow service clients to directly invoke methods in the service implementation back-end. Use of local call optimizations greatly reduce service invocation time and memory footprint of service request processing, allowing for greater scalability in service implementations, more fine-grained communication models for interservice communication, and promotes a model of service aggregation where modules from constituent services can function as local Java objects in aggregated services [22]. When building systems aggregated from services there are also indirect benefits of this model. In the GJMF, this results, e.g., in a reduced need for polling to maintain distributed state coordination as state update notifications are less likely to be dropped due to excessive service container load. All services of the GJMF can be distributed in separate service deployments, but are recommended to be deployed in the same service container for performance reasons.

As any GJMF service can at any time be invoked directly by a service client, regardless of whether or not it is used as part of the framework, service invocation patterns can be hard to predict and are likely to vary over time. For this, as well as for reasons of transparency, all interservice communication is routed through the GJMF client API, which allows invocation modes and service communication optimizations to be ubiquitous and completely transparent to services, service clients, and end-users.

5.2. Deployment Scenarios

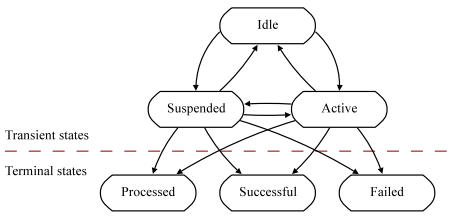
The GJMF has been designed to be as versatile as possible in terms of deployment and usage without imposing complexity of administration or loss of user control. The dynamic configuration structures, and the customizable code modules used throughout the framework provide options for modification of framework behavior combined with fully functional default configurations.

The hierarchical architecture of the GJMF is intended to provide clients a set of job management interfaces that offer an increasing range of automation of the job submission process without sacrificing user control. Services in lower layers offer fine-grained job submission interfaces with high degrees of explicit control, while services in higher layers attempt to automate the job submission process and offer control through configuration of behavior and optional use of customization point modules. The construction of the framework as a SOA with local call optimizations allows the framework transparent distribution of components combined with high efficiency in interservice communication when services are codeployed.

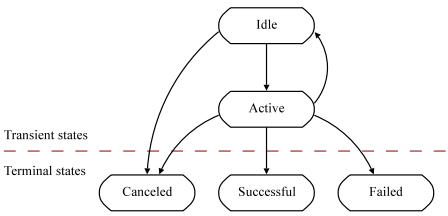
Envisioned usage scenarios for the framework include, e.g.,

- Running the framework on a gateway server to act as a middleware-independent multi-user Grid job submission interface.
- Running the framework on a client computer to act as a convenient personal job submission and management tool for Grids access.
- Running multiple instances of the framework to provide partitioning and load balancing of large job submission queues and multiple Grids.
- Running multiple instances of the framework utilizing different configurations to provide alternative job submission behaviors.

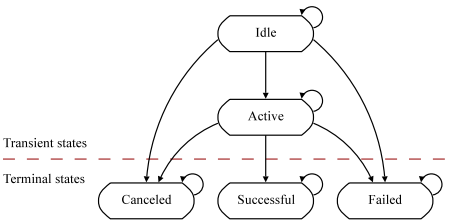
A natural overlap between these usage scenarios exist, and each of these are expected to be seen in hierarchical or other types of federated Grid environments, as well as in federated Cloud computing systems. Typical usage scenarios for the GJMF are expected to include hierarchical (or other forms of) combinations of multiple deployments of the framework, on top of multiple Grid middlewares and resource manager systems. To meet advanced application requirements, e.g., transparent workflow enactment, the GJMF is expected to be utilized in combination with high-level tools such as the Grid Workflow Execution Engine (GWEE) [19].



(a) The GJMF task group state model.



(b) The GJMF task state model.



(c) The GJMF job state model.

Figure 8: The GJMF state models. Task group states are used in the TGMS, task states in the TGMS and the TMS, job states in the JCS. The JCS job state model is semantically identical to the state model of the OGSA BES [28]. The recurring states of the GJMF job state model are used to incorporate and abstract state information from more fine-grained Grid middleware state models.

The deployment and utilization flexibility of the GJMF makes the framework viable for application within a number of computational settings, including high-performance computing (HPC) (depending on support from underlying middlewares for some functionality, e.g., execution of MPI jobs), high-throughput computing (HTC), as well as the more recently defined many-task computing (MTC) [57] paradigm. In MTC, focus is placed on enactment of loosely coupled applications constituted by large numbers of short-lived, data intensive, heterogeneous tasks with high (non-message passing) communication requirements, a setting envisioned in the design of the GJMF.

5.3. State Models

As the GJMF is composed of (possibly distributed) interoperating services, state management and coordination is inherently complex. To address this, the GJMF employs a hierarchical model for distributed state updates,

State	Interpretation
Transient states	
Idle	Work unit successfully submitted.
Active	Work unit currently being processed.
Suspended	Work unit temporarily suspended (TGMS).
Terminal states	
Successful	Work unit successfully processed.
Canceled	Work unit processing canceled.
Failed	Work unit processing failed.
Processed	Work unit processed with partial success (TGMS).

Table 1: GJMF state interpretations.

where each service hosting a job description resource is responsible for coordinating state updates to clients. As state updates for services are delivered via WS-BaseNotifications, the distributed state model of the GJMF is event driven; services respond to state changes in lower layers by updating state and producing notifications that are propagated up the service hierarchy. To compensate for dropped state notifications due to network failures or service container load, all services implement a state monitoring mechanism that regularly checks for missing notifications through polling. This mechanism simplifies state management and allows framework state coordination mechanisms to consider state delivery transparent and reliable.

As illustrated in Figure 8, each type of GJMF job definition has a corresponding finite state model that drives the processing of jobs, tasks, and task groups in the GJMF. In this processing, a job, task, or task group is referred to as a work unit, and is assigned an individual work unit context which is exposed to clients through service interfaces and WS-ResourceProperties. Table 1 gives a brief summary of work unit processing state interpretations in the GJMF.

5.4. Data Management

To maintain middleware transparency, the GJMF does not by default actively participate in data transfers between clients and computational resources. The GJMF assumes that data files are available and can be transfered to and from computational resources by the enacting Grid middleware

via a file transfer mechanism chosen by the middleware. File staging information is conveyed from clients to middlewares by the GJMF as part of job descriptions, typically in the form of GridFTP [16] URL tags in job and task JSDL.

In the GJMF, file transfers are expected to be initiated and performed by the enacting Grid middleware, existing data files are expected to be available prior to job submission (i.e., the GJMF does not verify the existence of data files during brokering), and computational resources and clients are responsible for maintaining file system allocations capable of accommodating incoming and outgoing data files respectively. If required, JSDL annotations can be used to provide job brokering hints related to storage requirements for computational elements.

Data transfer URLs are translated by the JTS to formats recognized by the underlying middleware as part of the job description translation process. If the underlying middleware does not support file staging, the JCS customization points can be used to provide data transfer capabilities as part of the middleware job submission process without coupling GJMF clients or services to underlying middlewares. Plans to extend the GJMF with utility mechanisms and services for data management are under consideration, see Section 8.

5.5. Resource Brokering

To decouple the GJMF services from Grid middlewares and each other, all job to computational resource brokering activities are in the GJMF abstracted by the RSS, which in turn relies on Grid middleware information systems for monitoring of computational resource availability, characteristics, and load. As middleware information systems typically contain large volumes of cached information, and federated Grid environments are likely to contain multiple concurrent job submission and management systems, it is observed that a brokering component will always operate on information deprecated to some extent [24].

In the GJMF model, the RSS has been limited to provide computational resource recommendations and rankings, services and clients are expected to handle submission and failure handling for jobs without providing feedback to the RSS. This abstraction implies that the RSS is agnostic of whether a particular execution plan is enacted or not. To compensate for middleware information system update latencies, it would be possible for the RSS to maintain an internal cache of prior execution plans and update resource load

weights through speculation based on this information. As the RSS enforces user-level isolation of service capabilities, a unique cache would be created for each user and restricted to contain only recommendations for that user.

To improve quality of job to resource brokering, it would also be possible to interface the RSS with Grid accounting and load balancing systems, e.g., the SweGrid Accounting System (SGAS) [34], as well as provide the RSS with feedback from the JCS or job submission systems such as the Job Submission Service (JSS) [24]. To reduce system complexity and maintain a clean separation of concerns, the RSS does not implement speculative resource load prediction or brokering behavior, but offers customization points for third party implementation of advanced brokering algorithms where such feedback loops can be implemented without affecting the design of the framework.

The current implementation of the RSS is to be regarded a prototype, we foresee development of additional RSS versions with resource selection capabilities of particular interest for certain users [23]. Evaluation of RSS brokering performance and quality of execution plans is out of scope for this work.

5.6. Security

The GJMF employs the Grid Security Infrastructure (GSI) [30] security model provided by the Globus Toolkit [26], and can be configured to use the Secure Message, Secure Conversation, or Credentials Delegation (i.e. use of the Globus Delegation Service) communication mechanisms. Client and service security modes are individually configured using security descriptors, and service clients identities are established and verified for all service invocations from standalone clients. For service invocations using GJMF local call optimizations, i.e. from clients codeployed with the service invoked, credential proxies are accepted from the caller without verification of caller identity. This relaxation of authentication is done for performance reasons and is deemed as acceptable for situations where services trust the deployment environment of the service, and where service environments trust software deployed in it. Should verification of caller identity be required, GJMF local call optimizations can be disabled or replaced with Axis local call optimizations.

All types of job definitions, including task groups, are upon submission to a GJMF service associated with a set of user credentials used for, e.g., user authentication, resource ownership, and job execution privileges. User credentials are inherited in subsequent submissions within the GJMF, i.e.

task group credentials are assigned to tasks upon submission to a TMS, and jobs are assigned task credentials when submitted to a JCS. Task groups distinguish themselves from jobs and tasks by the ability to be suspended in execution, e.g., upon expiration of task group credentials.

For each user invoking a service in the GJMF, a separate service implementation (back-end) is instantiated and used for request processing. This imposes a degree of user-level isolation of service functionality and enforces sandboxing of service resources between users. Service caller identity is also used to enforce a similar restriction of access to service WS-Resources.

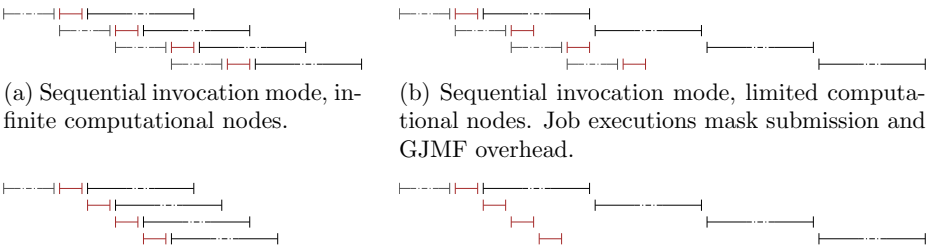
To facilitate the construction of job submission proxies, a requirement in, e.g., Grid portal construction [21], it is possible to submit a task to the GJMF specifying different credentials for job execution than those used in submission to the GJMF. For these situations, the authenticity of caller credentials are validated in the GJMF service invocation and the authenticity of the submission credentials are validated in the Grid middleware job submission process. As tasks will inherit credentials from task groups, as jobs will from tasks, resulting GJMF resources will be owned by the identity of the credentials used for job execution rather than the caller identity. This means that, e.g., a task group submitted using a certain set of user credentials will result in job submissions that use credentials belonging to that user, and only that user will be able to inspect details of the GJMF's processing of the task group (including LAS logs).

6. Performance Evaluation

To evaluate and analyze the performance of the framework prototype we run a series of tests using a standard setup of the framework on a deployment of a Grid middleware representative of production use.

6.1. Performance Measurement

To measure the efficiency of the framework, we define overhead as the time penalty imposed by use of the framework and use it as a cost function for efficiency. To quantify overhead incurred by the GJMF, we configure a GJMF deployment to operate on top of a Grid middleware and compare job submission performance and makespan to using the middleware directly for corresponding tasks. Total overhead imposed by the GJMF is in this performance evaluation computed as the total makespan of processing a group of jobs subtracted by the theoretical minimum time required to execute all jobs



(a) Sequential invocation mode, infinite computational nodes.

(b) Sequential invocation mode, limited computational nodes. Job executions mask submission and GJMF overhead.

(c) Batch invocation mode, infinite computational nodes.

(d) Batch invocation mode, limited computational nodes. Job executions mask GJMF overhead.

Figure 9: GJMF overhead components and invocation modes. Submission overhead, processing overhead, and execution overhead (illustrated in gray, red, and black, respectively) are independent components of the total makespan of a job.

in the group on an ideal system, i.e. a system that does not impose overhead associated with execution of jobs.

The overhead model used in the performance evaluation is illustrated in Figure 9, and expresses overhead associated with execution of groups of jobs. The illustration details four overhead scenarios spanned by the permutations of two invocation modes and two workload scenarios.

As illustrated in Figure 9, overhead associated with execution of an individual job is in the model divided into three sequential components; submission overhead, processing overhead, and execution overhead. Submission overhead is defined as overhead incurred prior to a job description being present in a GJMF service and typically consists of factors such as Java class loading and Web Service invocation time. Processing overhead is the GJMF contribution to the total overhead and consists of factors such as internal GJMF communication latencies and time spent performing job management tasks, e.g., job brokering and failure handling. Execution overhead is defined as time spent performing actions related to execution of a job on a computational resource, e.g., Grid middleware submission, file staging, job execution, execution environment clean-up, and status update delivery.

As also illustrated in Figure 9, parallel processing of job management activities allow the GJMF to partially mask individual overhead contributions through temporal overlaps with job executions and other job management activities. Total system overhead imposed by the GJMF is thus constituted by the sum of all overhead contributions associated with individual jobs sub-

tracted by overhead the GJMF is able to mask by parallel execution of job management tasks.

When the number of available computational hosts exceeds the number of jobs, the GJMF ability to mask overhead is limited and total system overhead bound by the submission and processing overhead components, as illustrated in Figure 9a and Figure 9c, respectively. When the number of jobs exceed the number of available computational hosts, total system overhead will be bound by the job execution overhead component. The GJMF ability to mask overhead contributions from individual jobs in these situations is illustrated in figures 9b and 9d.

To isolate individual contributions to the total system overhead we employ deployment options designed to minimize the contribution and impact of external, i.e. non-GJMF, overhead components, and measure job submission time and makespan for all GJMF job management components. To quantify the GJMF contributions to total system overhead, measurements of Grid middleware overhead are used as a comparative baseline for the minimum time required to process groups of jobs.

6.2. Test Environment

As the tests of the performance evaluation focus on illustrating overhead imposed by use of the GJMF, a limited test environment is sufficient for testing as these performance limitations are independent of the number of computational resources used, and will be representative for larger-scale use.

The test environment used in the evaluation is comprised of four identical 2 GHz AMD Opteron CPU, 2 GB RAM machines, interconnected with a 100 Mbps Ethernet network, and running Ubuntu Linux 2.6 and Globus Toolkit 4.0.5. Another set of four identical 1.8 GHz quad core AMD Opteron CPU, 4 GB RAM machines, interconnected using a Gigabit Ethernet network, and running Ubuntu Linux 2.6, Torque 2.3, and Maui 3.2.6 are employed as computational nodes in job throughput tests. The Java version used in tests is 1.6.0, and Java memory allocation pools range in size from 512 MB to 1 GB.

We employ GT4 WS-GRAM as Grid middleware and run */bin/true* executions for ideal jobs (zero execution time) and */bin/sleep* executions for jobs with known, non-zero execution times. To maximize the impact of the GJMF overhead when testing ideal jobs, we utilize the GT4 Fork mechanism for job dispatchment. For tests of more realistic scenarios we use */bin/sleep*

to get exact job execution times and use the GT4 PBS module for job dispatchment, which submits jobs to a local cluster using Torque. To minimize impact of stochastic network behaviors in our overhead measurements we do not use jobs that involve file transfers.

In all tests, one machine deploys the GJMF (or the WS-GRAM client) and the other three act as WS-GRAM/GT4 resources. For the GJMF tests, the RSS retrieves GT4 Monitoring and Discovery Service (WS-MDS) information from one of the three resources, which aggregates information from the other two. A single instance of each GJMF service is deployed in a common service container, and the services are configured (by default) to use local call optimizations for invocations.

In the tests, we use the GT4 WS-SecureConversation [5] security mechanism with client and service security descriptors in all Web Service invocations, including communication with the underlying Grid middleware. This mechanism performs both authentication and encryption of communication channels and will increase communication overhead and significantly reduce invocation throughput for Web Service invocations. The security setup used is deemed representative for intended production use in federated Grid environments.

A major performance factor in service invocation using Java is the impact of Java class loading. In service-to-service invocations, class loading overhead will impact framework performance differently than in client-to-service interaction, as services are more likely to have a class loaded, and may utilize local call optimizations when codeployed in the same container. Typically, overhead associated with Java class loading will impact performance severely during the submission phase, and show up in measurements as a one-time initial performance cost that obscure contributions of individual overhead components. In these tests, all service clients have been codeployed with the GJMF services to minimize the (potentially stochastic) impact of Java class loading issues on client performance. To emulate behavior of standalone service clients, full Web Service invocations are made between clients and services. For tests of codeployed clients local call optimizations are used.

6.3. Performance Tests

The purpose of the performance evaluation is to investigate and quantify individual contributions to total system overhead, and to verify that overhead imposed by the GJMF is sufficiently small in relation to the functionality offered by the framework. In the performance evaluation, we perform a set of

tests of service invocation capabilities, job submission performance, and job throughput to quantify and evaluate the impact of the GJMF overhead on the total system overhead. The tests performed are based on the overhead model presented in Section 6.1 and are designed to illustrate individual aspects of the framework overhead. The five types of tests performed are:

1. Job submission tests (Section 6.3.1). Investigate GJMF service client overhead associated with job submission and illustrate impact of, and trade-offs between, different service deployment and invocation methods.
2. Job throughput tests for ideal computational settings (Section 6.3.2). Investigate service-side overhead for scenarios illustrated by figures 9a and 9c, where the number of available computational resources exceed the number of jobs. This test setting constitutes a worst-case scenario for GJMF overhead and serves to quantify an upper bound for overhead imposed by use of the framework.
3. Job throughput tests for realistic computational settings (Section 6.3.3). Investigate service-side overhead for scenarios illustrated by figures 9b and 9d, where the number of jobs exceed the number of available computational resources. These tests illustrate the GJMF's ability to mask overhead by parallel processing of job management and execution activities.
4. Service invocation capability tests (Section 6.3.4). Investigate invocation throughput for the GJMF auxiliary services to quantify their contributions to the total system overhead and illustrate trade-offs between service communication overhead and service complexity.
5. Service invocation optimization tests (Section 6.3.5). Investigate performance trade-offs for different types of service invocation optimization mechanisms and illustrate impact of local call optimizations and their ability to reduce service communication overhead.

6.3.1. Job Submission

To evaluate the submission overhead component of the total system overhead, we measure the framework's job submission throughput and quantify overhead incurred by the GJMF against a baseline measurement of the GT4 WS-GRAM job submission performance. To illustrate trade-offs involved when using the GJMF from service clients, we perform tests using sequential and batch invocation modes for Web Service invocations and local call optimization invocations.

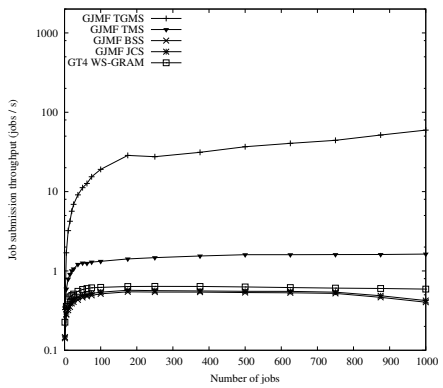
For all tests, job, task, and task group submission performance is measured as turn-around time for submission in service clients using realistic job descriptions. The average job submission makespan is used as a direct measurement of the overhead incurred by the GJMF for job submission.

As can be seen in Figure 10, JCS and BSS job submission throughput is slightly lower than that of GT4 WS-GRAM. This result is expected as both these services perform synchronized invocations to the underlying middleware for job submission, and thus add their overhead contributions to the middleware's overhead contribution. The JCS also performs a job description translation from JSDL to GT4 RSL (via a JTS) and in addition to this, the BSS also performs a task to resource matching (via a RSS). TGMS and TMS throughput is higher than GT4 WS-GRAM throughput as they contain submission buffers that allow them to perform asynchronized processing of jobs, resulting in delayed submissions to underlying services. The TGMS exhibits the highest throughput as it submits multiple tasks in single service invocations. As can be seen in Figure 10b, use of batch invocation modes enables the TMS to submit multiple tasks in a single WS invocation, and thus increase submission throughput. Compared to the TGMS however, TMS throughput is slightly lower. This is due to the TMS incurring overhead from multiple synchronized calls to the TMS service back-end during the submission phase.

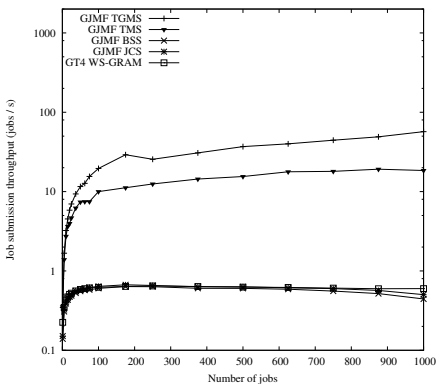
When using local call optimizations, as illustrated in Figure 10c and 10d, submission overhead can be reduced for all GJMF services. The TGMS and TMS achieve very high submission throughput due to their ability to perform asynchronous job submissions. Use of local call optimizations reduce invocation overhead to a range where impact of this overhead component becomes almost negligible.

6.3.2. Job Throughput for Ideal Computational Settings

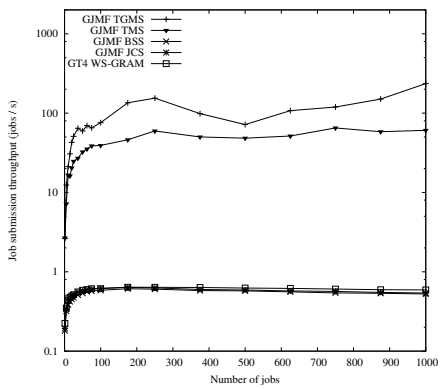
To evaluate and get an upper bound for the processing overhead component of the total system overhead, we measure the job processing capacity of the framework in terms of throughput and quantify overhead incurred by the GJMF against a baseline measurement of the GT4 WS-GRAM job processing performance when the GJMF's ability to mask overhead is minimized. As indicated in Figure 9, this occurs in situations where the number of available computational resources exceeds the number of jobs. To simulate this, and isolate and maximize the impact of the GJMF overhead, we use jobs with zero execution time, i.e. */bin/true* executions, submitted to the GT4 middle-



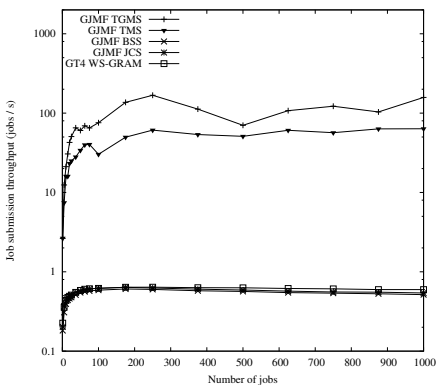
(a) GRAM and GJMF job submission throughput. Web Service invocations, sequential job submission.



(b) GRAM and GJMF job submission throughput. Web Service invocations, batch job submission.



(c) GRAM and GJMF job submission throughput. Local call optimization invocations, sequential job submission.



(d) GRAM and GJMF job submission throughput. Local call optimization invocations, batch job submission.

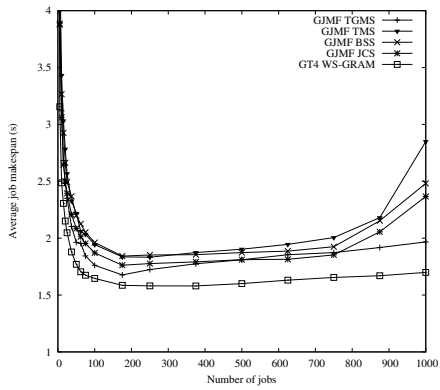
Figure 10: GRAM and GJMF job submission performance. Job submission throughput as a function of number of jobs, vertical axis logarithmic.

ware using the Fork dispatcher, which starts all jobs in parallel on the same machine with minimal delay. As this test setting will minimize the GJMF's ability to mask processing overhead through task parallelization, it will constitute a worst-case scenario for GJMF overhead and is used to quantify an upper bound for the GJMF overhead (for non-failing jobs). Job, task, and task group throughput are measured using sequential and batch invocation modes for Web Service invocations and local call optimization invocations.

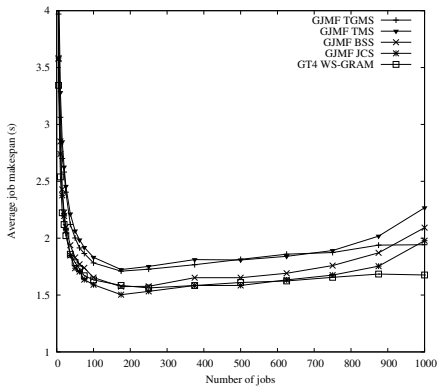
As can be seen in Figure 11, the GJMF incurs an average performance penalty of less than one second per job for ideal (zero execution time) jobs. This overhead includes factors such as job submission, interservice communication, job brokering, and distributed state management. When using batch invocation modes for Web Service invocation job submissions (Figure 11b) the overhead incurred can be somewhat mediated for the GJMF services. Particularly, the overhead for using the JCS is reduced to a level close to that of using GT4 WS-GRAM directly. This is due to the fact that the JCS does not perform any type of task to resource matching. The BSS and the services using the BSS, i.e. the TGMS and the TMS, suffer overhead from the brokering process that, as illustrated in figures 9a and 9b, is partially masked by the submission overhead when using sequential invocation modes (Figure 11a).

When using service clients codeployed with the GJMF, as illustrated in figures 11c and 11d, GJMF local call optimizations allow the JCS overhead to be reduced to close to GT4 WS-GRAM performance regardless of invocation mode. Local call optimizations do not greatly affect the throughput of the other GJMF services as these are still bound by the brokering overhead. It is worth noting that while local call optimizations do not increase throughput in these tests, they do reduce memory load for clients and services involved, promoting system scalability. BSS brokering overhead can also be masked by external overhead and job execution times, allowing higher order GJMF services to approach WS-GRAM throughput.

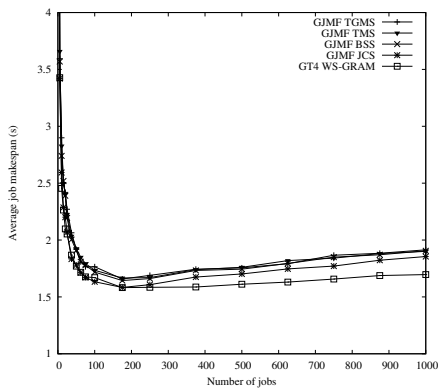
Use of the GT4 Fork mechanism for job dispatchment results in all jobs executing as spawned processes on the local machine. Despite the use of a computationally cheap process, this still causes increased load on the machine that in the measurements show as a slight decrease in average job throughput for all services (including the WS-GRAM) as the number of jobs increase. In tests using large numbers of jobs, use of full Web Service invocations results in memory starvation effects in the service container, negatively affecting service processing throughput. This effect can be observed for the TMS, BSS, and



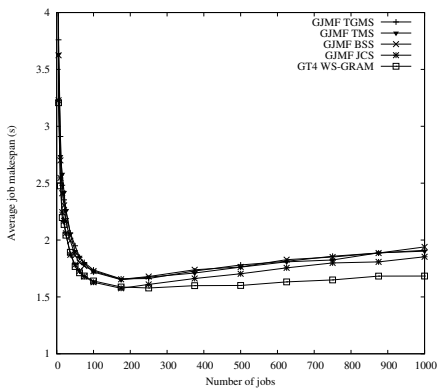
(a) GRAM and GJMF job processing throughput. Web Service invocations, sequential job submission, ideal jobs.



(b) GRAM and GJMF job processing throughput. Web Service invocations, batch job submission, ideal jobs.



(c) GRAM and GJMF job processing throughput. Local call optimization invocations, sequential job submission, ideal jobs.



(d) GRAM and GJMF job processing throughput. Local call optimization invocations, batch job submission, ideal jobs.

Figure 11: GRAM and GJMF job processing performance for ideal computational settings. Average job makespan as a function of number of jobs.

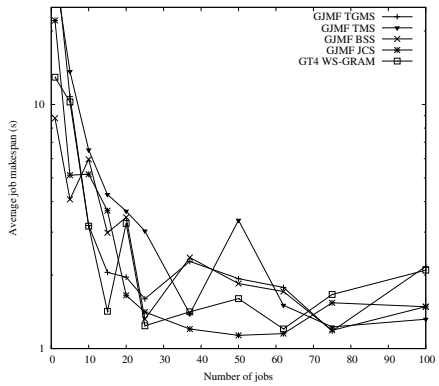
JCS in figures 11a and 11b. Note that the TGMS does not suffer from this effect as it performs single service invocations for task group submissions, and uses delays between subsequent TMS task submissions. Note also that use of batch invocation modes alleviates this effect, but does not eliminate it as back-end invocations still marshal requests and create job and task resources.

6.3.3. Job Throughput for Realistic Computational Settings

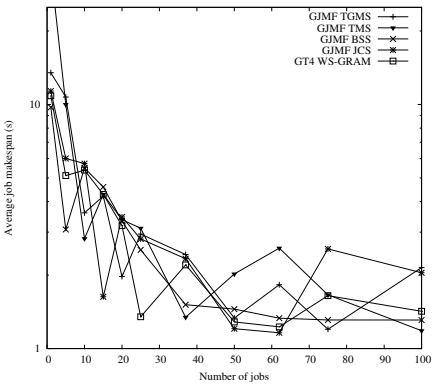
To evaluate the processing overhead component of the total system overhead under more realistic circumstances, we measure the job processing capacity of the framework in terms of throughput and quantify overhead incurred by the GJMF when deployed with a production environment system (GT4 and PBS Torque) against a baseline measurement of the GT4 WS-GRAM job processing performance. In these tests, computational power is limited as the number of jobs exceed the number of available computational resources, allowing the GJMF to mask overhead through parallel task processing. To establish a theoretical minimum time required to execute a set of jobs, we employ */bin/sleep* jobs of a known, non-zero execution length in the tests. Job, task, and task group throughput are measured using sequential and batch invocation modes for Web Service invocations and local call optimization invocations.

In Figure 12, a theoretical minimum time for execution of a set of jobs (based on number of jobs, job execution length, and number of available computational hosts) has been subtracted from each measurement to better illustrate remaining overhead components. As illustrated, a stochastic element has now been introduced to the overhead model for the system. This is a result of using the PBS scheduler, which has two polling intervals for job submission and job status inspection (in the tests set to 60 and 120 seconds respectively). PBS Torque also implements a behavior where jobs arriving to an empty PBS queue are scheduled faster than the scheduling interval may suggest. In the tests job execution lengths are set to 60 seconds, which combined with the PBS scheduling intervals result in each set of jobs receiving an overhead contribution from PBS of between 0 and 180 seconds depending on when in the scheduling cycle a job arrives and terminates. PBS overhead contribution appears stochastic as the GJMF and the PBS scheduling mechanisms are not synchronized. The GJMF overhead has in the tests been partially masked by job execution times and is, independently of invocation mode and mechanism, small enough to be masked by the PBS component.

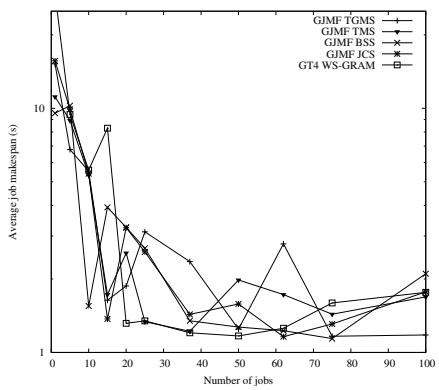
The term realistic computational settings used in this test refers to the



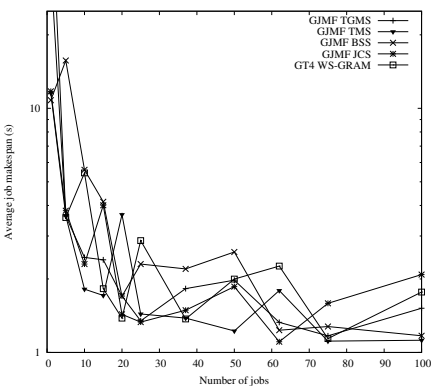
(a) GRAM and GJMF job processing throughput. Web Service invocations, sequential job submission, non-ideal jobs.



(b) GRAM and GJMF job processing throughput. Web Service invocations, batch job submission, non-ideal jobs.



(c) GRAM and GJMF job processing throughput. Local call optimization invocations, sequential job submission, non-ideal jobs.



(d) GRAM and GJMF job processing throughput. Local call optimization invocations, batch job submission, non-ideal jobs.

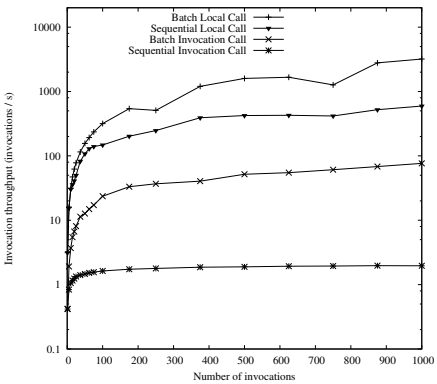
Figure 12: GRAM and GJMF job processing performance for realistic computational settings. Average job processing makespan as a function of number of jobs, vertical axis logarithmic.

job management components operating in a setting where non-zero job execution overhead and duration allow the GJMF to mask individual component overhead contributions. In realistic scenarios, job execution durations would typically be several orders of magnitude larger, and mask GJMF overhead even more. Job execution durations used here are selected to be sufficiently small to allow for greater numbers of tests.

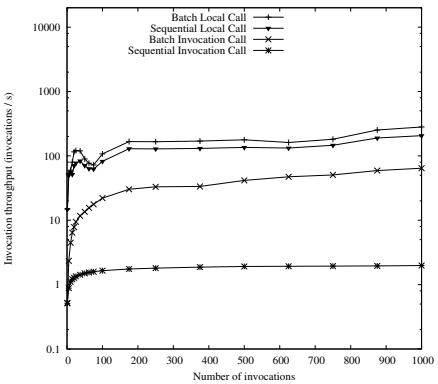
6.3.4. GJMF Auxiliary Services

To evaluate performance of the GJMF auxiliary services, quantify RSS overhead contributions in job throughput tests, and illustrate impact of invocation modes and mechanisms on interservice communication within the framework, we measure invocation capacity of the LAS, the JTS, and the RSS using sequential and batch invocation modes for Web Service invocations and local call optimization invocations. For all tests, typical GJMF tasks containing full JSDL documents are used as service invocation parameters. In LAS tests tasks are stored in logs, for JTS tests task JSDLs are translated to GT4 RSL, and in RSS tests tasks are brokered to computational resources.

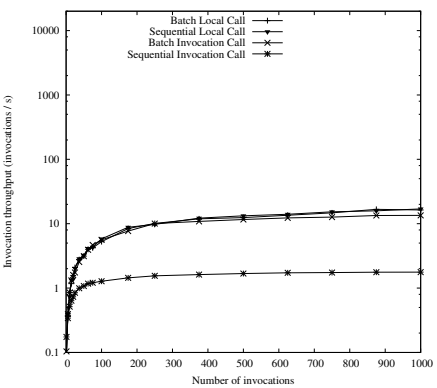
As can be seen in Figure 13, local call optimizations allow for much greater invocation throughput than Web Service invocations. This is natural as they avoid network transport and mitigate the need for message serialization and parsing. Use of batch invocation modes also increase invocation throughput as they reduce the number of service invocations required. As the LAS implements asynchronous storage queues, it allows for an asynchronous communication model and can thus reduce service client invocation overhead to be bound by communication overhead (Figure 13a). The JTS and RSS provide synchronous request processing models, and are therefor performance bound by the processing limitations of the service implementation as well as the communication overhead (figures 13b and 13c, respectively). The JTS is able to process requests in a manner efficient enough to increase invocation throughput by use of local call optimizations as it implements context-dependent job description translations through customization points. While the RSS implements background information retrieval for brokering information, the brokering process itself is complex enough to become the limiting factor for invocation throughput. In this case, use of local call optimizations does not greatly affect invocation throughput, but will serve to conserve memory in service invocation. As these measurements are made using the same setup as the job submission and throughput tests of sections 6.3.1,



(a) The Log Accessor Service.



(b) The JSDL Translation Service.



(c) The Resource Selection Service.

Figure 13: Invocation performance for the auxiliary services of the GJMF. Invocation throughput as a function of number of invocations, vertical axis logarithmic. Sequential and batch invocation mode performance for local call optimization and Web Service invocation calls.

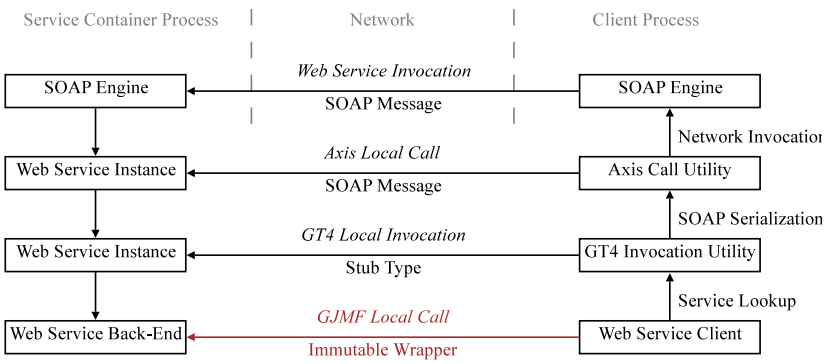


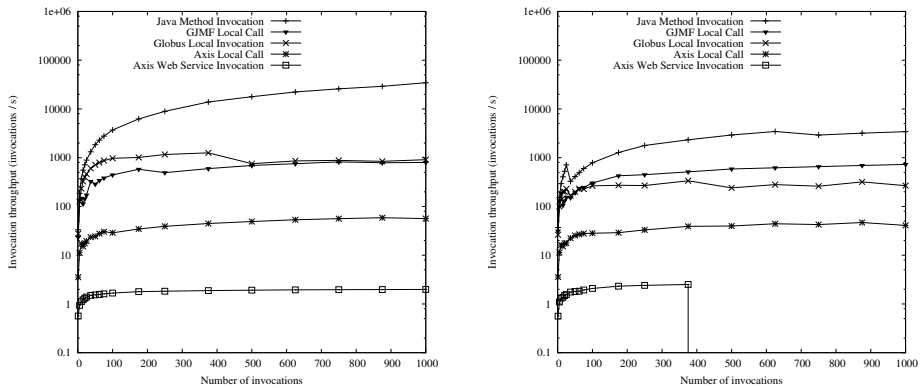
Figure 14: Local call optimization types. Illustrates actors and overhead involved.

6.3.2, and 6.3.3, the values for the local call optimization tests can be used as rough estimates of the individual overhead contributions of these services to the GJMF processing overhead.

6.3.5. Local Call Optimizations

To evaluate performance and impact of the GJMF local call optimization mechanisms, we measure invocation throughput for a reference service using the GJMF local call optimizations and compare it to invocation throughput for the same service using Axis Local Calls, Globus Local Invocations, Axis Web Service invocations, and direct Java method invocations to the service implementation. Invocations are made sequentially and in parallel (using a multithreaded service client) with small messages as parameters and a lean service method implementation to minimize the impact of memory starvation effects in the tests.

The different types of service invocation mechanisms used in the tests are illustrated in Figure 14. GJMF local call optimizations identify service implementation back-ends based on class name and perform marshalling of service invocation data using immutable wrapper types. Globus Local Invocations perform service implementation lookup through a Java Naming and Directory Interface (JNDI) [60] based container service registry and utilize generated stub types for service invocation data representations. Axis Local Calls locate service implementations through the same container service registry and perform full SOAP serializations of service messages.



(a) Sequential invocations.

(b) Parallel invocations.

Figure 15: Web Service invocation capacity comparison. Service invocation throughput as a function of number of invocations, vertical axis logarithmic.

It should be noted that the GJMF local call optimizations also provide local call capabilities for state notification delivery with comparable performance. This has not been evaluated in the tests as neither Globus Local Invocations or Axis Local Calls offer this functionality.

As illustrated in Figure 15, use of local call optimizations greatly improve service invocation throughput. The GJMF local call optimizations provide invocation performance comparable to existing Axis and Globus optimizations. All invocation methods scale well for parallel invocations, which is to be expected as they are designed for this use case. For large numbers of parallel invocations, the Axis Web Service invocation mechanism throughput drops drastically as the service container is unable to handle large numbers of concurrent service invocations due to memory and thread pool exhaustion issues.

While not illustrated by the tests, the GJMF local call optimizations require less memory than Axis and Globus invocation optimizations as the GJMF mechanisms do not perform message serialization, maintain message contexts, or invoke message handlers for local service invocations. While this trait does not directly affect service invocation times, it reduces the memory load of the service container when using WS-BaseNotification based notification schemes for services handling large numbers of objects, e.g., a TGMS containing large task groups. As can be seen in Figure 15a, the Globus lo-

cal invocation mechanism outperforms the GJMF local calls for sequential service invocations due to two factors. First, the Globus local invocation mechanism performs a caching of Web Service objects between invocations, something the GJMF is unable to do as the GJMF enforces a user-level isolation of service capabilities for each call. Second, the GJMF performs context-based type validation of job description data in immutable wrapper types, e.g., parses job descriptions and verifies that that all required information is present, a process that simplifies service development but imposes additional computational overhead. For the parallel invocation case illustrated in Figure 15b, the GJMF local call optimizations outperforms the Globus local invocations mechanism, which is attributed to the lower memory usage of the GJMF local call optimizations.

6.4. Performance Discussion

In the GJMF, job processing overhead is parallelized between services and, as illustrated in Figure 9, masked by job submission and job execution overhead. As also illustrated, parallelization of job execution is independent of GJMF overhead and a function of the number of computational nodes available. When the number of jobs exceeds the number of nodes available for immediate job submission (as illustrated in 9b and 9d), GJMF job processing will be performed in parallel with job executions, and job durations will mask impact of overhead incurred by the GJMF. For realistic scenarios, e.g. use of codeployed GJMF services in computational Grids, job durations are typically several orders of magnitude larger than overhead incurred by the GJMF and will help mask GJMF overhead even when large numbers of computational nodes are available. As Grids typically have high utilization rates and individual Grid users rarely have exclusive access to computational nodes, this effect is expected to effectively mediate the impact of GJMF overhead on total job makespan.

Furthermore, as the greater bulk of the GJMF job processing overhead is constituted of service invocation times, co-hosting the services of the GJMF allow GJMF local call optimizations to reduce the overhead contribution of the GJMF job processing mechanisms to a level where the initial job submission overhead component becomes dominant. When using standalone clients, i.e. clients not co-located with the GJMF, job submission overhead can be mediated to a one-time cost by using batch invocation modes (illustrated in 9c and 9d). In most cases, submission overhead will impact total overhead regardless of whether the GJMF is used or not, but the various invocation

and deployment modes of the GJMF can be used to mediate this component. GJMF processing overhead can be mediated by GJMF deployment and configuration options, e.g., by use of codeployment of services and local call optimizations. Use of batch invocation modes for service invocations conserve network bandwidth and reduce the memory footprint of both the GJMF services and GJMF service clients. Use of local call optimizations eliminates network bandwidth requirements and reduces memory used for service invocations to a minimum.

The overhead model used here (illustrated in Figure 9) is somewhat simplified as the GJMF will incur additional overhead associated with failure handling and resubmission in situations where jobs fail. As common causes for Grid job submission failures include, e.g., submission of erroneous job descriptions, Grid congestion scenarios (lack of available computational resources), and resource overload situations [50], the GJMF has been design to approach these situations using incremental back-off behaviors modeled after network failure handling protocols. As a result, the overhead component associated with failure handling is expected to quickly become dominant in the total system overhead for individual jobs, but should not affect other Grid jobs, resources, or end-users. As rational failure handling depends on the failure context, i.e. why and how a job fails, this behavior is hard to objectively quantify in general settings and has therefor not been evaluated in tests.

Tests reveal that use of the GJMF for job management imposes an average overhead of less than one second per job, and that the GJMF is able to partially mask this overhead by parallel processing of job management tasks and job executions. The mainstay of the GJMF overhead is constituted by service communication overhead, and can be mitigated by service invocation modes, codeployment of services, and use of local call optimizations. The individual overhead contributions of the GJMF auxiliary services are sufficiently small to not greatly affect the total system overhead of the GJMF. The local call optimizations of the GJMF perform competitively when compared to existing service invocation optimizations and provide additional functionality required by the deployment model of the GJMF. Local call optimizations provide great reductions in service invocation overhead and memory requirements for services, and serve to reduce total system overhead and increase scalability of service-based systems.

7. Related Work

A number of contributions that in various ways relate to the job management architecture proposed in this work have been identified. Standardization efforts such as JSDL [9], GLUE [8], OGSA BES [28], and OGSA RSS [31] have helped shape boundaries between niches in the Grid infrastructure component ecosystem, and directly impacted the design of the proposed architecture. Standardized Web Service and security technologies such as WSRF [27], WSDL [14], SOAP [40], and GSI [5] have outlined the architecture communication models, and Grid middleware and resource manager systems such as the Globus middleware [35], NorduGrid ARC [17], Condor [62], and BOINC [7] have all contributed to the design of the architecture's middleware abstraction layer. Standardization and interoperability efforts such as The Open Grid Services Architecture (OGSA) [29], the Open Middleware Infrastructure Institute (OMII Europe) [54], and Grid Interoperation/Interoperability Now (GIN) [39], as well as contributions such as [67], [47], [55], and [10] have provided perspective, insight, and inspiration regarding interoperability aspects of the architecture design.

The Grid resource management system survey presented in [48] provides a taxonomy of Grid job management systems. In this model, the GJMF is classified as a job management system providing soft quality of service for computational Grids. Resource organization, namespace, information system, discovery, and dissemination as defined in this model are all determined by the underlying middleware. Type of scheduler organization is determined by how the framework is employed, but is typically expected to be decentralized for multi-user use of the framework. Non-predictive state estimation models are currently provided by the RSS, along with event-driven and extensible (re)scheduling policies.

A set of job management systems exhibiting similarities in design or intended use have also been identified, and include, e.g., the GridWay Metascheduler [41], a framework for adaptive scheduling and execution of Grid jobs. Like the GJMF, GridWay builds on the Globus Tookit and offers an abstracted ("submit and forget") type of Grid job submission focused on reliable and autonomous execution of jobs. Both systems provide failover capabilities through resubmission of jobs, where GridWay offers job migration capabilities through checkpointing and migration interfaces, whereas the GJMF focuses on abstraction of Grid middleware capabilities and system composability, and offers coarse-grained resubmission policies in higher services. GridWay also

offers a performance degradation mechanism which may be used to detect and trigger job migration mechanisms. The GJMF assumes computational hosts maintain acceptably consistent performance levels and relies on Grid applications and middlewares to handle checkpointing and application preemption issues.

The Falkon [58] framework provides a fast and lightweight task execution framework focused on task throughput and efficiency in task dispatchment. Falkon is by design not a fully featured local resource manager, and achieves high job submission throughput rates through, e.g., elimination of features such as multiple submission queues and accounting, and the use of custom protocols for state updates. Both Falkon and the GJMF are service-based frameworks and make use of notifications for distributed state notifications, but are in essence designed for different use cases. Falkon is, e.g., designed for efficient job submissions and achieve much higher submission throughput that the GJMF, whereas the GJMF, e.g., provides middleware-independence to service clients.

The Minimum intrusion Grid (MIG) [46] is a framework aimed at providing Grid middleware functionality while placing as little requirements as possible on Grid users and resources. Building on existing operating system and Grid tools such as SSH and X.509 certificates, the MIG provides a non-intrusive integration model and abstracts the use of Grid resources through service-based interfaces. The approaches differ on a number of points, e.g., where the MIG uses a centralized and monolithic job scheduler the GJMF provides a framework of composable services and relies on underlying middlewares for job to resource submissions.

The Imperial College e-Science Networked Infrastructure (ICENI) [33] is a composable OGSA Grid middleware implementation based on Jini [44]. ICENI provides a semantic approach to build autonomously composable Grid infrastructure components where services are annotated with capability information and new services are instantiated through SLA negotiations with existing services. The ICENI composability approach differs from the GJMF one, whereas the GJMF only provides mechanisms for framework (re)composition and service customization. ICENI also exposes service implementations locally through the Jini registry, a mechanism similar to the GJMF local call optimizations, and provisions for plug-in implementations of schedulers and launchers [70] in a way similar to the GJMF RSS customization points. Compared to ICENI, the GJMF provides additional functionality in terms of higher-level abstractions of job management, client APIs, more

flexible deployment options, and greater standardization support.

The Job Submission Service (JSS) [24] is a resource brokering and job submission service developed in the GIRD [65] project. The JSS supports advanced brokering capabilities, e.g., advance reservation of resources and coallocation of jobs, customization of algorithms through plug-ins, and standards-based middleware-independent job submission. Compared to the JSS, the GJMF provides additional functionality in, e.g., management and monitoring of jobs and groups of jobs, client APIs, logging capabilities, and translation of job descriptions. Work on the GJMF began as a functionality extension and refactorization effort targeted towards the OGSA BES and RSS standardizations, and builds on experiences from the JSS project.

All of these approaches are considered to operate in, or close to, the Grid middleware layer in the GJMF architectural model, and could be integrated with the GJMF as Grid middleware providers.

eNANOS [59] is a resource broker that abstracts Grid resource use and provides an API-based model of Grid access. Internally, uniform resource and job descriptions combined with XML-based user multi-criteria descriptions provide dynamic policy management mechanisms facilitating use of advanced brokering mechanisms. Job and resource monitoring mechanisms are provided, and failure handling through resubmission of jobs is supported. The primary difference between eNANOS and the GJMF lies in the flexibility of the GJMF architecture, which allows dynamic composition of the framework and provides additional levels of abstraction of job management functionality. The GJMF also builds on more recent standardization efforts such as JSDL, WSRF, and the OGSA BES.

The Community Scheduler Framework (CSF4) [69] is an OGSA-based open source Grid meta-scheduler. Like the GJMF, CSF4 is constructed as a framework of Web Services, builds on GT4, provides WSRF compliance, and exposes abstractions for job submission and control. In addition to this, CSF4 also provides user-selectable job submission queues and a mechanism for advance reservation of resources (via local resource managers). Compared to the CSF4, the GJMF provides support for concurrent use of multiple middlewares, framework composability, standards compliance, and a Java-based client API.

The GridLab Grid Application Toolkit (GAT) [4] is a high-level application programming toolkit for Grid application development. The fundamental ideas behind the GAT and the GJMF are similar, both projects aim to decouple Grid applications from Grid middlewares by providing middleware-

independent Grid access through client APIs aimed at simplifying Grid application development. The GAT builds on the GridLab [3] architecture which aims to be a complete Grid utilization platform, providing, e.g., data management services (including data transfer and replica management capabilities), monitoring services, and services for visualization of data, while the GJMF provides a composable and lean architecture for Grid utilization focusing on functionality required for job management, and relying on underlying middlewares for functionality such as job control and file staging.

GridSAM [49] is a standards-based job submission system that builds on standardization efforts such as JSDL, and aims to provide transparent job submission capabilities independent of underlying resource manager through a Web Service interface. Similar to the asynchronous job processing of the GJMF, GridSAM employs a job submission pipeline inspired by the staged event-driven architecture (SEDA) [68] that allows for short response times in job submission. Fault recovery capabilities are in GridSAM built by persisting event queues and job instance information, similar to the failure handling mechanisms of the GJMF that provide redundancy and resubmission capabilities. Compared to GridSAM, the GJMF provides additional functionality for composition of the job management framework, external exposure of job description translation functionality, job monitoring capabilities, and multiple job submission and control modes.

Nimrod-G [12] provides a layered architecture for resource management and scheduling for computational Grids. Nimrod-G provides an economy-driven broker that supports user-defined deadline and budget constraints for schedule optimizations [1], and manages supply and demand of resources through the Grid Architecture for Computational Economy (GRACE) [11]. Like the GJMF architecture, the Nimrod-G provides layered abstractions of middleware access components and facilitates use of parameter-sweep style applications. While the GJMF lacks capabilities for economy-based scheduling decisions, it does offer customization points for these types of mechanisms in the RSS, and provides a flexible architecture that can incorporate such usage-pattern specific adaptations with only local modifications.

The Gridbus [66] broker is a Grid broker that mediates access to distributed data and computational resources, and brokers jobs to resources based on data transfer optimality criteria. Gridbus extends the resource broker model of Nimrod-G, defining a hierarchical model for job brokering containing separate resource discovery, Grid scheduling, and monitoring components. Like in the GJMF, tasks are defined as sequences of commands that

describe user requirements, including, e.g., file staging and job execution information, located within the task description itself. Task requirements drive resource discovery and tasks are resolved into jobs, here defined as units of work sent to Grid nodes, i.e. instantiations of tasks with unique combinations of parameter values. The Gridbus broker also abstracts use of multiple middlewares through a service-based interface. Differences between the two platforms include, e.g., Gridbus heuristics-based scheduling strategies, and the GJMF's ability to dynamically reconfigure framework deployment during runtime.

GMarte [6] is a Grid metascheduler framework exposing a high-level Java API for Grid application development. Like the GJMF, the GMarte architecture is built in layers and employs a middleware abstraction layer that abstracts use of multiple middlewares. GMarte also provides failure handling through resubmission of jobs, and extends upon this through provisioning for application-level checkpointing of job executions. GMarte exposes a Java client API, plug-in points for information system access, and a service-based interface through GMarteGS [51], which supports WS-BaseNotification based state updates. The GJMF differs from the GMarte on a number of points, e.g., through the use of standardization efforts like JSDL and the OGSA BES, and by providing a dynamically composable architecture.

All of these contributions are considered to operate on a layer higher than the Grid middleware layer in the GJMF architecture, and are as job management solutions considered alternative approaches to the GJMF. Each system could naturally be incorporated with the GJMF as Grid middleware accessors, or could with modifications utilize the GJMF in a similar manner. While there are many viable workflow-based approaches to Grid job management, e.g., ASKALON [25], Pegasus [15], and GWEE [19], these have been omitted here as the scope of this work is restricted to generic job management architectures rather than workflows. Naturally, with modifications, most of these could make use of the GJMF for middleware-independent Grid access.

Finally, a few slightly different approaches have been identified, e.g., P-GRADE [45], which is a high-level environment for transparent enactment of parallel and Grid execution of applications. P-GRADE abstracts use of Grid resources through Condor and Globus interfaces, and provides enactment of individual jobs, MPI jobs, and workflows through generation of job wrapper scripts that stage, checkpoint, and execute jobs on computational resources. P-GRADE also supports monitoring of jobs and resources through tools provided by the environment, and job migration through checkpointing.

Compared to P-GRADE, the GJMF provides a different approach, focusing on providing infrastructure for autonomic job management rather than facilitation of Grid execution of applications. The GJMF assumes the existence of Grid applications and provides functionality to automate the job management process, e.g., high-level abstractions for execution of groups of tasks and client APIs.

EMPEROR [2] is an OGSA-based Grid meta-scheduler framework for dynamic job scheduling. EMPEROR provides a framework for integrating performance-based scheduling optimization algorithms based on time-series analysis of job history, as well as support for advance reservations (through local resource managers). The GJMF does not perform speculative scheduling or advance reservations, but offers customization points in the RSS for injection of such mechanisms. Compared to EMPEROR, the GJMF provides a more flexible architecture, greater standardization support, and levels of job management abstractions.

The Application Level Scheduling (AppLeS) [13] project provides a methodology, application software, and software environments for adaptive scheduling and deployment of Grid applications. In the AppLeS methodology, project developers team up with application experts to develop customized scheduling agents for applications that dynamically generates schedules for application staging and execution in a continuous process. Here each agent perform resource discovery and selection, schedule generation and selection, and executes and monitors applications. AppLeS agents interact directly with resource managers, perform all application management tasks, including, e.g., file staging, and can enact collations of applications, e.g., parameter sweeps. The AppLeS agents are similar in concept to use of personal deployments of the GJMF as individual job management clients, but differ in both technology chosen and the fact that the GJMF defers much functionality to underlying middlewares.

8. Future Work

A number of possible future extensions to the proposed architecture have been identified and are under consideration for investigation.

- Data management. In a future extension, the GJMF is envisioned to be complemented with a service-based, middleware- and transport-independent data management abstraction that builds on top of mechanisms such as GridFTP and Grid Storage Brokers, and integrates

seamlessly with the GJMF services and service clients. Support for data management would need to be provided by implementations of GJMF middleware customization points in the JCS, as well as by GJMF service clients. Interesting research questions regarding this extension include investigation of how transport-independence can be maintained while providing efficient functionality abstractions well adjusted to seamless integration with generic job management solutions.

- Workflow management. While the GJMF currently integrates with workflow management solutions by offering middleware-independent Grid job management interfaces, the framework itself lacks support for execution of interdependent tasks. Inclusion of a middleware-independent tool for execution of task graphs and static workflows would provide clients with a fire-and-forget type of workflow management solution similar to the functionality offered by the higher-order services of the GJMF for tasks and task groups.

- Evaluation of experiences from production use. Experiences from future production use of the framework is expected to provide feedback and suggest alterations or redesigns of parts of the framework.

9. Conclusion

We have proposed a flexible and loosely coupled architecture for middleware-independent Grid job management built as a composable set of Web Services. Intended for use in federate Grid environments, the architecture makes no assumptions of central control of resources or omniscience in scheduling, and abstracts resource and system heterogeneity in multiple levels. Focus is placed on maintaining non-intrusive coexistence and integration models, and Grid and Web Service standardization efforts such as JSDL, WSRF, OGSA BES, and OGSA RSS are built upon and leveraged.

The architecture is organized in hierarchical layers of functionality, where services in layers abstract and aggregate functionality from underlying layers. Services in lower layers provide explicit job submission capabilities and a fine-grained control model for the job management process while services in higher layers attempt to automate the job management process and provide a more coarse-grained control model through preconfigured job control and failure handling mechanisms. The architecture is designed to decouple Grid applications for Grid middlewares and infrastructure components, and

abstracts Grid functionality behind generic Grid job management interfaces. Applications built on the framework will be loosely coupled to underlying Grids, gaining portability and flexibility in deployment, as well as ability to utilize heterogeneous Grid resources transparently.

In this work we have also presented a proof-of-concept implementation of the architecture that builds on emerging Grid and Web Service standards and supports a range of Grid middlewares. Middleware independence is provided the framework through a set of foundational middleware abstraction services and aggregated Grid job management functionality is built on top of these. Services of the framework are individually configurable, and can be customized through configuration and the use of plug-ins without affecting other framework components. Framework composition can dynamically be altered and will adapt to failures occurring in job submission or execution.

All services in the framework provide a degree user-level isolation of service capabilities that function as if each user has exclusive access to the framework. Any service can at any time be used by service clients as an autonomous job management component while concurrently serving as a component in the framework. The use of local call optimizations allow service composition techniques to be used to construct software that simultaneously function as networks of services and monolithic architectures. Use of service client factories embedded in the client API make local call optimizations completely transparent to services, service clients, and end-users.

The underlying software design principles developed within the project have been described and findings from the project have been presented along with an evaluation of the performance of the proof-of-concept implementation. Tests in the evaluation show that overhead imposed by use of the framework for job submission, brokering, monitoring, and control is small, on average less than 1 second per job, and that overhead imposed by the framework is partially masked by job execution times in realistic applications. Codeployment of services enabling the use of local call optimizations and batch service invocation modes can further reduce overhead imposed by the framework on both client and service side.

10. Acknowledgements

We extend gratitude to Peter Gardfjäll, Arvid Norberg, and Johan Tordsson who's prior work and feedback have provided a foundation to build upon in this work. We acknowledge the Swedish Research Council (VR) who have

supported the project under contract 621-2005-3667, and the High Performance Computer Center North (HPC2N) on who's resources the research has been performed.

References

[1] D. Abramson, R. Buyya, and J. Giddy. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems*, 18(8):1061–1074, 2002.

[2] L. Adzigogov, J. Soldatos, and L. Polymenakos. EMPEROR: An OGSA Grid meta-scheduler based on dynamic resource predictions. *J. Grid Computing*, 3(1–2):19–37, 2005.

[3] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor. Enabling applications on the Grid - a GridLab overview. *Int. J. High Perf. Comput. Appl.*, 17(4), 2003.

[4] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Toward generic and easy application programming interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, 2005.

[5] The Globus Alliance. Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective. http://www.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf, May 2009.

[6] J.M. Alonso, V. Hernández, and G. Moltó. Gmarte: Grid middleware to abstract remote task execution. *Concurrency and Computation: Practice and Experience*, 18(15):2021–2036, 2006.

[7] D.P. Anderson. BOINC: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004.

[8] S. Andreozzi, S. Burke, F. Ehm, L. Field, G. Galang, B. Konya, M. Litmaath, P. Millar, and J.P. Navarro. GLUE specification

v. 2.0. http://www.ogf.org/Public_Comment_Docs/Documents/2008-06/ogfglue2rendering.pdf, May 2009.

[9] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, A. S. McGough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) specification, version 1.0. http://www.ogf.org/documents/GFD.56.pdf, May 2009.

[10] N. Bobroff, L. Fong, S. Kalayci, Y. Liu, J.C. Martinez, I. Rodero S.M. Sadjadi, and D. Villegas. Enabling interoperability among meta-schedulers. In T. Priol et al., editors, *CCGRID 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 306–315, 2008.

[11] R. Buyya, D. Abramson, and J. Giddy. An economy driven resource management architecture for global computational power grids, 2000.

[12] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An architecture of a resource management and scheduling system in a global computational grid. *CoRR*, cs.DC/0009021, 2000.

[13] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-level middleware for the Grid m{1}. *Scientific Programming*, 8(3), 2000.

[14] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, May 2009.

[15] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, and D.S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[16] W. Allcock (editor). GridFTP: Protocol extensions to FTP for the Grid. http://www.ogf.org/documents/GFD.20.pdf, May 2009.

[17] M. Ellert, M. Grønager, A. Konstantinov, B. Kónya, J. Lindemann, I. Livenson, J. L. Nielsen, M. Niinimäki, O. Smirnova, and A. Wäänänen. Advanced resource connector middleware for lightweight computational Grids. *Future Generation Computer Systems*, 27(2):219–240, 2007.

[18] E. Elmroth, P. Gardfjäll, A. Norberg, J. Tordsson, and P-O. Östberg. Designing general, composable, and middleware-independent Grid infrastructure tools for multi-tiered job management. In T. Priol and M. Vaneschi, editors, *Towards Next Generation Grids*, pages 175–184. Springer-Verlag, 2007.

[19] E. Elmroth, F. Hernández, and J. Tordsson. A light-weight Grid workflow execution engine enabling client and middleware independence. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science, vol. 4967*, pages 754–761. Springer-Verlag, 2008.

[20] E. Elmroth, F. Hernández, J. Tordsson, and P-O. Östberg. Designing service-based resource management tools for a healthy Grid ecosystem. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science, vol. 4967*, pages 259–270. Springer-Verlag, 2008.

[21] E. Elmroth, S. Holmgren, J. Lindemann, S. Toor, and P-O. Östberg. Empowering a flexible application portal with a soa-based grid job management framework. In *The 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, to appear, 2009.

[22] E. Elmroth and P-O. Östberg. Dynamic and Transparent Service Compositions Techniques for Service-Oriented Grid Architectures. In S. Gorlatch, P. Fragopoulou, and T. Priol, editors, *Integrated Research in Grid Computing*, pages 323–334. Crete University Press, 2008.

[23] E. Elmroth and J. Tordsson. Grid resource brokering algorithms enabling advance reservations and resource selection based on performance predictions. *Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications*, 24(6):585–593, 2008.

[24] E. Elmroth and J. Tordsson. A standards-based grid resource brokering service supporting advance reservations, coallocation and cross-grid interoperability. *Concurrency Computat.: Pract. Exper.*, 2009. accepted.

[25] T. Fahringer, R. Prodan, R.Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and

M. Wieczorek. ASKALON: A development and Grid computing environment for scientific workflows. In I. Taylor et al., editors, *Workflows for e-Science*, pages 450–471. Springer-Verlag, 2007.

[26] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In H. Jin, D. Reed, and W. Jiang, editors, *IFIP International Conference on Network and Parallel Computing, LNCS 3779*, pages 2–13. Springer-Verlag, 2005.

[27] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, and S. Weerawarana. Modeling stateful resources with Web services. http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf, May 2009.

[28] I. Foster, A. Grimshaw, P. Lane, W. Lee, M. Morgan, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, and M. Theimer. OGSA© basic execution service version 1.0. http://www.ogf.org/documents/GFD.108.pdf, May 2009.

[29] I. Foster, H.Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, Version 1.5. http://www.ogf.org/documents/GFD.80.pdf, May 2009.

[30] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational Grids. In *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[31] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, version 1.5, 2006. http://www.ogf.org/documents/GFD.80.pdf, May 2009.

[32] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[33] N. Furmento, W. Lee, A. Mayer, S. Newhouse, and J. Darlington. ICENI: an open grid service architecture implemented with Jini. In *Pro-*

ceedings of the 2002 ACM/IEEE conference on Supercomputing, pages 1–10. IEEE Computer Society Press Los Alamitos, CA, USA, 2002.

[34] P. Gardfjäll, E. Elmroth, L. Johnsson, O. Mulmo, and T. Sandholm. Scalable Grid-wide capacity allocation with the SweGrid Accounting System (SGAS). *Concurrency Computat.: Pract. Exper.*, 20(18):2089–2122, 2008.

[35] Globus. http://www.globus.org. May 2009.

[36] S. Graham and B. Murray (editors). Web Services Base Notification 1.2 (WS-BaseNotification). http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf, May 2009.

[37] S. Graham and J. Treadwell (editors). Web Services Resource Properties 1.2 (WS-ResourceProperties). http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf, May 2009.

[38] S. Graham, A. Karmarkar, J. Mischkinsky, I. Robinson, and I. Sedukhin (editors). Web Services Resource 1.2 (WS-Resource). http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf, May 2009.

[39] Grid Interoperability Now. http://wiki.nesc.ac.uk/read/gin-jobs. May 2009.

[40] M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, H. Frystyk Nielsen, A. Karmarkar, and Y. Lafon. SOAP version 1.2 part 1: Messaging framework. http://www.w3.org/TR/soap12-part1/, May 2009.

[41] E. Huedo, R.S. Montero, and I.M. Llorente. A framework for adaptive execution on Grids. *Software - Practice and Experience*, 34(7):631–651, 2004.

[42] Cluster Resources inc. Torque resource manager. http://www.clusterresources.com/pages/products/torque-resource-manager.php, May 2009.

[43] ISO/IEC. ISO/IEC 9075:1992, Database Language SQL - July 30, 1992. http://www.contrib.andrew.cmu.edu/ shadow/sql/sql1992.txt, May 2009.

[44] Jini. http://www.jini.org, May 2009.

[45] P. Kacsuk, G. Dzsa, J. Kovcs, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombs. P-GRADE: a Grid programming environment. *Journal of Grid Computing*, 1(2):171 – 197, 2003.

[46] H.H. Karlsen and B. Vinter. Minimum intrusion Grid - The Simple Model. In *14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE'05)*, pages 305–310, 2005.

[47] A. Kertesz and P. Kacsuk. Meta-Broker for Future Generation Grids: A new approach for a high-level interoperable resource management. In *CoreGRID Workshop on Grid Middleware in conjunction with ISC*, volume 7, pages 25–26. Springer, 2007.

[48] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of Grid resource management systems for distributed computing. *Softw. Pract. Exper.*, 32(2):135–164, 2002.

[49] W. Lee, A. S. McGough, and J. Darlington. Performance evaluation of the GridSAM job submission and monitoring system. In *UK e-Science All Hands Meeting*, pages 915–922, 2005.

[50] H. Li, D. Groep, L. Wolters, and J. Templon. Job Failure Analysis and Its Implications in a Large-Scale Production Grid. In *Proceedings of the 2nd IEEE International Conference on e-Science and Grid Computing*, 2006.

[51] G. Moltó, V. Hernández, and J.M. Alonso. A service-oriented WSRF-based architecture for metascheduling on computational grids. *Future Generation Computer Systems*, 24(4):317–328, 2008.

[52] MySQL. http://www.mysql.com/, May 2009.

[53] OASIS Open. Reference Model for Service Oriented Architecture 1.0. http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf, May 2009.

[54] OMII Europe. OMII Europe - open middleware infrastructure institute. http://omii-europe.org, August 2008.

[55] G. Pierantoni, B. Coghlan, E. Kenny, O. Lyttleton, D. O'Callaghan, and G. Quigley. Interoperability using a Metagrid Architecture. In *ExpGrid workshop at HPDC2006 The 15th IEEE International Symposium on High Performance Distributed Computing*, Paris, France, February 2006.

[56] PostgreSQL. http://www.postgresql.org/, May 2009.

[57] I. Raicu, I.T. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1–11, 2008.

[58] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In *Proceedings of IEEE/ACM Supercomputing 07*, 2007.

[59] I. Rodero, J. Corbalán, R. M. Badia, and J. Labarta. eNANOS Grid Resource Broker. In P. M. A. Sloot, A. G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *Advances in Grid Computing - EGC 2005, LNCS 3470*, pages 111–121, 2005.

[60] Sun Microsystems. Java Naming and Directory Interface (JNDI). http://java.sun.com/products/jndi/, May 2009.

[61] Sun Microsystems. The Java Database Connectivity (JDBC). http://java.sun.com/javase/technologies/database/, May 2009.

[62] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency Computat. Pract. Exper.*, 17(2–4):323–356, 2005.

[63] The Apache Software Foundation. Apache Derby. http://db.apache.org/derby/, May 2009.

[64] The Globus Project. An "ecosystem" of Grid components. http://www.globus.org/grid_software/ecology.php, May 2009.

[65] The Grid Infrastructure Research & Development (GIRD) project. Umeå University, Sweden. http://www.gird.se, May 2009.

[66] S. Venugopal, R. Buyya, and L. Winton. A Grid service broker for scheduling e-Science applications on global data Grids. *Concurrency Computat. Pract. Exper.*, 18(6):685–699, May 2006.

[67] S. Venugopal, K. Nadiminti, H. Gibbins, and R. Buyya. Designing a resource broker for heterogeneous grids. *Softw. Pract. Exper.*, 38(8):793–825, 2008.

[68] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-connected scalable internet services. *Operating System Review*, 35(5):230–243, 2001.

[69] W. Xiaohui, D. Zhaohui, Y. Shutao, H. Chang, and L. Huizhen. CSF4: A WSRF Compliant Meta-Scheduler. In *The 2006 World Congress in Computer Science, Computer Engineering, and Applied Computing*, pages 61–67. GCA'06, 2006.

[70] L. Young, S. McGough, S. Newhouse, and J. Darlington. Scheduling architecture and algorithms within the ICENI Grid middleware. In Simon Cox, editor, *Proceedings of the UK e-Science All Hands Meeting*, pages 5 – 12, 2003.