Virtual Infrastructures for Computational Science: Software and Architectures for Distributed Job and Resource Management

Per-Olov Östberg

PhD Thesis, March 2011
Department of Computing Science
Umeå University
Sweden

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

p-o@cs.umu.se

ISBN 978-91-7459-194-1
ISSN 0348-0542
UMINF 11.02

Printed by Print & Media, Umeå University, 2011

Abstract

In computational science, the scale of problems addressed and the resolution of solutions achieved are often limited by the available computational capacity. The current methodology of scaling computational capacity to large scale (i.e. larger than individual resource site capacity) includes aggregation and federation of distributed resource systems. Regardless of how this aggregation manifests, scaling of scientific computational problems typically involves (re)formulation of computational structures and problems to exploit problem and resource parallelism. Efficient parallelization and scaling of scientific computations to large scale is difficult and further complicated by a number of factors introduced by resource aggregation, e.g., resource heterogeneity and coupling of computational methodology. Scaling complexity severely impacts computation enactment and necessitates the use of mechanisms that provide higher abstractions for management of computations in distributed computing environments.

This work addresses design and construction of virtual infrastructures for scientific computation that abstract computation enactment complexity, decouple computation specification from computation enactment, and facilitate large-scale use of computational resource systems. In particular, this thesis discusses job and resource management in distributed virtual scientific infrastructures intended for Grid and Cloud computing environments. The main area studied is Grid computing, which is approached using Service-Oriented Computing and Architecture methodology. Thesis contributions discuss both methodology and mechanisms for construction of virtual infrastructures, and address individual problems such as job management, application integration, scheduling job prioritization, and service-based software development.

I addition to scientific publications, this work also makes contributions in the form of software artifacts that demonstrate the concepts discussed. The Grid Job Management Framework (GJMF) abstracts job enactment complexity and provides a range of middleware-agnostic job submission, control, and monitoring interfaces. The FSGrid framework provides a generic model for specification and delegation of resource allocations in virtual organizations, and enacts allocations based on distributed fairshare job prioritization. Mechanisms such as these decouple job and resource management from computational infrastructure systems and facilitate the construction of scalable virtual infrastructures for computational science.

Sammanfattning

Inom beräkningsvetenskap begränsar ofta mängden tillgänglig beräkningskraft både storlek på problem som kan ansättas såväl som kvalitet på lösningar som kan uppnås. Metodik för skalning av beräkningskapacitet till stor skala (dvs större än kapaciteten hos enskilda resurscentras) baseras för närvarande på aggregering och federation av distribuerade beräkningsresurser. Oavsett hur denna resursaggregering tar sig uttryck tenderar skalning av vetenskapliga beräkningar till storskalig nivå att inkludera omformulering av problemställningar och beräkningsstrukturer för att bättre utnyttja problem- och resursparallellism. Effektiv parallellisering och skalning av vetenskapliga beräkningar är svårt och kompliceras ytterligare av faktorer som medföljer resursaggregering, t.ex. heterogeneitet i resursmiljöer och beroenden i programmeringsmodeller och beräkningsmetoder. Detta utbytesförhållande illustrerar komplexiteten i utförande av beräkningar och behovet av mekanismer som erbjuder högre abstraktionsnivåer för hantering av beräkningar i distribuerade beräkningsmiljöer.

Denna avhandling diskuterar design och konstruktion av virtuella beräkningsinfrastrukturer som abstraherar komplexitet i utförande av beräkningar, frikopplar design av beräkningar från utförande av beräkningar samt underlättar storskalig användning av beräkningsresurser för vetenskapliga beräkningar. I synnerhet behandlas jobb- och resurshantering i distribuerade virtuella vetenskapliga infrastrukturer avsedda för *Grid* och *Cloud computing* miljöer. Det huvudsakliga området för avhandlingen är *Grid computing*, vilket adresseras med service-orienterad beräknings- och arkitekturmetodik. Arbetet diskuterar metodik och mekanismer för konstruktion av virtuella beräkningsinfrastrukturer samt gör bidrag inom enskilda områden som jobbhantering, applikationsintegrering, jobbprioritering och service-baserad programvaruutveckling.

Utöver vetenskapliga publikationer bidrar detta arbete också med bidrag i form av programvarusystem som illustrerar de metoder som diskuteras. *The Grid Job Management Framework (GJMF)* abstraherar komplexitet i hantering av beräkningsjobb och erbjuder en uppsättning *middleware*-agnostiska gränssnitt för körning, kontroll och övervakning av beräkningsjobb i distribuerade beräkningsmiljöer. *FSGrid* erbjuder en generisk modell för specifikation och delegering av resurstilldelning i virtuella organisationer och grundar sig på distribuerad rättvisebaserad jobbprioritering. Mekanismer som dessa frikopplar jobb- och resurshantering från fysiska infrastruktursystem samt underlättar konstruktion av skalbara virtuella infrastrukturer för beräkningsvetenskap.

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, 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 II 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 III 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 IV P-O. Östberg and E. Elmroth. GJMF - A Composable Service-Oriented Grid Job Management Framework. Submitted for journal publication, 2010.

Paper V P-O. Östberg and E. Elmroth. Impact of Service Overhead on Service-Oriented Grid Architectures. Submitted for conference publication, 2011.

Paper VI 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 A.C. Elster et al., editors, *Applied Parallel Computing: State of the Art in Scientific Computing, Lecture Notes in Computer Science, vol. 6127*. Springer-Verlag. To appear, 2011.

Paper VII P-O. Östberg, D. Henriksson, and E. Elmroth. Decentralized, Scalable, Grid Fairshare Scheduling (FSGrid). Submitted for journal publication, 2011.

Paper VIII P-O. Östberg and E. Elmroth. Increasing Flexibility and Abstracting Complexity in Service-Based Grid and Cloud Software. In F. Leymann, I. Ivanov, M. van Sinderen, and B. Shishkov, editors, *Proceedings of CLOSER 2011 - International Conference on Cloud Computing and Services Science*. SciTePress. Accepted, 2011.

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, the Swedish Government's strategic research project eSSENCE, and the European Community's Seventh Framework Programme (FP7/2001-2013) under Grant agreement 257115 (OPTIMIS). The author acknowledges the Lawrence Berkeley National Laboratory (LBNL) for supporting the work under U.S. Department of Energy Contract DE-AC02-05CH11231.

In addition to the included papers, material and results from the following publications are included in the thesis.

E. Elmroth and P-O. Östberg. A Composable Service-Oriented Architecture for Middleware-Independent and Interoperable Grid Job Management. *Technical Report UMINF 09.14, Department of Computing Science, Umeå University*, October 2009.

P-O. Östberg. Architectures, Design Methodologies, and Service Composition Techniques for Grid Job and Resource Management. *Licentiate Thesis, Technical Report UMINF 09.15, ISSN 0348-0542, ISBN 978-91-7264-861-6, Department of Computing Science, Umeå University*, October 2009.

M. Jayawardena, C. Nettelblad, S. Toor, P-O. Östberg, E. Elmroth, S. Holmgren. A Grid-Enabled Problem Solving Environment for QTL Analysis in R. In *Proceedings of the 2nd International Conference on Bioinformatics and Computational Biology (BICoB), ISCA, ISBN 978-1-880843-76-5*, pp. 202-209, 2010.

Acknowledgements

A number of people have directly or indirectly contributed to the work of 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 his patience, enthusiasm, and the positive environment he creates in our research group. On the same note, I would also like to thank my co-advisor Bo Kågström for inspiring discussions, and the unique perspective he brings to them. Among my (current and former) colleagues in our research 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 (in no particular order) Francisco Hernández, Daniel Henriksson, Petter Svärd, Wubin Li, Ahmed Ali-Eldin, Lei Xu, Ewnetu Bayuh Lakew, Mina Sedaghat, Tomas Ögren, Mikael Öhman, Sebastian Gröhn, Raphaela Bieber-Bardt, Arvid Norberg, Peter Gardfjäll, Lennart Edblom, Christina Igasto, and Markus Karlsson for all their contributions to our collective effort. Among our research partners I would like to thank Sverker Holmgren, Jonas Lindemann, Salman Toor, Mahen Jayawardena, Carl Nettelblad, and Lavanya Ramakrishnan for interesting collaborations, and the support staff of the High-Performance Computing Center North (HPC2N) for their contributions and knowledge of the research systems used. At the Lawrence Berkeley National Laboratory, I would like to thank my hosts, Deb Agarwal and Horst Simon, as well as Keith Jackson, Dan Gunter, Tahgrid Samak, and Keith Beattie for unique experiences and interesting discussions. For valuable discussions and constructive feedback on my licentiate thesis, I would also like to thank Vladimir Vlassov and Frank Drewes. Finally, on a personal level, I would like to thank my family and friends for all the love and support they provide. None of this would be possible without you.

Thank you all.

Contents

Chapter 1

Introduction

This work addresses software, architecture, and infrastructure design for virtual scientific computational environments. In particular, it discusses construction of virtual infrastructures that are capable of scaling computations to global scale while abstracting complexity of computation enactment. Contributions are made to individual problems such as job and resource management, fairshare-based job prioritization, and software development methodology.

Scientific computation is a broad field with applications in many domains of science and industrial settings. Evolution of scientific computation methodology is fast and construction of infrastructures for computational science lies at the front edge of technical development. Scaling of scientific computations typically involve parallelization of problems and mapping of computational tasks to aggregated resource sets. At large scale, a number of factors complicate efficient enactment of scientific computations. For example, coupling of computational applications to specific types of resource sets and heterogeneity in computation methodologies severely impact the complexity of computation enactment.

The approach taken in this work addresses these issues through design and implementation of abstractive virtual infrastructures for distributed computational science. The virtual infrastructures discussed decouple computational applications from computation enactment, abstract complexity introduced by heterogeneity in computation methodology, and facilitate large-scale use of aggregated computational capacity. The vision of this work details infrastructures that provide transparency in resource utilization, scalability in computational capability, and flexibility in usage scenarios and system applicability. The main area addressed is Grid computing infrastructure design and application integration, which is approached using Service-Oriented Computing and Architecture methodology. Thesis contributions include scientific publications addressing, e.g., virtual infrastructure job and resource management, fairshare-based resource capacity allocation enactment, and software and architecture design for virtual scientific computational infrastructures, as well as software artifacts intended for construction of virtual computational science infrastructures.

The rest of this thesis is structured as follows. Section 2 provides a brief introduction to enactment of scientific computations and a discussion of the relationship between heterogeneity and coupling of computational methodology and the complexity of computation enactment. Section 3 discusses Grid computing, an approach to construction of virtual infrastructures for computational science based on aggregation and federation of computational resources into large virtual computational environments. Section 4 details Cloud computing, a paradigm for construction of virtual computational infrastructure based on hardware-enabled paravirtualization technologies and exposure of computational capacity as services in stratified service models. Section 5 gives an introduction to software development for virtual infrastructures and discusses methodology for distributed and service-oriented software development in Grid and Cloud environments. Section 6 summarizes the contributions of the thesis, relates thesis contributions to the field, and presents the thesis papers.

Chapter 2

Enactment of Scientific Computations

Efficient enactment of scientific computations is a widely studied and complex field. Due to potential performance gains, computational applications are often optimized for specific resource environments, or utilize computational frameworks that make extensive assumptions about computational resource organization and characteristics. In general, such assumptions introduce a degree of coupling between computational applications and infrastructure that complicates migration of computational applications between environments and reduce the general applicability of computational methodology. To illustrate some of the complexity of enactment of scientific computations, and motivate the use of virtual infrastructures for computational science, here follows a brief introduction to some current paradigms for scientific computation and a discussion of the relationships between concepts such as computation coupling and heterogeneity and the complexity of enactment of computations.

2.1 Heterogeneity in Scientific Computations

Scientific computations employ computer-based resources to quantify, analyze, simulate, and in other ways investigate phenomena that can be modeled computationally. Use of computation as a tool influences development of not only traditional computational sciences and engineering, but finds application in fields as diverse as, e.g., financial modeling, geographical mapping, social studies, meteorology, and life sciences. In addition to computation, computer-based resources are highly utilized to manage, store, and analyze datasets that result from scientific experiments, observations, and simulations. Data mining and analysis techniques are used to extract knowledge from data, and provide a foundation for scientific experimentation even in non-traditional computational fields. Scientific computation is a broad field with applications in most

established sciences, and computation methodology and resources provide tools applicable to a wide variety of scientific and industrial problems.

In computational science, the scale of problems addressed and the resolution of solutions achieved tend to be limited by the available computational capacity. Access to faster and more powerful computational resources enable new methodologies, facilitate new forms of scientific collaborations, and allow scientists to address larger and more complex problems. As the cost of constructing (individual) computer systems increases dramatically beyond the capabilities of commercially viable mass produced high-end systems, computational capacity is typically scaled up through aggregation of multiple (possibly distributed) resource systems. Regardless of whether this aggregation manifests as physical compute clusters, federations of resource sites, or virtual collaborations over the Internet, scaling of scientific computational problems typically involves (re)formulation of computational structures and problems to exploit problem parallelism and mapping of computations onto parallel computational resources. Aggregation of computational resource capacity introduces hardware and software heterogeneity in computational resource sets, as well as heterogeneity in methodologies for computation specification and enactment.

Due to its broad applicability, scientific computation is a field with extensive heterogeneity in methodology, approaches, and tools. Problems addressed range in scope from design of nano-materials and simulation of molecule interactions to long-term analysis of global climate changes and metric expansion of space, and are naturally addressed using domain-specific methodologies. To properly address complex problems and increase knowledge yields from large experiments, scientific computations are often performed in interdisciplinary collaborations. Projects that bring together scientists from different fields, with varying traditions of computing and levels of computing experience, tend to introduce further heterogeneity in computation methodology and tool requirements. Heterogeneity in computation methodology is dynamic and evolves with development of computational models, availability of computational tools, and formulation of new problems and interdisciplinary collaborations.

2.2 Paradigms for Computation Enactment

A number of paradigms for scientific computations that each have unique requirements for computation enactment exist. Variation in approaches arises from factors such as heterogeneity in problem and solution requirements, applicability of traditional methodology, and availability and functionality of tools. Common to all these paradigms is the exploitation of problem parallelism to increase computation scalability. Scalability within parallel scientific computations can be measured in multiple ways, but is typically quantified using metrics that express the proportions of overhead or speedup introduced by parallelization relative to the number of computational resources used and a corresponding sequential solution [122].

Traditional definition of computing infrastructure details computational resources and systems native to computational resources. In this work the definition of (virtual) computing infrastructure is extended to include middlewares and other software frameworks that provide abstraction layers to facilitate virtualization of computation and increase transparency in resource utilization. The term virtual infrastructure is here used to refer to systems that collectively are used to enact utilization of computational resources, typically through abstraction and virtualization of computation enactment, data management, and computational resources.

Evolution of methodology for computation is affected by a number of factors ranging from introduction of new technology such as multi- and many-core Central Processing Units (CPUs) to paradigm shifts such as the advents of Grid and Cloud computing. Design of infrastructures for computational science and computational methodology evolve together and influence each other. This work studies construction of virtual infrastructures for computational science that aim to abstract enactment of computation and allow computational methodologies to evolve within computational paradigms without limitations imposed by complexity in computation enactment.

To illustrate some of the challenges in enactment of scientific computations, here follows a brief overview of current computational paradigms and (concrete and virtual) infrastructures for scientific computation. The definition of computation paradigm used here details methodology for computation enactment, and application- and system-level computation behavior rather than algorithms and methods for actual computations. With this definition, classification of computational paradigms can be performed by observation of facets of paradigm methodology, for example how separation and coordination of computations are expressed (e.g., explicit or implicit control flows), what synchronization requirements are imposed on parallel computations (ranging from fine-grained explicit synchronization to embarrassingly parallel problems), or how computations are steered and visualized (e.g., batch or interactive applications). Characterization of scientific computations can be done by paradigm or by computational and data requirements and topology, behavior of computational applications, or the type of computational infrastructure used for computation enactment.

2.2.1 High-Performance Computing (HPC)

High-Performance Computing (HPC) [49] is a paradigm for utilization of high-end computational resources for addressing and solving large computational problems as quickly as possible. HPC computations typically consist of tightly coupled, synchronized, parallel, and highly optimized code running on dedicated high-end systems. Considered traditional supercomputing, HPC applications include engineering problems, large-scale mathematical problems, physics and chemistry simulations, and other similar problems that are well understood and where performance gain can be achieved by mapping compu-

tations to homogeneous high-end resource sets. HPC applications are typically designed for high resource utilization and maximization of computational efficiency, make strict assumptions about homogeneity in resource capabilities and interconnects, and aim to achieve high performance through fine-grained synchronization using communication frameworks and programming models such as Message Passing Interface (MPI) [177] mechanisms.

Traditional HPC deployments organize (predominantly) homogeneous sets of high-end computational resources dedicated full-time to computation. Today, HPC resources are normally organized in clusters of network-accessible servers and administrated and operated in computing centers. HPC resources typically employ low latency, high bandwidth interconnects and distribute program executions (jobs) to resources through batch systems and centralized schedulers. As resource site schedulers have complete information about queued, pending, and running jobs, and continuously updated views of the state and availability of computational resources, HPC scheduling environments can enact advanced job requirements and achieve high resource utilization rates through, e.g., backfilling [178] and fairshare [114] job prioritization techniques.

In the HPC model, end-users can make extensive assumptions about the homogeneity and capacity of the computational resources, as well as about the system environment of the resources. End-users are expected to provide executables compatible with resource hardware and software environments, and submit jobs through batch systems or portals. Data files are accessed through shared file systems or staged in and out of the execution environment, and storage requirements for data files and maximal runtime limits for computations are often expected to be quantified by end-users in advance.

For end-users, the HPC computational model requires extensive knowledge about resource environments. HPC system complexity may impact end-user productivity directly or indirectly through, e.g., batch queue latencies complicating error diagnostics and development of computational applications, or by requiring users with resource allocations on multiple resource sites to maintain concurrent compatibilities with multiple environments. Efficient use of HPC computational resources requires high end-user maturity in computational methodology as well as advanced programming experience.

2.2.2 High-Throughput Computing (HTC)

High-Throughput Computing (HTC) [128] is a computational paradigm that defines computational efficiency in terms of throughput over time, i.e. without imposition of strict deadlines for computations. HTC applications typically break computational problems into large numbers of small, loosely coupled subproblems without interdependencies. As such, HTC computational tasks can be processed in parallel, and make low or no assumptions about synchronization of computations or the computational capacity of resource sets. HTC computations tend to be separated in time, and time scales for measuring computational efficiency in HTC applications may range over long periods of time,

e.g., weeks or months. HTC applications are often deployed on distributed and dynamical resource sets and are concerned with resilience in addition to efficiency in computation.

HTC infrastructures encompass distributed systems where coordination nodes distribute computational tasks to autonomous clients running on dynamic resources. Task distribution is often fully abstracted to end-users, which do not know where their computations are performed, and have limited influence over resource selection or the computational environment of the resources. HTC computational resource sets typically exhibit high heterogeneity and consist (at least partially) of non-dedicated (shared) resources that run computations in the background at low system priority or during machine idle phases. In HTC systems, resource availability may vary (resources join or leave resource sets dynamically), and computations may be preempted or deprioritized at any time. Resource volatility is a major factor in HTC systems and infrastructures typically provide computational resilience through fault recovery mechanisms such as checkpointing [166] and redundancy in computation executions.

As HTC computations may be performed on volatile, non-dedicated resources, HTC tasks tend to be self-contained, limited in scope, have low system requirements, and require little or no synchronization. Computational tasks mapped onto HTC infrastructures are typically embarrassingly parallel and consist of large numbers of autonomous computations that can be processed in arbitrary order. Computational tasks are matched to resources dynamically e.g., using the classad [165] technique of Condor [183], and may employ multiple parallel executions to compensate for erroneous results from non-trusted resources (a technique used, e.g., by BOINC [15]).

Characterizations of different kinds of HTC infrastructures can be made on, e.g., the level of trust placed on resources (desktop Grids [119] versus volunteer computing [171]), whether special purpose hardware is employed (distributed.net [1]), or the generality of the software framework used to distribute computations (ranging for specialized applications such as SETI at home [16] to generalized frameworks such as BOINC [15], Entropia [38], and Condor [183]). Data management in HTC applications typically follow patterns similar to parameter sweeps, where data sets are pre-processed and segmented into smaller blocks for individual processing, and results are assembled, post-processed, and aggregated in synchronization steps. As HTC computations typically are separated in time, HTC computation data may require more storage capacity per time unit than other types of computations.

2.2.3 Many-Task Computing (MTC)

Many-Task Computing (MTC) [162] is recently introduced hybrid paradigm that aims to bridge the gap between the HPC and HTC paradigms. MTC applications are concerned with computational throughput and efficiency over limited time frames and typically run large numbers of short-lived computations. MTC applications behave as HTC applications, but employ application-specific

performance metrics measuring computational efficiency over short time periods. As MTC computations tend to be short lived, enactment tasks such as file staging and I/O operations impose a larger relative proportion of overhead, which imposes challenges on MTC applications and systems to maintain high computational efficiency. MTC computational efficiency is concerned with aggregated computational throughput for large sets of short computations with limitations in response times. MTC computations are typically realized on computational resource sets similar to HPC deployments retrofitted with special computation enactment frameworks such as Falkon [163] and Kestrel [181].

2.2.4 Peer-to-Peer Computing (P2P)

The Peer-to-Peer Computing (P2P) [147] paradigm reverses the traditionally hierarchical communication model of client-server systems and focuses on construction of decentralized loosely coupled networks of distributed components (nodes). P2P systems share computational resources, capacity, or data for distributed applications, and are organized in abstract overlay networks using decentralized organization structures such as distributed hash tables. P2P nodes may fulfill any communication or computation role over the lifespan of an application and are typically equally privileged within a network. As they build on dynamical participation of peers, P2P systems are generally designed for redundancy and resilience in, e.g., computation enactment or data replication. For scientific computation, P2P systems may be utilized to share computational capacity of resources, or to distribute, store, and replicate data.

P2P infrastructures are typically constituted by dynamically assembled nodes that form large, highly parallel virtual systems with high throughput for data transfers. Current visible examples of applications of P2P techniques include file sharing networks and frameworks such as BitTorrent [159], Gnutella [169], and OneSwarm [106]. P2P infrastructures can be categorized based on infrastructure topology, e.g., by observation of how data distribution models exploit parallelism [33], or by classification of coordination models. P2P techniques can be used in virtual infrastructures for computational science to e.g., distribute data or facilitate dynamic resource discovery [96, 192]. Further characterization and analysis of P2P systems is available in [111, 136].

The benefits of P2P communication systems lie in that system-wide coordination can be distributed, replicated, and segmented (e.g., each data set gets a unique torrent in BitTorrent). Coordination in P2P systems is computationally cheap and can be performed by virtually any node in the system, and coordination network requirements are low, allowing nodes with great variation in link quality to participate in networks. P2P networks in general form resilient and scalable virtual infrastructures well suited to be mapped onto distributed, heterogeneous networks such as the Internet. The general applicability of P2P infrastructures for computational science is however low, and application of P2P networks tend to be limited to data distribution applications with low security requirements.

2.3 Coupling and Enactment Complexity

As illustrated by the contrast between the HPC and HTC paradigms, behavior and requirements of scientific computations vary greatly. As further illustrated by the MTC paradigm, a range of hybrid approaches that are difficult to crisply categorize within established computational paradigms exist and evolve with application requirements and the development of computational environments.

In general, there exists a correlation between the degree of coupling of computations, and the complexity of enactment of computations. Tightly coupled computations like, e.g., HPC MPI applications tend to make extensive assumptions about synchronization, capabilities, and environments of computational resources that affect scheduling parameters and resource requirements. Loosely coupled computations and applications, that have low (if any) synchronization requirements and can be run on arbitrary hosts, are better suited for enactment on virtual infrastructures where resources are discovered and utilized on demand. The impact of computation coupling on enactment and development complexity is best illustrated by applications residing at the ends of this spectrum; special purpose hardware approaches and abstract workflows.

2.3.1 Special Purpose Hardware Approaches

As certain computationally expensive operations may be performed more efficiently and cost-effectively on special purpose hardware platforms, a pattern of utilizing dedicated special purpose hardware has long existed within computational science. Due to high acquisition and operation costs, special purpose hardware systems have historically been limited to computing centers and use of special purpose hardware considered part of the HPC paradigm. Recently however, with the introduction of customizable special purpose hardware solutions like Field-Programmable Gate Arrays (FPGAs) [191], and commercially available mass produced technologies suitable for scientific computation such as Graphics Processing Unit (GPU) systems [67], and Cell processor [98] based entertainment systems, special purpose hardware systems are now more easily attainable and find new applications within computational science.

The applicability of special purpose hardware for computations is often limited by the nature of the problem, e.g., the inherent scalability and granularity of the computational task [37]. For example, in GPU computing, problems are segmented into computational tasks that run in parallel threads, which are mapped to execution pipelines on the graphics card using specialized languages and APIs such as CUDA [142] and GPGPU [129]. With GPUs, computational speedup can be achieved as long as frequent task synchronization can be avoided [37]. The GPU approach is comparable to MPI programming in that computational threads are mapped to the graphics pipeline in a way similar to how jobs are matched to computational resource nodes in MPI. Tools for conversion of synchronized parallel programming models to special purpose hardware solutions are emerging and include, e.g., OpenMP [44] code conver-

sion to GPGPU thread models [125]. In FPGA computing, a reconfigurable circuit is compiled and configured to match computation patterns by specialized compilers in a pre-computation step. FPGA computations are in general more efficient than traditional CPUs when sequences of large numbers of fine-grained computational steps are to be processed [37], as is often the case in, e.g., encryption algorithms.

Computationally, special purpose hardware systems can be highly efficient, but are often more expensive to acquire, administrate, and maintain compared to general purpose systems. Compared to development for standard CPU systems, special purpose hardware systems introduce new programming models and developing or adapting applications for special purpose hardware systems are complex tasks that often require specialized programming constructs. Seen from a programming model point of view, the current trend of increasing computational capacity in CPUs through provisioning of more cores per CPU can be seen as a type of special purpose hardware approach. Programming models for multi- and many-core systems currently utilize similar thread-based constructs as single-core CPU models. More advanced concurrent programming constructs, e.g., transactional memory [97], are expected to be required to increase programmer productivity in these kinds of environments.

2.3.2 Workflows and Parameter Sweeps

In contrast to special purpose hardware approaches that make extensive assumptions about computational resource environments and achieve efficiency by optimizing code for specific architectures, there also exist programming models for scientific computation that abstract knowledge of computation enactment environments and decouple computation specification from computation enactment. Parameter sweep and workflow applications focus on coordination of (potentially large) sets of computational tasks and (virtualized) enactment of computations. Computational tasks within parameter sweeps and workflows may be mapped to any suitable computational infrastructure without affecting the structure of the application.

Workflow systems [99] provide meta-language constructs for organizing sequences of computational steps that form applications and provide a high level of abstraction of computation enactment. In computational science, workflows are typically used to detail structured enactments of sets of sequential or parallel computation tasks. For example, a workflow may detail tasks such as staging and preprocessing of data, computation, staging and post-processing of results, and aggregation and statistical analysis of result sets. Workflows can be further sub-categorized as static or dynamic, control flow or data flow oriented, etc., and are often modeled using topological constructs such as directed acyclic graphs. Parameter sweeps are a specific form of scientific workflows where simulations, experiments, or tests are repeated multiple times using different permutations of parameter sets. Computation results may be analyzed during runtime and influence the processing of the parameter sweep, or in separate

post-processing steps. Computational tasks in parameter sweep applications are typically embarrassingly parallel, i.e experiment sets using different parameter sets are fully independent of each other, and computations may be mapped onto computational infrastructures in arbitrary order.

2.4 Virtual Infrastructures for Computational Science

Tightly coupled computational methodologies that make extensive assumptions about computational environments and computation synchronization find legitimate use through increased computational efficiency. However, as illustrated by scientific workflows and parameter sweeps, there exists large classes of computational applications that for reasons of complexity and computational topology are not efficient to map to computational environments explicitly. In particular, when scaling computations to global scale (e.g., to use computational resources from multiple resource sites), there exists an inversely proportional relationship between the degree of coupling and the scalability of an application. In general, assumptions about computation environments or synchronization tend to increase computation enactment complexity and introduce significant development overhead.

For reasons of computational efficiency, computational methodologies are often designed for both problems addressed and the types of computational infrastructures available. As scaling of scientific computations typically involves organizational collaborations and aggregation of computational capacity from multiple resource sets, multiple types of heterogeneity are introduced in computation enactment. As complexity of computation enactment increases as a function of multidimensional heterogeneity, large scale resource use necessitates the use of abstractive virtual infrastructures that isolate end-users from computation enactment and provide scalability in aggregation and federation of resource sets. The relationships between computation coupling and heterogeneity to the complexity of computation enactment demonstrate that abstractive and loosely coupled models for computations are more easily scaled to large scale resource use in virtual environments.

The approach of this work details construction of virtual scientific infrastructures that abstract computation enactment complexity and allow for greater flexibility in scientific computation methodology through decoupling of computation design and enactment. Virtual infrastructures for computational science promote computation scalability through facilitation of (automated) large scale use of aggregated computational capacity. The remainder of this thesis discusses paradigms and methodology for construction and application of virtual scientific computational infrastructures.

Chapter 3

Grid Computing

Grid computing [76] is a computational paradigm describing aggregation and federation of distributed resource sets from multiple administrational domains to form virtual high-performing computational systems. For applications, the Grid computing paradigm is primarily concerned with transparency in resource utilization. Applications employ middlewares, brokers, and information systems to decouple end-users from computation enactment tasks and allow discovery and utilization of computational resources on demand. Grids organize user bases in Virtual Organizations (VOs) [81] that are mapped to virtual computational infrastructures using distributed collaborative security models.

In the last 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., [74, 80, 117, 127, 190], and while the scientific community has reached a certain level of agreement on what a Grid is [180], 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 is concerned with service availability, performance scalability, virtualization of services and resources, and resource (access) transparency [82, 180]. 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 [80]. 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 Grid systems 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 (asynchronized) parallelization. Motivations for migration to Grid environments vary, but often include envisioned performance benefits, synergetic collaboration effects, and facilitation of large-scale resource utilization.

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.

3.1 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 [80]. 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 [25], and very large-scale simulation and analysis applications such as the Astrophysics Simulation Collaboratory [170].
- data intensive, e.g., experimental data analysis projects such as the European Data Grid [173], and image and sensor analysis applications such as SETI@home [16].
- distributed collaboration efforts, e.g., online instrumentation tools such as ROADnet [94], segmented simulation and analysis efforts such as Folding@home [123], or remote visualization projects such as the Virtual Observatory [201].

Based on computational requirements and topology of applications, computational Grid applications can broadly be classified as HPC, HTC, or hybrid approaches. Grid 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. HPC application computations are typically structured to maximize application computational efficiency for a particular problem, e.g., through MPI frameworks. Analysis of

workload traces for computational Grids environments [103] show that Grid workload patterns significantly differs from traditional HPC workloads [68] even when running on HPC resources [101, 102]. The predominant form of computation in HPC Grids is the bag-of-tasks model, where large numbers of computational tasks are performed more or less sequentially [104].

Grid 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 MTC 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 [162].

Differentiation of Grid HPC, HTC, and MTC applications from corresponding non-Grid applications can be done primarily on the use of dynamic resource discovery, brokering, and Grid data staging techniques in computation enactment. 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 of 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 and resource 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 systems are typically constructed through aggregation and federation of computational resources, allowing Grids to exhibit very high levels of system availability despite system capacity varying over time. 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. Computational Grids focus on providing abstracted views of computational resource access, and address very large computational problems. Data Grids focus on providing virtualization of data storage capabilities and provide non-trivial and scalable qualities of service for management of very large data sets. The work of this thesis is 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 [80], and technology advances have 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 have achieved much in terms of system utilization, scalability and performance, much work on reduction of system complexity and increased system usability still remains [180].

3.2 Infrastructure

The name Grid computing originates from an analogy in the initial vision of the field; to provide access to computational resource capabilities in a way similar to how power grids provide electricity [80], 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 Grid application and user requirements 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 infrastructure

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 scientific applications and systems [57, 59].

From a systems perspective, the Grid computing paradigm addresses concepts such as performance scalability, resource virtualization, and access transparency [82]. 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 application 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 abstracted by middlewares and native resource access layers. 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 networks, virtualization through interface abstraction mechanisms, and transparency through automation of core Grid component tasks (such as resource discovery, resource brokering, file staging, job submission and monitoring, etc.).

To facilitate flexibility in resource usage models, Grid users and resource allotments are typically organized in Virtual Organizations (VOs) [81]. 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 factors such as resource availability, sharing conditions, and organizational structure and 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 aggregation of resource capacity across ownership domains, and provide a way to provision resource usage without pre-existing trust relationships between resource owners and individual Grid users.

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

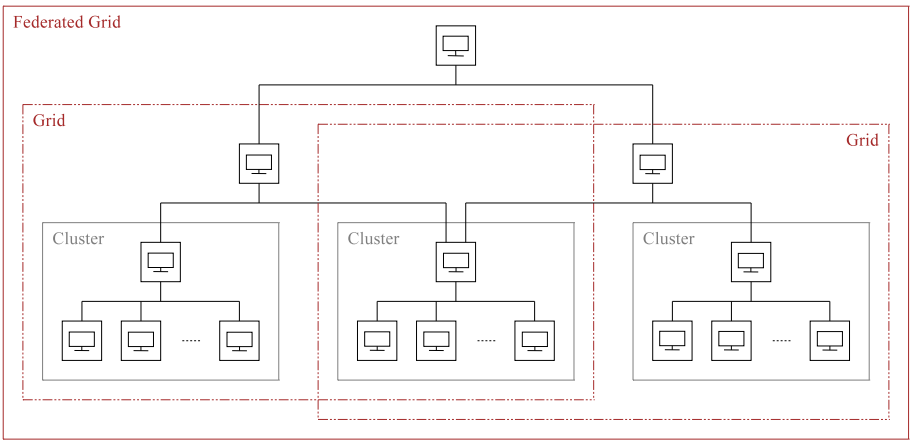


Figure 1: A naive Grid model. Grids aggregate clusters of computational resources to form larger, virtual systems. Resources may be part of multiple Grids, and federated Grids are composed of collaborative federations of Grids.

3.3 Job and Resource Management

A key functionality set of any Grid infrastructure is job and resource management, a term here used to collectively refer to a set of processes and issues related to execution of programs on computational resources in distributed virtual environments. This includes, e.g., management, monitoring, and brokering of computational resources; description, submission, and monitoring of jobs; fairshare scheduling 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. The approach taken in this work is to primarily regard components as parts of systems, and focus on component interoperability to promote system composition flexibility [59]. The focus of the job and resource management contributions presented here is to abstract system complexity and heterogeneity, and provide transparent resource utilization models that do not couple applications to specific Grids or Grid middlewares [149].

3.3.1 Aggregation and Federation of Resources

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 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, national, 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 resources and systems.

A core requirement in Grid systems is that resource owners at all levels retain full administrative control over their respective resources. This Grid characteristic, to be devoid of centralized control [74], is a design trait aimed to promote scalability in design and implementation of Grids, and imposes a number of cross-border administration 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) [134] and certificates [4]. Grid capacity allocations are typically specified at VO level and mapped to site allocations in resource site systems.

3.3.2 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 [196] 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 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 Services Resource Framework (WSRF) [77] specifications address Web Service state management issues, and contain interface definitions and notification mechanisms that may be used 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.

3.3.3 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 [51].
- batch systems and scheduling mechanisms, e.g., PBS [24] and Maui [185].
- job and resource monitoring tools, e.g., GridLab Mercury Monitor [20].
- computation frameworks, e.g., MPI [177] toolkits.

Grid HTC resources are of more varied nature. CPU-cycle scavenging schemes such as Condor [183] for example typically utilize standard desktop machines, while volunteer computing efforts such as distributed.net [1] may see use of any type of computational resource provided by end-users. HTC Grids often deploy software that can be considered part of Grid middlewares on computational resources, e.g., Condor and BOINC [15] 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, abstract native resource 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 [52], Globus [87], UNICORE [182], LCG/gLite [35], 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 control.
- authentication and authorization of users.

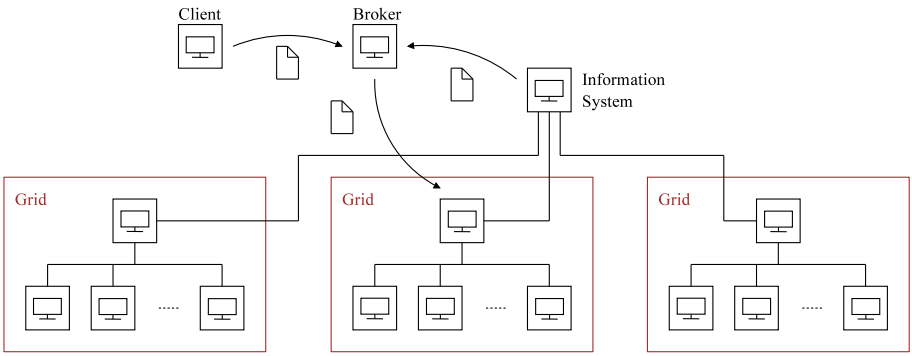


Figure 2: Grid resource brokering. Autonomous resource brokers act on behalf of clients and dynamically discover and match jobs to computational resources using job descriptions and resource state information from information systems.

Additionally, middlewares and related systems can incorporate solutions for advanced functionality such as resource brokering [183], accounting [85], and Grid-wide load balancing [35].

While the original motivations for construction of Grids included addressing 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 [62]; 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. The approach of this work is to address this level of interoperability issues through abstraction and standardization. The Grid Job Management Framework (GJMF) [149] for example provides a range of middleware agnostic job management interfaces that abstract middleware interoperability issues and build on standardized formats for job descriptions, resource discovery, and job control.

3.3.4 Resource Brokering

A fundamental task in Grid job management is resource brokering; matching of jobs to computational resource(s) suitable for job execution. As illustrated in Figure 2, resource brokers typically 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 [75], ARC XRSL [52], as well as standardization efforts such as the Job Submission Description Language (JSDL) [17].

Typical information specified by job descriptions include

- 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 [62].

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 [63]. In these settings, Grid brokers and schedulers need to adapt to their environments and design emphasis should be placed on coexistence [149]. 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.

3.3.5 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 task sequence.

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 transferred 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 transferred 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, the 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 remote 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, clients must be provided failure context information. 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 results in information leakage. Typically, Grids make use of advanced security features that make failure management, administration, and direct access to resource systems complicated.

3.3.6 Job Management

Advanced high-level Grid applications require job management functionality beyond generic resource brokering and job control capabilities. For example, efficient mechanisms for monitoring and workflow-based scheduling of jobs are required to facilitate management of large sets of jobs.

Conceptually, there exist two basic types of Grid job monitoring mechanisms; pull-based and push-based mechanisms. 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 sometimes be supplemented with pull model mechanisms [149].

Push notifications can also be extended to notification brokering scenarios, and be incorporated in notification brokering schemes based on Message-Oriented Middleware (MOM) [22] or Enterprise Service Bus (ESB) [36] frameworks. The WS-Notification specification [90] details interfaces for push model status notifications that may be used 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 [139], 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) [11], 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 [145] and Pegasus [47], and Grid infrastructures to automate execution of job sets. Important questions here include abstraction of level of detail, and balancing of automation against level of control for advanced job management systems [56].

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 do not require end-users or administrators to replace entire system components, but rather offer flexible configuration and code injection mechanisms [59, 149].

3.3.7 Usage Allocation Enforcement

Grid usage allocations are typically specified on Virtual Organization level and distributed using academic grant schemes. Usage allocations can take the form of abstract usage credits, be mapped to resource specific metrics such as CPU hours, and be enforced using dynamic quota models on resource sites. Grid usage is typically tracked using accounting systems such as the SweGrid Accounting System (SGAS) [85], or through monitoring of workload trace logs.

Fairshare scheduling systems (as defined by [114]) perform dynamic load balancing in systems by live comparison of usage consumption to usage allotment preallocations. Fairshare load balancing is achieved by allowing usage ratios to influence scheduling decisions, in effect creating priority queue systems that rank jobs after owner usage consumption ratios. As fairshare mechanisms using this model require access to complete views of both usage allocation and usage consumption data, fairshare tend to be enforced locally on resource systems (in schedulers such as Maui [185] and SLURM [204]) rather than on abstract Virtual Organization level. The FSGrid [151] system extends the local fairshare scheduling model to Grid level and defines a decentralized system for global enactment of fairshare allocation through local computations on distributed data.

3.4 (Non-Intrusive) Interoperability

A large portion of Grid infrastructure operation builds on automation of Grid functionality tasks. Automation in Grid environments is achieved through Grid component and system collaboration, and thus requires 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 [82], e.g., the Extensible Markup Language (XML) [29] and Grid Web Services [84]. To promote non-intrusive interoperability in Grid system design, many Grid systems are realized as Service-Oriented Architectures [144].

Grid standardization efforts have proposed interfaces for many interoperability systems ranging from job description formats, e.g., JSDL [17], job submission and control interfaces, e.g., OGSA BES [78], to resource discovery, e.g., OGSA RSS [79], and Cloud computing interfaces, but broad consensus on best practices for Grid application and infrastructure construction is yet to be reached. Further treatment of Grid application and infrastructure integration issues is available in [58, 108].

Chapter 4

Cloud Computing

Cloud computing [18] is a paradigm that extends the virtualization approach of Grid computing and focuses on virtualization of infrastructures, platforms, and applications through isolation of job executions in virtual machines run on consolidated servers. While Grid computing provides scalability through federation and aggregation, Cloud computing provides elasticity through abstraction and virtualization. Cloud systems enable a resource utilization model where end-users are fully decoupled from exact knowledge of where, how, and on what physical resources computations are enacted. Cloud applications are not limited to the batch processing model common to other paradigms, and may be supplied custom execution environments through virtual machines.

The Cloud computing paradigm evolved from the convergence of a number of software development and infrastructure construction trends including, e.g., autonomic computing [116], Grid computing [76], utility computing [30], virtualization, and service-oriented software development [175]. Technologically, development of Cloud computing was facilitated by the development of efficient hardware paravirtualization technology and hypervisors (virtual machine monitors) such as Xen [23], VMWare [198], and KVM [118],

In Cloud computing, computational resources, platforms, applications, and infrastructures are abstracted and virtualized, and offered as services with metered cost models [205, 18]. What primarily distinguishes Cloud computing environments from other environments is that software are run as services with unknown lifecycles and that the (virtual) infrastructure itself can be scaled dynamically to meet varying capacity requirements with minimal manual efforts from administrators. The name Cloud computing stems from a metaphor for the Internet, which is often depicted as a network cloud in network diagrams, and references the virtualization properties of Cloud computing systems.

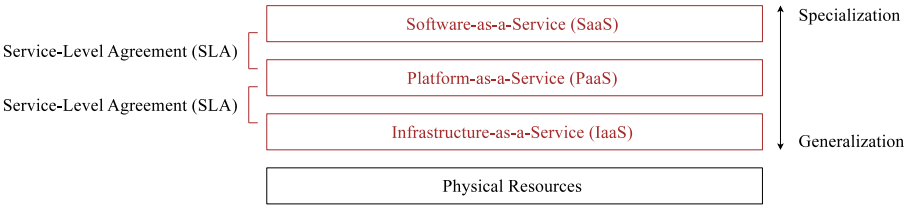


Figure 3: Cloud computing service models. Cloud environments virtualize access to, and realization of, infrastructures, platforms, and applications (software), and offer computational capacity through metered service models. Provisioning of Cloud services is regulated in Service-Level Agreements.

4.1 Service Models

Cloud computing environments offer multiple levels of virtualization, and services offered by Clouds are often categorized in stratified service models as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) services (illustrated in Figure 3).

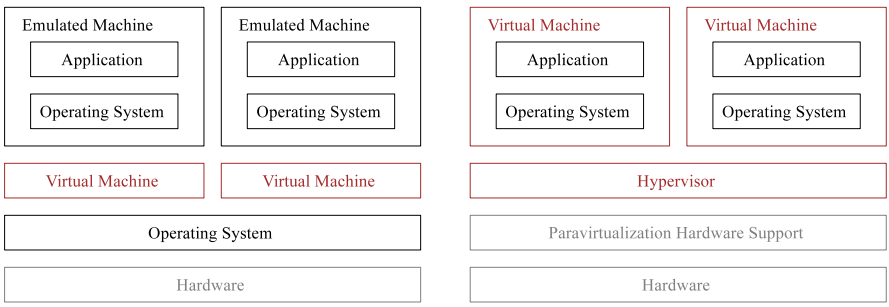
4.1.1 Infrastructure-as-a-Service (IaaS)

Cloud Infrastructure-as-a-Service (IaaS) service models refer to provisioning of dynamic infrastructure capabilities as virtualized environments offered as metered services for agreed-upon time periods [115]. IaaS services are typically accessed through abstractive Cloud interfaces that expose APIs for instantiation of virtual servers and coordination of customized infrastructure environments.

Through resource elasticity and server consolidation techniques, Cloud infrastructures are able to rapidly adapt to increased (or decreased) requirements for computational capacity, and provide on-demand access to computational resources using pay-per-use accounting models. Use of virtualized Cloud infrastructures facilitates management of expected and unexpected peaks in computational capacity requirements, adaptation to regional variations in usage patterns, and minimization of costs for (over)provisioning of hardware.

Typically, Cloud infrastructures utilize gateway mechanisms that accept or reject requests for instantiation of services based on internal models for infrastructure load and capacity. Conditions for provisioning of services in Clouds may be specified and regulated using Service-Level Agreements (SLAs).

IaaS service models have found widespread acceptance in industry, where providers offer access to virtualized infrastructure services in public Clouds environments such as the Amazon Elastic Compute Cloud (EC2) [12] and Simple Storage Service (S3) [14], and Microsoft Azure [186]. For construction of private Cloud infrastructures, a number of open source Cloud toolkits exist, e.g., Nimbus [140], Eucalyptus [141], and OpenNebula [72].



(a) Traditional virtualization. (b) Hypervisor paravirtualization.

Figure 4: Traditional virtualization versus hypervisor paravirtualization. Traditional virtualization employs virtual machines to emulate computational resources on top of operating systems. Hardware-enabled paravirtualization employs hypervisors to virtualize computational resources and encapsulate program execution environments in virtual machines.

4.1.2 Platform-as-a-Service (PaaS)

Platform-as-a-Service (PaaS) service models refer to provisioning of computational platforms as services in Cloud environments. Utilization of PaaS services allows developers and end-users to, e.g., customize software environments, use metered cost models for software acquisition, and adapt to multi-tenant architecture requirements. Support for machine-to-machine communication mechanisms, such as Web Services, allow exposure of PaaS capabilities in integration of heterogeneous software environments.

4.1.3 Software-as-a-Service (SaaS)

Software-as-a-Service (SaaS) is a service model where access to software packages are offered as services through metered subscription or pay-per-use cost models. SaaS allows end-users to avoid rigid license agreements and fixed cost models and is driven by multiple use cases in economical settings such as Cloud computing where platforms are virtualized, network capacity is growing more rapidly than compute capability, and computing is a tradeable commodity.

4.2 Virtualization

As indicated by the *aaS service models used to describe Cloud capabilities, Cloud environments provide multiple levels of virtualization. The foundational virtualization capability of Clouds lies in that Clouds utilize paravirtualization technology and hypervisors to enact virtual machines rather than execute binary software components (illustrated in Figure 4). Enactment of virtual machines virtualizes and isolates programs from computational environments

(operating systems, libraries, and software stacks) as well as operating systems from computational resources (hardware platforms).

Isolation of operating systems from hardware platforms extends the capabilities of software execution environments. Through service consolidation techniques, multiple systems that do not operate at full hardware capacity may share the same hardware resources and allow infrastructures to improve resource utilization rates and reduce size, hardware requirements, and operational costs for computational infrastructure. As resource systems are virtualized, software systems may also be mapped to virtual computational resources with elastic resource capabilities that can be adjusted on-demand, i.e. where hardware resources used to enact computations can be added or removed based on current computation needs. Virtualization and encapsulation of software in virtual machines also extend infrastructure instance management capabilities to allow, e.g., computational environments to be paused, migrated to other resources, and resumed with minimal impact on application performance.

For software, Cloud virtualization completely encapsulates the software execution environment. Compatible virtual machine disk images are typically supplied on-demand by Cloud providers, and may be customized by end-users before instantiation in Clouds. Customization of virtual machine images allow great flexibility in software deployment, and can be used to, e.g., support and integrate legacy systems, provide access to customized computation environments, or emulate existing environments for isolated testing of software.

While the term virtualization has in the context of Cloud computing become somewhat synonymous with hardware-enabled paravirtualization, it is in this work used in the more broad sense to describe the process of making something (e.g., computational resources or access to software libraries) virtual.

4.3 Infrastructure

The National Institute of Standards and Technology (NIST) [188] defines Cloud computing to be "a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [135].

As noted by this definition, a key enabling characteristic of Cloud computing environments is that infrastructures are able to automate on-demand scaling of computational capacity. To end-users, Cloud systems provide the illusion of infinite resource capacity through dynamic infrastructure load balancing techniques such as service consolidation, virtual machine migration, and Cloud federation. To infrastructure providers, Cloud systems provide virtualization capabilities that facilitate scaling, consolidation, and administration of large resource sets. Cloud environments are by definition distributed and virtualized, and typically focused on service-based provisioning of computational resources.

Cloud deployment models can be categorized based on federation and access models in four major categories [135].

- private Clouds are operated solely for the purposes of an organization and restrict access to Cloud resources to organization entities.
- community Clouds are operated and shared between a community of organizations and restrict access within the community.
- public Clouds are operated by organizations or industry groups that provide Cloud services to the general public.
- hybrid Clouds are composed by multiple Clouds that are bound together using technology that provide data and application portability within the hybrid Cloud.

Future Cloud infrastructure models are expected to be categorized as [69]

- bursted internal Clouds where service providers utilize commercial Clouds to extend the computational capacity of local infrastructures.
- federated Clouds where Cloud infrastructures migrate virtual machine instances to other Clouds to extend Cloud capacity.
- multi-Clouds where dedicated Cloud brokers delegate service provisioning between Cloud actors (end-users, client software, services, and Clouds).

For computational science, Cloud computing infrastructures provide location transparent access to computational infrastructures that virtualize computational resources and provide elastic computational capacity on-demand. Computational applications are in the stratified Cloud service model residing in the SaaS layer and operating on virtual resources residing in the IaaS layer. Due to the administrational flexibility and the execution environment customizability offered by Cloud systems, scientific computational resources are expected to be organized in private Clouds on large scale. Public Cloud environments are currently expected to be used for scientific computations primarily to meet temporary peaks in computational capacity requirements and to avoid resource (over)provisioning costs.

4.4 Scientific Applications

While Grid and other forms of scientific computations typically take the form of batch executions of programs that assume static resource behavior, Cloud applications are not limited in this way. Cloud applications may be batch or interactive, short or long running, sequential or parallel, synchronized or embarrassingly parallel, and may vary in computational performance with resource and infrastructure elasticity. As Cloud applications execute within virtual machines, data management in Cloud applications tend to be decoupled from

machine images through distributed file systems or Cloud storage solutions. Virtualization of storage and computational resources allows reformulation of computational models and utilization of data-parallel programming models, e.g., MapReduce [45] and Dryad [105], using data-centric system designs and declarative programming languages [10].

Mapping scientific computations to Cloud environments follows the same model as generic Cloud applications. Virtual machine disk images are acquired, customized, and instantiated, and computations are started and controlled both on process and virtual machine level. As Cloud environments are designed to provide short ramp-up time, low start-up costs, and utilize metered pay-per-use cost models, scientific computation applications may benefit from use of Cloud environments for algorithm testing, prototype design, and small-scale experimentation. As Clouds run software as services with unknown lifecycles, interactive applications are expected to be easier to migrate and benefit more from utilization of computational capacity in Cloud environments.

While the use of spot instance and trading models [203] can make access to public Cloud computational capacity affordable for small-scale experiments, the use of traditional computational environments tends to be more economically feasible for large, long-running experiments. Use of computational capacity in private Cloud environments is subject to the same kind of economical models used in traditional computational environments.

Application migration between Cloud providers is made difficult by image, contextualization, and API compatibility issues. The first two are addressed by the use of virtual appliances (encapsulated virtual machine images), from which providers can derive implementations. API compatibility issues are mediated by providers offering semantically equivalent interfaces for, e.g., deploying and terminating environments, and addressed through standardization [115].

For computational science, the efficiency of the use of Cloud computing infrastructures is likely to depend on the operational characteristics of individual applications. For example, tightly coupled applications with explicit synchronization requirements, e.g., MPI applications, may suffer performance degradations from resource elasticity and communication synchronization [107] while high-throughput applications may be used opportunistically in Cloud scheduling and potentially suffer performance degradations from virtual machine image migration overhead [131].

Early benchmark results indicate that compared to HPC Grid resources, Cloud resources exhibit lower suitability for computational science [197]. As modern hypervisor technology imposes low performance overhead on computations [23, 189, 206], this is attributed to factors such as use of lower performance network interconnects and resource sharing [152]. Emerging availability of special purpose hardware and high-performance computing [13] services in Clouds is likely to address these issues. Regardless of capacity issues, use of Cloud infrastructures for computational science afford scientific applications access to rapidly scalable computational capacity that can be used to meet temporary increases in computational resource requirements.

4.5 Comparing Grid and Cloud Computing

As two major recently introduced paradigms for virtual computational infrastructures, Grid and Cloud computing have evolved and influenced each other in methodology, application, and development. While the Grid and Cloud computing paradigms share many goals and approaches, e.g., transparency in resource utilization, virtualization of computation enactment, transparent management of large resource sets, provisioning of computation as a utility, and use of service-oriented utilization and development models [82], a number of key differences between Grids and Clouds exist and can be characterized by

- origin and application. While Grid computing has emerged and found extensive support in the scientific community, Cloud computing has primarily been developed and found application in industry.

- integration models. Grid environments are constructed through federation of computational resources that are aggregated and integrated through middlewares and standardized resource interfaces. Cloud resource environments are fully abstracted by Cloud infrastructure interfaces and may be constructed using technology of the Cloud provider's choosing. Standardized Cloud federation interfaces are emerging, but have yet to gain widespread acceptance.

- economical models. Being primarily scientific environments, Grid usage allocations are typically distributed using academic application and grant models, and are enforced as resource utilization quotas. Cloud usage models vary for different cloud types. Public Clouds typically employ resource capacity metering models and charge users using pay-per-use cost models. Private Clouds use domain-specific usage allocation models, e.g., grant models for academic private Clouds.

- infrastructure and resource administration models. Grids aggregate computational resources and form virtual systems that are mapped onto existing computational resources. A key characteristic of Grid environments is that resource administrators retain full administrative control of their own resources, and can explicitly control the availability, capacity, and environment of their resources. Cloud environments isolate and virtualize infrastructure, platforms, and applications, and allow end-users to administrate virtual machines and select the amount of resource capacity dedicated to enacting computations in virtual machines. Cloud administrators retain full control over physical resources used to enact virtual infrastructure services.

- resource organization models. While most (HPC) Grid and Cloud computational resources are organized in server models dedicated to computation, Grid computational resource availability is subject to administration policies and may be provisioned through multiple concurrent

resource organization configurations, e.g., HPC interfaces and Grid interfaces. Cloud resource organization is abstracted within Cloud infrastructure and virtualized computational capacity is offered through metered services models.

- virtualization models. Grids virtualize system-wide resource capacity, user bases, and computation enactments tasks such as brokering and resource selection for computational tasks. Clouds build on hardware-enabled paravirtualization technology and virtualize computational resources, infrastructures, and enactment of virtual machines.

- computation models. Grid environments are like most computational science environments limited to batch execution models for programs, and run job executions with finite lifecycles on physical resources. Cloud environments enact virtual machines that can execute any kind of computational task, and run software as services without lifecycles on virtual resources with elastic computational capacity.

More extensive comparisons of Grid and Cloud computing as paradigms and environments are available in [83, 168].

In summary, Grid and Cloud Computing are related fields that strive to realize an existing vision of resource transparency and computing as a utility. While Grids focus on integration and federation of resources, Clouds enable resource and system virtualization and provide abstraction models that isolate infrastructure components and introduce resource elasticity. Differences between Grids and Clouds tend to manifest in technology drivers and infrastructure actor stakeholding. What is shared between the two fields is a unifying view of enabling resource usage without imposing infrastructure acquisition and operation costs directly. Dynamic and highly efficient virtual environments such as Grid and Cloud computing environments alter existing software requirements and usage patterns, and introduce new possibilities for computational science.

4.6 Sky Computing

While the Cloud computing paradigm extends the resource and infrastructure virtualization properties of Grid computing environments, current Cloud computing environments lack some of the high-level computation enactment virtualization support of Grids [164]. With the relative immaturity of the paradigms, and the overlap of goals and methodology between Grid and Cloud computing, hybrid approaches to realization of virtual infrastructures for scientific computing are emerging. For example, Sky computing [115] is an emerging computing pattern that combines dynamically provisioned distributed domains built over multiple Clouds and aim to provide high-level support for virtualized computation enactment by mapping virtual cluster and Grid environments to Cloud infrastructures.

By combining the resource transparency, virtual organization support, and accounting models of Grids with the infrastructure virtualization and resource elasticity of Clouds, Sky computing applications aim to provide high-level abstraction models suitable for computational science applications in multi-Cloud environments. Mapping of Grid environments onto Cloud infrastructures is expected to introduce new usage models for computations and reduce the complexity of maintaining and enacting computations in Grid environments [109]. The approach taken in this work is similar to Sky computing in the focus on abstraction and decoupling of virtual infrastructures from (physical) computational infrastructure. However, this work focuses on integration across computational paradigms rather than multi-Cloud integration issues, and aims to identify suitable architecture design and software development methodologies for abstraction of multiple types of computational infrastructures.

Chapter 5

Virtual Infrastructure Software Development

Virtual infrastructures and dynamic environments such as those of Grid and Cloud computing change the dynamics and interactions of computational science software. As these environments are virtualized and distributed, software components are often modeled and exposed as services. The definition of a software service used here details a service to be a network-accessible software component with an always-on semantic. The concept of Service-Oriented Computing [175] details construction of systems and software modules that are realized as location transparent, dynamically discoverable, and self-describing (machine understandable) software services. Dynamic service discovery and invocation are typically resolved through use of service description techniques and service registries. Construction of software components as location transparent, distributed services facilitates dynamic integration of systems and components in virtual infrastructures and increases software flexibility and applicability [148]

5.1 Distributed Computing

Construction of virtual computational infrastructures for computational science entails design and deployment of distributed systems [41] that aggregate the computational capacity of multiple distributed, autonomous resources and systems that communicate over networks. Distributed computing distinguishes itself from centralized computing primarily in that fewer assumptions can be made about availability, synchronization, and capability of computational resources, and that distribution of a system typically introduces heterogeneity in computational resources and additional failure models. In general, the study of distributed computing problems resolves to study of trade-offs in terms of costs for communication overhead versus computational capability gained, management of distributed failure models, and identification and design of suitable

abstraction levels for distributed applications. Abstraction models provided by computational systems today approach a level where end-users no longer are required to have extensive knowledge of where their computations are performed or how their data are stored.

The public perception of distributed computing is that the distributed era came about with the advent of the World Wide Web. Historically however, distributed computing can be seen to be almost as old as computing itself. Early supercomputers and mainframes utilized client-server interaction models that have survived to date. Most computer systems today provide some form of Internet connect, and operating systems typically support the use of distributed file systems, network-based data sharing, and remote compute resources. When computational resources are aggregated to form distributed systems, aggregation typically introduces heterogeneity in the resource set. System heterogeneity manifests in many forms, e.g., hardware types, operating systems, communication stacks, software availability, or protocol types. To manage system heterogeneity, distributed systems typically implement abstractive interfaces that virtualize system resources and capabilities.

In distributed systems, communication performance is measured using metrics such as latency, bandwidth, and throughput. Compared to centralized systems, distributed systems generally suffer performance degradations from synchronization and communication overhead. In the case of scientific computation, this can for example manifest itself in overhead for transferring data to and from computational resources, so called data staging. Distributed systems can often compensate for this by exploiting parallelism in system tasks, e.g., by transferring data required for future calculations while other calculations are performed. Peer-to-peer mechanisms like, e.g., BitTorrent [159], demonstrate that high system-wide throughput can often result in higher performance for individual nodes than naive point-to-point transfer schemes allow.

System performance requirements often categorize systems in distributed system scenarios. Applications that measure performance in response times, e.g., interactive applications, require low network latencies for computational efficiency. Systems that measure performance in data or computational throughput, e.g., file transfer systems, are dependent on maintaining high network bandwidth to sustain performance. In some cases, e.g., multi-user computer games, high bandwidth can be used to compensate for high latency by performing speculative execution and use corrections on local nodes when synchronized state updates arrive.

For scalability reasons, public networks are today built using packet-switched networks and maintain unreliable communication channels. As in the case of the Transmission Control Protocol (TCP) [158], reliable communication channels may be emulated on top of unreliable mechanisms, but a number of complicated interactions still result in distributed call semantics being far more complex than centralized call semantics. Distributed failure scenarios include e.g., synchronization errors, message drops, and communication timeouts, and may result from programming errors (e.g., race conditions, deadlocks, and busy

wait overloads) or communication infrastructure errors (e.g., power outages, network congestion, router failures, security attacks, and server overloads). In general, synchronization and security requirements in distributed systems add additional dimensions to development complexity and result in a need for abstractive mechanisms that handle system communication and integration.

In summary, when compared to centralized systems, distributed systems suffer overhead and complexity issues for distributed state synchronization, distributed invocation semantics, distributed failure management, distributed security issues, response time (latency) issues, and throughput (bandwidth) issues. Programming models for distributed systems typically either rely on specialized software frameworks (e.g., middlewares) for synchronization and communication coordination, or expect programmers to handle such issues explicitly. Few programming languages today abstract distributed programming issues beyond managing network connections. To decouple scientific computational applications from computation enactment, virtual infrastructures for computations should not only abstract computation enactment tasks, but also provide failure management models that abstract distributed computing errors and provide transparency in resource utilization.

5.2 Service-Oriented Computing (SOC)

Large-scale computational science applications and virtual infrastructures, such as Grid and Cloud environments, often build on and integrate existing components and systems. Integration of software systems tends to produce code specific to the integration environment and limit software reusability. To address these issues, software services are often employed to abstract legacy components and provide network-accessible interfaces well suited for integration. Service-Oriented Computing (SOC) [175] is a software development paradigm that models software components as network-accessible services and is concerned with concepts such as loose coupling, late binding, and location transparency. In SOC, software components are modeled as services, and systems model service interactions defined in terms of service interfaces. Current SOC methodology renders services as Web Services, which extend the notion of a service to, e.g., include platform independent (often text-resolved) data types, definition of service interfaces in machine interpretable service descriptions, and communicate using message-oriented communication patterns.

Production of software components as services introduces new software requirements on both component and system level, and skews the traditional software development landscape. Services are typically self-contained and modular, provide high levels of interface abstraction, and shift focus from component implementations to component interfaces. As services are loosely coupled in multiple dimensions [155], services make few assumptions about spatial or temporal dependencies between service clients and services, and service communication tends to be kept coarse-grained and document-oriented. As service

invocation patterns in dynamic virtual infrastructures may vary with, e.g., request patterns, and resource and network load, asynchronous communication patterns are often employed for non-trivial service operations.

In industry, service-oriented systems are often employed to expose legacy functionality and integrate business systems and components. As service-oriented systems incorporate business logic that require constant modification to accommodate business requirements [208], production of highly maintainable software is a key challenge of SOC [156]. Studies estimate that a majority of software development resources are spent on software maintenance and system updates [160, 209], which is often used to motivate the use of SOC methodology.

5.3 Service-Oriented Architecture (SOA)

Service-Oriented Architecture (SOA) [64] is an architectural style in SOC that realizes software architectures in terms of services and service interfaces. Similar to how object-orientation models objects and relationships between objects, SOA models service actors and relationships between actors. Object-orientation has greatly influenced the development of service-orientation in general, and SOAs in particular. Many of the goals of the paradigms overlap, e.g., improving software modularity and reducing maintenance overhead and development costs, and many SOA component-level design patterns [65] have corresponding object-orientation patterns.

SOA utilizes a high-level, top-down design perspective that focuses on isolation and integration of components to form systems. Services are classified as atomic (a.k.a. informational or simple services) or composite (i.e. services composed of other services) [154], and are modeled based on roles, interaction patterns, or functionality. Focus in component-level SOA design is placed on the utility of software components, rather than the mechanism of components, which is intended to make components easier to integrate and reuse.

Service interfaces are defined in service descriptions, and architectures are constructed through combinations of SOA techniques such as composition, orchestration, choreography, and aggregation of services. SOA actors may be modeled on, e.g., roles (developer, provider, consumer), interaction patterns (client, service, broker), or functionality (service, aggregator, registry). As SOA is a relatively recently introduced and widely applied field, definitions of service-orientation and SOA tend to detail methodology and guidelines rather than technology and realizations.

SOA design places great focus on identification and exposure of functionality (i.e. business logic), and details design requirements for component realizations in terms of abstraction, loose coupling, and interoperability. Key characteristics of a SOAs include, e.g., loosely coupled and dynamic service interactions, and service realizations that are fully abstracted by service interfaces.

In SOAs, services are typically hosted in service containers that abstract message transport logic, handle traditional server issues, and provide manage-

ment interfaces for services. Service containers abstract server development tasks and allow service developers to focus on design and implementation of service interfaces. Techniques such as thread pooling, instance duplication, and instance reuse make service containers efficient hosting environments that decouple service development from distributed programming issues, and help virtualize service instances.

5.4 Web Services

While many types of distributed component models, e.g., the Distributed Component Object Model (DCOM) [174], the Common Object Request Broker Architecture (CORBA) [157], or the Windows Communication Foundation (WCF) [130], can be used to realize services in SOAs, current technology renderings of SOAs tend to favor the use of Web Services. Web Services provide text-based, platform independent interfaces for message-oriented communication and build on freely available technologies and formats such as the Hypertext Transfer Protocol (HTTP) [70], the Extensible Markup Language (XML) [29], and the JavaScript Object Notation (JSON) [42] format.

While Web Services may provide human readable interface descriptions and message formats, they are designed and intended for machine-to-machine communication. Web Services originated as a technology for firewall-friendly distributed interprocess communication and have, due to generic applicability and use of established technology bases, evolved into a mature communication paradigm with wide acceptance in industry and academia. There are currently two major categories of Web Services: XML-based SOAP style Web Services that define explicit service interface descriptions and exchange generic messages, and RESTful Web Services that provide Web APIs for service communication and expose service resources in custom formats over HTTP.

5.4.1 SOAP Style Web Services

SOAP style Web Services [154] define service interfaces and exchange messages using standardized XML-based languages and protocols. SOAP [92] is a wire protocol (i.e. a communication protocol focused on application-level data representation) that supports multiple communication models including, e.g., one-way messaging and message routing. SOAP defines an extensible message representation model that allows protocol extension for, e.g., security and encryption of messages [8]. The use of SOAP as a wire protocol allows SOAP style Web Services to abstract message representation formats, decouple message (and service interface) representations from message transmission, and transfer messages over paths of intermediaries (message routing).

The Web Service Description Language (WSDL) [39] is a language for definition of self-contained and self-describing service descriptions. WSDL documents declare type sets using XML Schema [31], define service interfaces

in terms of message specifications, and map bindings of service interfaces to concrete service realizations, e.g., by defining SOAP and HTTP as wire and transmission protocols for accessing specific service instances. The use of standardized service descriptions allows SOAP style Web Service clients to dynamically discover and invoke services through publishing of service descriptions in service registries such as Universal Description, Discovery and Integration (UDDI) [3] and automated generation of service client communication stubs [120]. SOAP style Web Services support multiple interaction patterns, e.g., one-way, request-response, publish-subscribe, and are recommended to use coarse-grained, document-oriented, and literally encoded message schemes [21].

By definition, SOAP style Web Services are stateless. The Web Services Resource Framework (WSRF) [77] defines a set of standards for addressing service state management in SOAP style Web Services. WSRF defines message, interface, and protocol extensions for service and resource representation, identification, management, and aggregation.

5.4.2 RESTful Web Services

Criticism against SOAP style Web Services is typically directed against the overhead and complexity of the SOAP and WSRF software stacks. Representational State Transfer (REST) [71] outlines an alternative approach to representation of services in web environments. In REST, services are modeled around resources that are exposed in custom formats and transferred using HTTP. Resource-Oriented Architectures (ROA) [153] provide guidelines for, e.g., definition of resource representations, mapping of resource representations to access methods in HTTP, and identification of resources. As RESTful Web Services utilize HTTP as the sole transmission mechanism, RESTful Web Services are limited to request-response communication patterns.

RESTful Web Services may be implemented using standard component models, e.g., Java Servlets [138, 187], where service interfaces may be web-based and exposed through similar technologies such as JavaServer Pages (JSP) [26]. REST services typically provide service interface descriptions in documentation and access to services through RESTful web APIs. REST service interface implementation includes definition of structure and formats for resources, and mapping of resource access mechanisms to HTTP methods in Create, Retrieve, Update, and Delete (CRUD) operations. While the definition of CRUD operations is part of the service interface definition, guidelines for interpretation of REST service interfaces exist [71]. RESTful Web Service message serialization formats vary, and may include the use of, e.g., JSON, XML, or custom formats.

An argument sometimes made for REST is that service communication may be cached, which is theoretically true but rarely practically useful as the existing web infrastructure does not provide caching for all traffic. Internet scalability for services is achieved through segmentation of the infrastructure, in the case of the web by geographical dispersion of servers and clients.

5.5 Loose Coupling

Regardless of what kind of service technology is used to render SOA services, a key focus of service-orientation is to provide architectures built using loosely coupled components. Multiple definitions and dimensions of coupling exist [155]. In this work focus is placed on the aspect of minimization of formal knowledge required for component interaction. In terms of services, loose coupling can manifest in, e.g., production of service descriptions that fully abstract service implementations, i.e. provide all knowledge required to interact with services. The use of platform independent and widely available technology for service description allows service clients and service implementations to be constructed using different programming languages and hosted in different environments. Loose coupling in service components facilitate system behaviors such as dynamic service discovery, live migration and replacement of service instances, diversity in component communication patterns, flexibility in component integration, agility in system development, and reduced system maintenance through increased modularity.

In addition to component-level loose coupling, it is also desirable to decouple service instances from service environments. Like service interfaces should be decoupled from service implementation details, service implementations should be decoupled from service realization or hosting details. Service containers are used to virtualize and abstract hosting of services, and can themselves be abstracted in virtualization-based Cloud computing environments. For further decoupling of service instances from service communication enactment (message transmission), event-driven messaging engines such as Message-Oriented Middleware (MOM) [22] or Enterprise Service Bus (ESB) [36] solutions may be employed. ESBs and MOMs provide additional qualities of service in service communication and message delivery, such as message routing, security, and capability metering.

As illustrated in Figure 5, service components for client implementation, interface implementation, logic, and state management may be separated to facilitate flexibility and scalability in service development and deployment. Separation of service logic and interface implementations allows for the use of established component models such as the Component Object Model (COM) [202], the Common Object Request Broker Architecture (CORBA) [157], or Enterprise Java Beans (EJB) [133], and enables the use of component container technology to virtualize and scale logic component instantiation. Similarly, separation of state and storage management from logic implementation facilitates flexibility and scalability in storage management and allows service implementations to be formulated as classic three-tier architectures [50].

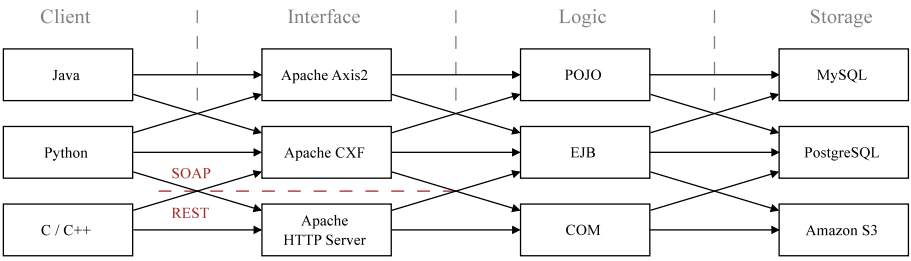


Figure 5: Decoupling of service interface from service logic and state. Decoupling of service interface implementations and service hosting environments (in the illustration the Apache Axis2, CXF, and HTTP Server containers and servers) from service logic gives freedom of choice in service logic implementation language and component model (in the illustration Plain Old Java Object (POJO), Enterprise Java Beans (EJB), and Component Object Model (COM)). Similarly, decoupling of service logic from service state management and storage gives freedom of choice in storage technology (in the illustration the MySQL and PostgreSQL database engines and the Amazon Simple Storage Service (Amazon S3)).

5.6 Service Coordination

As SOA applications and systems are constructed by combining services, tools and techniques for composing and orchestrating service interactions are required. Mashups are light-weight applications formed by combinations of Web Services to provide aggregated content, typically in the form of web interfaces constructed using specialized web integration APIs [161]. Workflows are formalized descriptions of sets of activities, data, and dependencies that model a flow of tasks through a system [99]. Workflows can be classified as, e.g., static or dynamic, abstract or concrete. Workflow techniques coordinating the use of Web Services are typically represented in text-based workflow descriptions. Workflows are often modeled using constructs such as directed acyclic graphs, and represented in formalized workflow descriptions. The Web Service Business Process Execution Language (WS-BPEL) [11] is an OASIS [143] standardized XML-based workflow language for coordination of SOAP Web Services. WS-BPEL defines a data model to represent (typically long-lived) business processes and their constituted components. In WS-BPEL, workflows are defined as services themselves and may because of this be recursively combined to form larger workflows.

Typically, in WS-BPEL service coordination workflows, some or all of the following are modeled to describe system interactions: message flow, data flow, control flow, process orchestration or choreography, fault and exception handling. Services coordination using workflows may be subject to Service-Level Agreements (SLAs) [95] that regulate terms for interactions between

services and consequences for breech of contract. Alternatives to WS-BPEL exist and include, e.g., the Web Service Choreography Description Language (WS-CDL) [113], the XML Process Definition Language (XPDL) [200], and the Yet Another Workflow Language (YAWL) [194]. Characterization of service coordination mechanisms can be done by, e.g., coordination perspective (neutral or enacting party), or how the coordination is expressed (e.g., message level or communication role).

Differentiation of mashups and workflows can be based on, e.g.,

- typing. Mashups are weakly typed and use ad hoc type systems based on what services and data are available. Workflows define strongly typed service interfaces designed for interoperability.

- coordination. Mashups use implicit, ad hoc coordination models while workflows define explicit coordination mechanisms that often detail service interactions on message or interface level.

- applicability. Mashups are designed to efficiently produce light-weight content aggregations. In computational science, this may manifest as web-based administration or data visualization interfaces. Workflows are generally applicable and are, in computational science, more often used to coordinate and model interactions among (sets of) computations.

For computational science, a number of tools for computation enactment and coordination exist, including workflow languages, e.g., Karajan [195], the Abstract Grid Workflow Language (AGWL) [66], and SMAWL [179], and enactment engines, e.g., Taverna [146], Pegasus [47], Kepler [9], Triana [40], and Gridflow [34, 2]. Curcin et al. [43] provide a framework for high-level comparison of scientific workflow tools detailing differences between data- and control-driven workflows. For classification of workflow systems used in Grid computing, Yu et al. [207] provide a taxonomy of workflow systems that details workflow design, scheduling, fault and data management, and coordination models. Deelman et al. [46] provide a survey of workflow systems for eScience, classifying workflows by representation, control flow, and application models.

While workflows provide end-users with little programming experience a meta-level programming construct, the expressive power and complexity of such constructs do not always scale to practical use [56]. The approach of this work is to employ workflow languages only as descriptions of automated sequences of computation enactments and perform computation coordination using programming language constructs. The line of reasoning supporting this approach is derived from the categorization of static and dynamic scientific computation workflows. For static (i.e. non-changing) workflows, specialized workflow languages are sufficient. For dynamic workflows (that change over time and may evolve depending on intermediary computation results), programming languages are utilized as their expressive power better captures the complexity of dynamic workflow enactment.

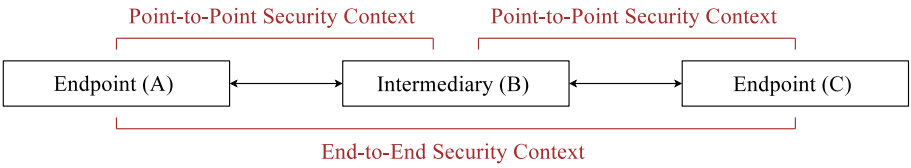


Figure 6: Point-to-Point versus End-to-End security contexts. Secure communication across intermediaries require end-to-end security context models.

5.7 Security

As service-based systems tend to be virtualized and distributed, security structures in service-oriented systems can rarely make assumptions about underlying security systems. When service communication is message-oriented and service interfaces are decoupled from transport protocols, service communication can be tunneled through secure communication channels. In the case of REST services, security is typically designed to be point-to-point and rely on existing security layers such as HTTPS [167] and SSL/TLS [48]. As SOAP supports message routing, i.e. delivery of messages through paths of intermediaries, additional security constructs are required to create end-to-end security models (illustrated in Figure 6). To this end, additional SOAP security extensions for message-level security are defined in the WS-Security specification [199], and a set of complementary security mechanisms for creation of security contexts are defined in the XML Key Management Specification (XKMS) [73], Security Assertion Markup Language (SAML) [93], and eXtensible Access Control Markup Language (XACML) [88] specifications.

Security requirements in computational science tend to vary with applications and the sensitivity of application data. Infrastructure level security requirements range from very high (HPC deployments) to very low (certain P2P and HTC infrastructures), and may be based on trust models where computational hosts are trusted (often the case in Grid and Cloud computing environments) or require computations to be performed redundantly to ensure correct results (certain HTC infrastructures). Applications may add application-level security constructs that are enforced by the applications themselves, e.g., adding encryption layers to data storage, but are generally subject to the trust model of the enacting infrastructure in virtual infrastructure environments.

5.8 An Ecosystem of Infrastructure 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 infrastructures, and lack ability to be generally applicable to computational problems [58]. This work addresses virtual infrastructure

software design for scientific applications that support the majority of current computational approaches, and places focus on infrastructure composition and scalability rather than specific problem sets [57].

The methodology of this work builds on the idea of an ecosystem of infrastructure software components [86], 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 computational resources to applications and end-users. Here, standardization of interfaces and software components help define niche boundaries, and continuous development of virtual infrastructure components and integration with scientific 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 a component ecosystem are central, and generally help in identification and formulation of research questions. Software designed using this methodology focuses on the establishment of core functionality, and adapts to, and integrates with, members of neighboring niches rather than attempts to replace them.

Currently, advanced scientific applications and computational infrastructures require software and systems to scale with problems and abstract heterogeneity issues introduced by this scalability. For usability, software also require interoperability and robustness to enable automation of computation enactment 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, and integrate with, existing middlewares and systems, and create autonomous components that co-exist and integrate non-intrusively in existing deployment environments. The overarching goal of this work is to identify and develop suitable methodologies for construction of virtual infrastructures that decouple computational applications from (physical) computational infrastructures.

Chapter 6

Summary of Contributions

This work addresses design and implementation of virtual infrastructures for distributed scientific computational infrastructures. The main area studied is virtual infrastructures for computational science intended for use in Grid and Cloud computing environments. Contribution focus is placed on individual (sub)problems such as job and resource management, usage allocation enactment, and service-based software development. The software development approach used is based on the perspective of the Service-Oriented Computing paradigm, and a design and development methodology aimed at sustainable service software development is explored. Contributions are made in the forms of scientific publications and software artifacts addressing construction of, and application integration with, virtual scientific computational infrastructures.

6.1 Job and Resource Management

Grid job and resource management systems are distributed systems that virtualize and manage resource sets, and can be classified, e.g., by type as computational, data, and service Grids [121]. In Grids, native resource systems are typically abstracted by middlewares [76] that virtualize resource views and decouple job submission and control interfaces from resource systems [27]. To virtualize and decouple job enactment from Grid resource systems and (to some extent) middlewares, Grid computations are often matched to computational resources in brokering models [172], where autonomous software components called brokers bridge the gap between clients and Grid systems.

Brokers may employ benchmarking tools such as GridBench [91] and GrenchMark [102] to categorize and quantify Grid resource performance. In addition to resource categorization information, brokers may utilize resource system information published in middleware information systems to adapt to resource states and optimize job to resource placements. A number of approaches to categorizing Grid resources and utilize benchmarks and historical informa-

tion to predict Grid computation performance and runtimes exist and include, e.g., [6, 193, 176, 110]. Depending on middleware construction, brokers may employ conceptual push (as in the case of, e.g., environments based on the Globus Toolkit [75]) or pull (as in the case of, e.g., BOINC [15] systems) models for job to resource placement.

In Grid environments, brokers typically operate on top of middleware and enact metascheduling policies defined by end-users [132]. A number of different metascheduling approaches exist and include, e.g., WSRF notification approaches [137], adaptive frameworks for high-level abstractive job scheduling [100], and approaches utilizing performance metrics to optimize job placements [5]. In addition, a number of approaches addressing interoperability between Grid middlewares and metaschedulers exist and include, e.g., approaches to standardize Grid interfaces [78, 79] and approaches proposing metascheduler interoperation models [28].

Wide variety in approaches to Grid middleware construction and integration exist, and include, e.g., approaches that provide integration toolkits and modular middleware services [75], approaches focused on enactment efficiency [163], and attempts to provide middleware functionality through native operating system services [112]. Programming models for computation enactment in Grid environments include middleware abstraction interfaces such as SAGA [89] and GridLab GAT [7], as well as a range of middleware-specific interfaces.

The approach taken in this work includes abstraction of middleware dependencies and provisioning of middleware agnostic interfaces for job management through a hierarchical Grid functionality model proposed in Paper III [54]. A realization of this model, the Grid Job Management Framework (GJMF) [149] presented in Paper IV, exposes functionality for resource selection, job brokering, submission, and control, and high level fault tolerant job management in Grid environments. The agility inherent from construction of the framework as a loosely coupled network of services that allows dynamic (re)composition and configuration of the framework is discussed in Paper II [59], and evaluated in the context of service-oriented software development in Paper VIII [148]. Paper V [150] investigates impact of Web Service overhead on Grid infrastructure components and quantifies the efficiency of the framework. Paper VI [58] outlines a model for integration of the framework with applications and Grid infrastructure components through combination of the framework service interfaces, the framework client API, and a set of integration bridge modules.

6.2 Usage Allocation Enactment

Resource usage capacity allocations are in scientific environments typically distributed using application and grant models. For example, HPC users may apply for usage capacity at resource sites from administrative organizations, and receive capacity allotments for specified grant periods (time windows). Public Cloud environments typically offer capacity as metered services, and

(like HPC and some Grid environments) quantify resource capacity using resource specific metrics, e.g., CPU hours. While private Cloud environments in general utilize similar service models as public Clouds, application and grant models for resource allocations are expected to be utilized in scientific environments. For reasons of scalability and flexibility, usage allocations in scientific environments should be specified using relative capacity metrics and in models that are capable of meeting changed allocation requirements dynamically [151]. Dynamicity in allocation requirements manifests at multiple levels, e.g., in dynamically changing computational capacity requirements for individual users and projects, or in that Grid users are organized in dynamic Virtual Organizations (VOs) that may gain or lose members at any time.

Capacity allocations are in Grid environments made on VO level and map to infrastructure capacity at resource site level. For Grids, usage records are typically assembled and monitored in accounting systems, and statistics for usage consumption patterns are made available in trace logs. While Cloud computing environments currently lack generic user-level organization support mechanisms, existing Grid mechanisms may be extended and mapped to Cloud infrastructure services to create virtual computational science environments.

While scheduling of jobs in parallel cluster environments is predominantly addressed as a space-sharing problem, usage allocation within user groups is essentially a time-sharing resource allocation problem. Operating system fairshare scheduling [114] is an allocation scheme that allows processes to share a centralized resource (CPU) via time-sharing and context switching, and aims to over time ensure that CPU time is shared on per-user basis rather than on per-process basis. Existing HPC cluster scheduler mechanisms such as Maui [185] and SLURM [204] contain fairshare mechanisms that extend the fairshare concept to share resource capacity in cluster environments on per-user basis. Fair resource utilization is here defined as resource capacity utilization over time converging to policy-defined usage (pre)allocations and is enforced by jobs being prioritized after comparisons of (user) usage allocations and historical usage consumption. Schedulers typically view finite time windows of usage records and influence of fairshare on job prioritization is modulated using mechanisms such as usage decay functions and scheduling factor weights.

As Grid usage policy enactment exhibits dynamicity in multiple dimensions, mechanisms for enactment of Grid usage allocations need flexibility to be able to match the dynamicity of Grid environments in, e.g., dynamic VO structure changes and resource volatility. Cloud environments with migratable virtual machines and elastic resource capacity are expected to introduce further dynamicity in usage allocation enactment. In addition, Usage allocations are likely to be scheduled, prioritized, or updated dynamically, and resource sites may switch, customize, schedule, or reprioritize usage policies at any time. To accommodate such dynamicity, virtual infrastructure policy definitions and policy enactment mechanisms need to be flexible in structure and provide mechanisms capable of supporting frequent updates in policy specifications and usage data, and management of large sets of usage data.

The approach of this work emphasizes decoupling of Virtual Organizations and virtual infrastructure mechanisms used for computation enactment. Paper VII [151] presents FSGrid, a distributed system for fairshare-based load balancing in federated Grid environments. The system is based on three contributions: an intuitive policy model that supports delegation of policy administration, an efficient algorithm for fairshare job prioritization, and a decentralized architecture for realization and integration of the system. While the system is developed for Grid environments, the generality of the model and the flexibility of the architecture allows integration of the system in any environment that performs job prioritization in scheduling, e.g., traditional HPC deployments and distributed HTC environments.

6.3 Sustainable Service Software Development

Software development for virtual scientific infrastructures such as Grid and Cloud computing environments is commonly addressed using service-based software development techniques. The current methodology of service-oriented computing renders software components as Web Services, and a number of software development techniques ranging from methods for formalization [124] and specification [32] of service-based component models to general purpose SOA design patterns [65] and full lifecycle service development methods [19] exist. The more specific problems of construction of scientific applications and integration of scientific applications with virtual infrastructures are addressed by software frameworks and middlewares such as the Globus Toolkit [75].

Due to high resource requirements and limitations in academic funding models, software development for computational science environments tend to lie on the front edge of technological development and consist mainly of prototype and system integration projects. As such projects contain volatility and dynamicity rarely captured by generic software lifecycle models, the applicability of established software development tools and techniques can be limited. In general, service-based software development is complex, and the immaturity of scientific software prototypes and open source tools (as compared to established commercial software) results in software stacks that are time consuming to develop and hard to maintain. Additionally, scientific applications and environments are typically more dynamic and evolve at a faster pace than most software environments, resulting in a need for greater flexibility and higher abstraction levels in scientific software development.

Paper I [57] analyzes virtual infrastructure software development from a software engineering perspective and formulates a view on software evolution captured by the notion of an ecosystem of virtual infrastructure components. The software composition model and architecture patterns of Paper II [59] address a need for flexibility in composition and construction of scientific service-based software. A key aspect of the approach of this work is to support development methodologies that allow software components to act as both network-

accessible services and local objects in applications, a mechanism intended to increase system development and deployment flexibility. Paper VIII [148] extends the work of papers I and II, and proposes a software development methodology aimed to abstract development complexity and increase system and component flexibility. Paper VIII also presents a toolset designed to support the methodology through mechanisms such as simplified service description formats and code generation mechanisms. The overall goal of this work is reduction of software complexity, and facilitation of development models that support automation and flexibility in software construction.

6.4 Papers

To limit scope, the publications of this thesis address individual problems within construction or integration of virtual infrastructures for computational science. Problems addressed include, e.g., job and resource management, enactment of usage allocations, system integration issues, and software development methodologies. Three of the papers outline and discuss approaches to virtual infrastructure software development; Paper I from a software engineering perspective, Paper II from a system (re)factorization point of view, and Paper VIII from a service-oriented software development methodology origin. Four of the papers (papers III, IV, V, and VI) investigate and outline a generic architecture for Grid job management capable of adoption in a majority of existing Grid computing environments. Papers III and IV discuss the architecture and design of the framework while Paper V studies the impact of Web Service overhead on framework performance and Paper VI integration of the framework in a production environment. Paper VII addresses enactment of fairshare-based usage allocation quotas in virtual scientific infrastructures, and presents an architecture and an implementation of a system for fairshare job prioritization. Paper VIII is placed last in the thesis as it contains a case study of the software frameworks presented in papers IV and VII.

6.4.1 Paper I

Paper I [57] 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 [184]; the Job Submission Service (JSS) [63], the Grid Job Management Framework (GJMF) [149], the Grid Workflow Execution Engine

(GWEE) [55], the SweGrid Accounting System (SGAS) [85], and the Grid-Wide Fairshare Scheduling System (FSGrid) [53].

6.4.2 Paper II

Paper II [59] addresses Service-Oriented Architecture methodology 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.

6.4.3 Paper III

Paper III [54] investigates software design issues for Grid job management tools. Building on experiences from previous work [60, 61, 63], an architectural model for construction of a middleware-agnostic 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 abstracts middleware dependencies through use of (possibly third party developed) middleware adaption plug-ins.

A prototype implementation based on the Globus Toolkit 4 [75] of some of the services in the architecture is presented, and the services are integrated with the ARC [52] and Globus [87] middlewares. To demonstrate the feasibility of the approach, preliminary results from prototype testing are presented along with a brief evaluation of system performance and system use cases.

6.4.4 Paper IV

Paper IV [149] elaborates on the work of Paper III and proposes a composable Service-Oriented Architecture-based framework architecture for middleware-agnostic 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 III 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 design and development details from the work are discussed throughout the paper.

The Grid ecosystem model of Paper I is further developed and discussed in the context of the proposed job management architecture, and the software composition techniques of Paper II 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, API, and middleware efforts within adjoining Grid ecosystem niches.

6.4.5 Paper V

Paper V [150] investigates the impact of Web Service-related overhead on the performance of the framework presented in Paper IV. A model for characterization and quantification of Grid overhead and overhead imposed by the GJMF is formulated. Extensive sets of tests are performed on the proof-of-concept implementation to quantify the impact of Web Service overhead, and evaluate the overhead imposed by the framework against the functionality offered by the framework in realistic computational settings.

The results of the evaluation characterize framework overhead to mainly consist of Web Service invocation overhead, and the effectiveness of a set of techniques, e.g., batch invocation modes, asynchronous message processing, and local call optimizations, designed to mediate the impact of Web Service overhead is evaluated. The results indicate that overhead imposed by the framework is limited to less than one second per job in realistic computational settings, and that Web Services combined with sound architectural designs and optimization techniques can constitute a viable platform for realization of service-oriented Grid architectures.

6.4.6 Paper VI

Paper VI [58] approaches 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 cross-platform mediating layer handling resource brokering and notification delivery. The proposed architecture is illustrated in a case study where the LUNARC application portal [126] is integrated with the Grid Job Management Framework [149] presented in papers III and IV. The proposed integration architecture is evaluated in a performance evaluation and findings from the integration efforts are presented throughout the paper.

6.4.7 Paper VII

Paper VII [151] extends earlier work documented in [53] and addresses usage allocation enactment in virtual scientific computational infrastructures. The paper proposes a decentralized architecture built on an intuitive policy specification model and an algorithm for distributed resource allocation enforcement. Enactment of usage allocations is performed through injection of a mechanism for fairshare job prioritization in Grid and HPC scheduling environments. A proof-of-concept implementation of the system is presented and evaluated in a performance evaluation. System behavior is characterized and mechanisms counteracting system convergence (of resource usage to capacity preallocations) are identified. Effectiveness of system mechanisms to achieve fair distribution of resource capacity is evaluated and system scalability demonstrated.

6.4.8 Paper VIII

Paper VIII [148] addresses service-oriented software development methodology for virtual scientific computational environments. The paper proposes a development methodology based on isolation of service software components and abstraction of the service software development process. The development perspective of Paper I and the software refactorization techniques of Paper II are further extended in the paper. A set of current service-oriented software development issues are identified and discussed, and a toolset designed to support the proposed methodology through abstraction of development complexity is presented. The impact of the proposed methodology and presented toolset on recent software development projects (presented in papers IV and VII) is evaluated in a brief case study.

Bibliography

[1] Distributed.net. http://www.distributed.net/, March 2011.

[2] Grid-Flow: A Grid-Enabled Scientific Workflow System with a Petri-Net-based Interface, author = Z. Guan and F. Hernández and P. Bangalore and J. Gray and A. Skjellum and V. Velusamy and Y. Liu, journal = Concurrency and Computation: Practice and Experience, volume = 18, number = 10, pages = 1115 - 1140, year = 2006.

[3] Universal Description, Discovery, and Integration (UDDI). http://uddi.xml.org/, March 2011.

[4] C. Adams and S. Farrell. Internet X. 509 Public Key Infrastructure Certificate Management Protocols, 1999.

[5] 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.

[6] A. Ali, A. Anjum, J. Bunn, R. Cavanaugh, F. van Lingen, R. McClatchey, M. A. Mehmood, H. Newman, C. Steenberg, M. Thomas, and I. Willers. Predicting Resource Requirements of a Job Submission. In *Proceedings of the Conference on Computing in High Energy and Nuclear Physics (CHEP 2004), Interlaken, Switzerland*, September 2004.

[7] 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.

[8] 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, March 2011.

[9] I. Altintas, A. Birnbaum, K. Baldridge, W. Sudholt, M. Miller, C. Amoreira, Y. Potier, and B. Ludaescher. A Framework for the Design and Reuse of Grid Workflows. In P. Herrero et al., editors, *International*

Workshop on Scientific Applications on Grid Computing (SAG'04), Lecture Notes in Computer Science 3458, pages 119–132. Springer-Verlag, 2005.

[10] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J.M. Hellerstein, and R. Sears. BOOM Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *Proceedings of the 5th European conference on Computer systems*, pages 223–236. ACM, 2010.

[11] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, et al. Web Services Business Process Execution Language Version 2.0. *OASIS Standard*, 11, 2007.

[12] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, March 2011.

[13] Amazon High Performance Computing (Amazon EC2 HPC). http://aws.amazon.com/ec2/hpc-applications/, March 2011.

[14] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/, March 2011.

[15] 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.

[16] 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.

[17] 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 2011.

[18] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[19] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley. SOMA: A Method for Developing Service-Oriented Solutions. *IBM Systems Journal*, 47(3):377–396, 2010.

[20] Z. Balaton and G. Gombas. Resource and Job Monitoring in the Grid. *Lecture notes in computer science*, pages 404–411, 2003.

[21] K. Ballinger, D. Ehnebuske, C. Ferris, M. Gudgin, C.K. Liu, M. Nottingham, and P. Yendluri. Basic Profile Version 1.1. *WS-I Organisation*, 2004.

[22] 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.

[23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM, 2003.

[24] 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.

[25] 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.

[26] H. Bergsten. *JavaServer Pages*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2003.

[27] F. Berman, G.C. Fox, and A.J.G Hey (editors). *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley and Sons Ltd, 2003.

[28] 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.

[29] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. eXtensible Markup Language (XML) 1.0. *W3C recommendation*, 6, 2000.

[30] J. Broberg, S. Venugopal, and R. Buyya. Market-Oriented Grids and Utility Computing: The State-of-the-Art and Future Directions. *Journal of Grid Computing*, 6(3):255–276, 2008.

[31] A. Brown, M. Fuchs, J. Robie, and P. Wadler. XML Schema: Formal Description. *W3C Working Draft*, 25, 2001.

[32] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. An Open Component Model and Its Support in Java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt Wallnau, editors, *Component-based Software Engineering*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer Berlin / Heidelberg, 2004.

[33] M. Cai, A. Chervenak, and M. Frank. A Peer-to-Peer Replica Location Service Based on a Distributed Hash Table. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 56. IEEE Computer Society, 2004.

[34] J. Cao, S.A. Jarvis, S. Saini, and G.R. Nudd. Gridflow: Workflow Management for Grid Computing. 2003.

[35] J. Knobloch (Chair) and L. Robertson (Project Leader). LHC Computing Grid Technical Design Report. http://lcg.web.cern.ch/LCG/tdr/, March 2011.

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

[37] S. Che, J. Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *Symposium on Application Specific Processors, 2008. SASP 2008*, pages 101–107. IEEE, 2008.

[38] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel and Distributed Computing*, 63(5):597–610, 2003.

[39] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2011.

[40] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming Scientific and Distributed Workflow with Triana Services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[41] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley Longman, 2005.

[42] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON), 2006.

[43] V. Curcin and M. Ghanem. Scientific Workflow Systems - Can One Size Fit All? In *Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International*, pages 1–9. IEEE, 2009.

[44] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 2002.

[45] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[46] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-Science: An Overview of Workflow System Features and Capabilities. *Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications*, 25(5):528–540, 2009.

[47] 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.

[48] T. Dierks and C. Allen. The TLS Protocol Version 1.0. http://www.ietf.org/rfc/rfc2246.txt, March 2011.

[49] K. Dowd. *High Performance Computing*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 1993.

[50] E. Eckerson. Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications. *Open Information Systems*, 10(1):3–22, 1995.

[51] W. Allcock (editor). GridFTP: Protocol Extensions to FTP for the Grid. http://www.ogf.org/documents/GFD.20.pdf, March 2011.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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*, 26(2):245–256, 2010.

[57] 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.

[58] 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 *Applied Parallel Computing: State of the Art in Scientific Computing, Lecture Notes in Computer Science, vol. 6127*. Springer-Verlag, to appear, 2011.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] E. Elmroth and J. Tordsson. A Standards-based Grid Resource Brokering Service Supporting Advance Reservations, Coallocation and Cross-Grid Interoperability. *Concurrency and Computation: Practice and Experience*, 21(18):2298–2335, 2009.

[64] T. Erl. SOA: Principles of Service Design. 2007.

[65] T. Erl. SOA Design Patterns. *The Prentice Hall Service-Oriented Computing Series From Thomas Erl*, 2009.

[66] T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid Workflow Applications with AGWL: an Abstract Grid Workflow Language. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, pages 676–685. IEEE, 2005.

[67] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU Cluster for High Performance Computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2004.

[68] D. Feitelson. Parallel Workloads Archive. *URL http://www. cs. huji. ac. il/labs/parallel/workload*.

[69] A.J. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, C. Zsigri, R. Sirvent, J. Guitart, R.M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S.K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forgó, T. Sharif, and C. Sheridan. OPTIMIS: a Holistic Approach to Cloud Service Provisioning. *Future Generation Computer Systems*, accepted, 2010.

[70] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol–HTTP/1.1, 1999.

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

[72] J. Fontán, T. Vázquez, L. Gonzalez, RS Montero, and IM Llorente. OpenNEbula: The Open Source Virtual Machine Manager for Cluster Computing. In *Open Source Grid and Cluster Software Conference*, 2008.

[73] W. Ford, P. Hallam-Baker, B. Fox, B. Dillaway, B. LaMacchia, J. Epstein, and J. Lapp. XML Key Management Specification (XKMS). *Retrieved from: http://www. w3. org/TR/2001/NOTE-xkms-20010330*, 2001.

[74] I. Foster. What is the Grid? A Three Point Checklist. *GRID today*, 1(6):22–25, 2002.

[75] 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, Lecture Notes in Computer Science 3779*, pages 2–13. Springer-Verlag, 2005.

[76] I. Foster and C. Kesselman (editors). *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.

[77] 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, March 2011.

[78] 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, March 2011.

[79] 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, March 2011.

[80] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[81] 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.

[82] I. Foster and S. Tuecke. Describing the Elephant: The Different Faces of IT as Service. *ACM Queue*, 3(6):26–34, 2005.

[83] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-degree Compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. Ieee, 2009.

[84] D. Gannon, R. Bramley, G. Fox, S. Smallen, A. Rossi, R. Ananthakrishnan, F. Bertrand, K. Chiu, M. Farrellee, M. Govindaraju, S. Krishnan, L. Ramakrishnan, Y. Simmhan, A. Slominski, Y. Ma, C. Olariu, and N. Rey-Cenvaz. Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. *Cluster Computing*, 5(3):325–336, 2002.

[85] 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*, 20(18):2089–2122, 2008.

[86] Globus. An "Ecosystem" of Grid Components. http://www.globus.org/grid_software/ecology.php. March 2011.

[87] Globus. http://www.globus.org. March 2011.

[88] S. Godik, A. Anderson, B. Parducci, P. Humenn, and S. Vajjhala. OASIS eXtensible Access Control Markup Language (XACML). 2002.

[89] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. Von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications. High-Level Application Programming on the Grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006.

[90] 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, March 2011.

[91] GridBench: A Tool For Benchmarking Grids. http://grid.ucy.ac.cy/gridbench/. March 2011.

[92] 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/, March 2011.

[93] P. Hallam-Baker. Security Assertions Markup Language. *May*, 14:1–24, 2001.

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

[95] P. Hasselmeyer, H. Mersch, B. Koller, H.-N. Quyen, L. Schubert, and Ph. Wieder. Implementing an SLA Negotiation Framework. In *Exploiting the Knowledge Economy: Issues, Applications, Case Studies (eChallenges 2007)*, 2007.

[96] F. Heine, M. Hovestadt, and O. Kao. Towards Ontology-Driven P2P Grid Resource Discovery. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 76–83. IEEE, 2005.

[97] M. Herlihy and J.E.B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, page 300. ACM, 1993.

[98] H.P. Hofstee. Power Efficient Processor Architecture and the Cell processor. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 258–262. IEEE, 2005.

[99] D. Hollingsworth. Workflow Management Coalition: The Workflow Reference Model. *Workflow Management Coalition*, 1993.

[100] 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.

[101] A. Iosup, C. Dumitrescu, D. Epema, H. Li, and L. Wolters. How Are Real Grids Used? The Analysis of Four Grid Traces And Its Implications. In *Grid Computing, 7th IEEE/ACM International Conference on*, pages 262–269. IEEE, 2007.

[102] A. Iosup, D. Epema, C. Franke, A. Papaspyrou, L. Schley, B. Song, and R. Yahyapour. On Grid Performance Evaluation using Synthetic Workloads. In *Job Scheduling Strategies for Parallel Processing*, pages 232–255. Springer, 2007.

[103] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D.H.J. Epema. The Grid Workloads Archive. *Future Generation Computer Systems*, 24(7):672–686, 2008.

[104] A. Iosup, O. Sonmez, S. Anoep, and D. Epema. The Performance of Bags-of-Tasks in Large-Scale Distributed Systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 97–108. ACM, 2008.

[105] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[106] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson. Privacy-Preserving P2P Data Sharing with OneSwarm. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, pages 111–122. ACM, 2010.

[107] K.R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H.J. Wasserman, and N.J. Wright. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. In *2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 159–168. IEEE, 2010.

[108] M. Jayawardena, C. Nettelblad, S.Z. Toor, P-O. Östberg, E. Elmroth, and S. Holmgren. A Grid-Enabled Problem Solving Environment for QTL Analysis in R. *In Proceedings of the 2nd International Conference on Bioinformatics and Computational Biology (BICoB)*, pages 202–209, 2010.

[109] S. Jha, A. Merzky, and G. Fox. Using Clouds to Provide Grids with Higher Levels of Abstraction and Explicit Support for Usage Modes. *Concurrency and Computation: Practice and Experience*, 21(8):1087–1108, 2009.

[110] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive Application-Performance Modeling in a Computational Grid Environment. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 71–80, 1999.

[111] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos. File-Sharing in the Internet: A Characterization of P2P Traffic in the Backbone. *University of California, Riverside, USA, Tech. Rep*, 2003.

[112] 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.

[113] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web Services Choreography Description Language Version 1.0. *W3C Working Draft*, 17:10–20041217, 2004.

[114] J. Kay and P. Lauder. A Fair Share Scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[115] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes. Sky Computing. *Internet Computing, IEEE*, 13(5):43–51, 2009.

[116] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[117] H. Kishimoto and J. Treadwell. Defining the Grid: A Roadmap for OGSA Standards. *Open Grid Services Architecture Working Group. http://www.ogf. org/documents/GFD*, 53:2007–05, 2005.

[118] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[119] D. Kondo, M. Taufer, C.L. Brooks, H. Casanova, and A.A. Chien. Characterizing and Evaluating Desktop Grids: An Empirical Study. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 26. IEEE, 2004.

[120] T. Koshida and S. Uemura. Automated Dynamic Invocation System for Web Service with a User-defined Data Type. In *Proceedings of the 2nd European Workshop on Object Orientation and Web Service (EOOWS 2004)*, pages 1–7, 2004.

[121] K. Krauter, R. Buyya, and M. Maheswaran. A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing. *Software: Practice and Experience*, 32(2):135–164, 2002.

[122] V. Kumar, A. Gupta, Army High Performance Computing Research Center, and University of Minnesota. Analyzing Scalability of Parallel Algorithms and Architectures. *Journal of parallel and distributed computing*, 22(3):379–391, 1994.

[123] S.M. Larson, C.D. Snow, M. Shirts, and V.S. Pande. Folding@ Home and Genome@ Home: Using Distributed Computing to Tackle Previously Intractable Problems in Computational Biology. 2009.

[124] Kung-Kiu Lau, Mario Ornaghi, and Zheng Wang. A Software Component Model and Its Preliminary Formalisation. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 1–21. Springer Berlin / Heidelberg, 2006.

[125] S. Lee, S.J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110. ACM, 2009.

[126] J. Lindemann and G. Sandberg. An Extendable GRID Application Portal. In *European Grid Conference (EGC)*. Springer Verlag, 2005.

[127] M. Linesch. Grid - Distributed Computing at Scale. http://www.ogf.org/documents/GFD.112.pdf, March 2011.

[128] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for High Throughput Computing. *SPEEDUP journal*, 11(1):36–40, 1997.

[129] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck. GPGPU: General-Purpose Computation on Graphics Hardware. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 208. ACM, 2006.

[130] A. Mackey. Windows Communication Foundation. *Introducing. NET 4.0*, pages 159–173, 2010.

[131] P. Marshall, K. Keahey, and T. Freeman. Improving Utilization of Infrastructure Clouds. 2011.

[132] G. Mateescu. Quality of Service on the Grid via Metascheduling with Resource Co-scheduling and Co-reservation. *Int. J. High Perf. Comput. Appl.*, 17(3):209–218, Fall 2003.

[133] V. Matena, B. Stearns, and L. Demichiel. *Applying Enterprise JavaBeans: Component-based Development for the J2EE Platform*. Pearson Education, 2003.

[134] U. Maurer. Modelling a Public-Key Infrastructure. *Lecture Notes in Computer Science*, 1146:325–350, 1996.

[135] P. Mell and T. Grance. The NIST Definition of Cloud Computing. *National Institute of Standards and Technology*, 2009. http://www.nist.gov/itl/cloud/upload/cloud-def-v15.pdf, March 2011.

[136] D.S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing, 2002.

[137] G. Moltó, V. Hernández, and J.M. Alonso. A Service-Oriented WSRF-based Architecture for Metascheduling on Computational Grids. *Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications*, 24(4):317–328, 2008.

[138] K. Moss. *Java Servlets*. McGraw-Hill, Inc. New York, NY, USA, 1999.

[139] 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.

[140] Nimbus. http://www.nimbusproject.org/. March 2011.

[141] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131. IEEE Computer Society, 2009.

[142] Nvidia. NVIDIA CUDA Programming Guide. *NVIDIA Corporation, Feb*, 2010.

[143] OASIS Open. OASIS: Advancing Open Standards for the Global Information Society. http://www.oasis-open.org/home/index.php, March 2011.

[144] OASIS Open. Reference Model for Service Oriented Architecture 1.0. http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf, March 2011.

[145] 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.

[146] T. Oinn, M. Greenwood, M. Addis, M. Nedim Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M.R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: Lessons in Creating a Workflow Environment for the Life Sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[147] A. Oram. Peer-to-Peer: Harnessing the Power of Disruptive Technologies. 2001.

[148] P-O. Östberg and E. Elmroth. Increasing Flexibility and Abstracting Complexity in Service-based Grid and Cloud Software. In

F. Leymann, I. Ivanov, M. van Sinderen, and B. Shishkov, editors, *Proceedings of CLOSER 2011 - International Conference on Cloud Computing and Services Science*. SciTePress. Preprint available at http://www.cs.umu.se/ds/, accepted, 2011.

[149] P-O. Östberg and E. Elmroth. GJMF - A Composable Service-Oriented Grid Job Management Framework. Preprint available at http://www.cs.umu.se/ds/, submitted, 2010.

[150] P-O. Östberg and E. Elmroth. Impact of Service Overhead on Service-Oriented Grid Architectures. Preprint available at http://www.cs.umu.se/ds/, submitted, 2011.

[151] P-O. Östberg, D. Henriksson, and E. Elmroth. Decentralized, Scalable, Grid Fairshare Scheduling (FSGrid). Preprint available at http://www.cs.umu.se/ds/, submitted, 2011.

[152] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. An Early Performance Analysis of Cloud Computing Services for Scientific Computing. *Delft University of Technology, PDS-2008-006*, 2008.

[153] H. Overdick. The Resource-Oriented Architecture. In *2007 IEEE Congress on Services (Services 2007)*, pages 340–347, 2007.

[154] M.P. Papazoglou. *Web Services: Principles and Technology*. Pearson Education Limited, 2008.

[155] C. Pautasso and E. Wilde. Why is the Web Loosely Coupled?: A Multi-Faceted Metric for Service Design. In *Proceedings of the 18th international conference on World wide web*, pages 911–920. ACM, 2009.

[156] M. Perepletchikov, C. Ryan, and Z. Tari. The Impact of Service Cohesion on the Analyzability of Service-Oriented Software. *Services Computing, IEEE Transactions on*, 3(2):89–103, 2010.

[157] A.L.M. Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998.

[158] J. Postel et al. Transmission Control Protocol, 1981.

[159] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The BitTorrent P2P File-Sharing System: Measurements and Analysis. *Peer-to-Peer Systems IV*, pages 205–216, 2005.

[160] R.S. Pressman. Software Engineering: A Practitioner's Approach. *McGraw Hill, New York*, 2002.

[161] ProgrammableWeb - Mashups, APIs, and the Web as Platform. http://www.programmableweb.com/, March 2011.

[162] 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.

[163] 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.

[164] L. Ramakrishnan, K.R. Jackson, S. Canon, S. Cholia, and J. Shalf. Defining Future Platform Requirements for e-Science Clouds. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 101–106. ACM, 2010.

[165] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 140–146. IEEE, 2002.

[166] B. Randell, P. Lee, and P.C. Treleaven. Reliability Issues in Computing System Design. *ACM Computing Surveys (CSUR)*, 10(2):123–165, 1978.

[167] E. Rescorla. RFC2818: HTTP over TLS. *RFC Editor United States*, 2000.

[168] T. Rings, G. Caryer, J. Gallop, J. Grabowski, T. Kovacikova, S. Schulz, and I. Stokes-Rees. Grid and Cloud Computing: Opportunities for Integration with the Next Generation Network. *Journal of Grid Computing*, 7(3):375–393, 2009.

[169] M. Ripeanu and I. Foster. Peer-to-Peer Architecture Case Study: Gnutella Network. *University of Chicago, Chicago*, 2001.

[170] 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.

[171] L.F.G. Sarmenta. *Volunteer Computing*. PhD thesis, Citeseer, 2001.

[172] J.M. Schopf. Ten Actions When Grid Scheduling. In J. Nabrzyski, J.M. Schopf, and J. Węglarz, editors, *Grid Resource Management State of the art and future trends*, chapter 2. Kluwer Academic Publishers, 2004.

[173] 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.

[174] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, Inc. New York, NY, USA, 1997.

[175] M.P. Singh and M.N. Huhns. *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons Inc, 2005.

[176] W. Smith, I. Foster, and V. Taylor. Predicting Application Run Times Using Historical Information. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 1459*, pages 122–142, 1999.

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

[178] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of Backfilling Strategies for Parallel Job Scheduling. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pages 514–519. IEEE, 2002.

[179] C. Stefansen. SMAWL: A Small Workflow Language based on CCS. In *CAiSE Short Paper Proceedings*, volume 131, page 132. Citeseer, 2005.

[180] H. Stockinger. Defining the Grid: A Snapshot on the Current View. *The Journal of Supercomputing*, 42(1):3–17, 2007.

[181] L. Stout, M.A. Murphy, and S. Goasguen. Kestrel: an XMPP-based Framework for Many Task Computing Applications. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–6. ACM, 2009.

[182] 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.

[183] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2–4):323–356, 2005.

[184] The Grid Infrastructure Research & Development (GIRD) Project. Umeå University, Sweden. http://www.cs.umu.se/ds, March 2011.

[185] The Maui Cluster Scheduler. http://www.clusterresources.com/products/maui/, March 2011.

[186] The Microsoft Windows Azure Platform. http://www.microsoft.com/windowsazure/, March 2011.

[187] The RESTlet Framework for Java. http://www.restlet.org/, March 2011.

[188] The (United States) National Institute of Standards and Technology (NIST). http://www.nist.gov. March 2011.

[189] A. Tikotekar, G. Vallée, T. Naughton, H. Ong, C. Engelmann, S.L. Scott, and A.M. Filippi. Effects of Virtualization on a Scientific Application Running a Hyperspectral Radiative Transfer Code on Virtual Machines. In *Proceedings of the 2nd workshop on System-level virtualization for high performance computing*, pages 16–23. ACM, 2008.

[190] J. Treadwell. Open Grid Services Architecture Glossary of Terms. In *Global Grid Forum, Lemont, Illinois, USA, GFD-I*, volume 44, pages 2–2, 2005.

[191] S. Trimberger. *Field-Programmable Gate Array Technology*. Springer, 1994.

[192] P. Trunfio, D. Talia, H. Papadakis, P. Fragopoulou, M. Mordacchini, M. Pennanen, K. Popov, V. Vlassov, and S. Haridi. Peer-to-Peer Resource Discovery in Grids: Models and systems. *Future Generation Computer Systems*, 23(7):864–878, 2007.

[193] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling User Runtime Estimates. *The 11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), Lecture Notes in Computer Science 3834*, pages 1–35, 2005.

[194] W.M.P. Van Der Aalst and A.H.M. Ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[195] G. von Laszewski and M. Hategan. Workflow Concepts of the Java CoG Kit. *J. Grid Computing*, 3(3–4):239–258, 2005.

[196] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. *Lecture Notes in Computer Science*, pages 49–64, 1997.

[197] E. Walker. Benchmarking Amazon EC2 for High-Performance Scientific Computing. *USENIX Login*, 33(5):18–23, 2008.

[198] B. Walters. VMware Virtual Platform. *Linux Journal*, 1999(63es):6, 1999.

[199] V. Welch. Grid Security Infrastructure Message Specification. http://www.ogf.org/documents/GFD.78.pdf, March 2011.

[200] W.P.D.I.X.M.L. WfMC. XML Process Definition Language (XPDL), WfMC Standards. Technical report, WFMC-TC-1025, http://www.wfmc. org, 2001.

[201] R. Williams. Grids and the Virtual Observatory. *Grid Computing: Making the Global Infrastructure a Reality*, pages 837–858, 2003.

[202] S. Williams and C. Kindel. The Component Object Model: A Technical Overview. *Dr. Dobbs Journal*, 356:356–375, 1994.

[203] S. Yi, D. Kondo, and A. Andrzejak. Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 236–243. IEEE, 2010.

[204] Andy Yoo, Morris Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. Springer Berlin / Heidelberg, 2003.

[205] L. Youseff, M. Butrico, and D. Da Silva. Toward a Unified Ontology of Cloud Computing. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. IEEE, 2009.

[206] L. Youseff, R. Wolski, B. Gorda, and C. Krintz. Paravirtualization for HPC systems. In *Frontiers of High Performance Computing and Networking–ISPA 2006 Workshops*, pages 474–486. Springer, 2006.

[207] J. Yu and R. Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *ACM Sigmod Record*, 34(3):44–49, 2005.

[208] O. Zimmermann, J. Grundler, S. Tai, and F. Leymann. Architectural Decisions and Patterns for Transactional Workflows in SOA. *Service-Oriented Computing–ICSOC 2007*, pages 81–93, 2010.

[209] H. Zuse. *A Framework of Software Measurement*. de Gruyter, 1998.

I

Paper I

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.cs.umu.se/ds

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.

II

Paper II

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.cs.umu.se/ds

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.

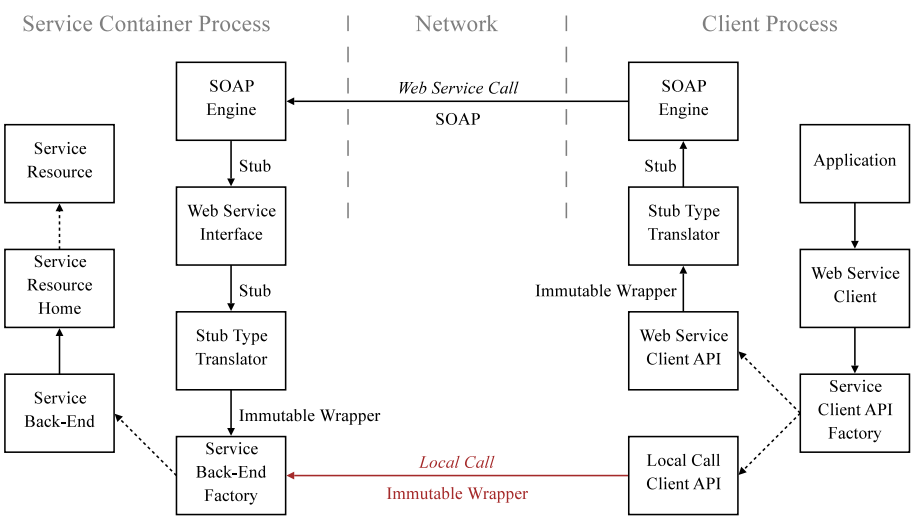


Figure 1. Illustration of local call optimizations for co-located services; dynamic resolution of service client APIs, back-ends and resources; and the use of immutable data wrappers.

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.

III

Paper III

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.cs.umu.se/ds

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.

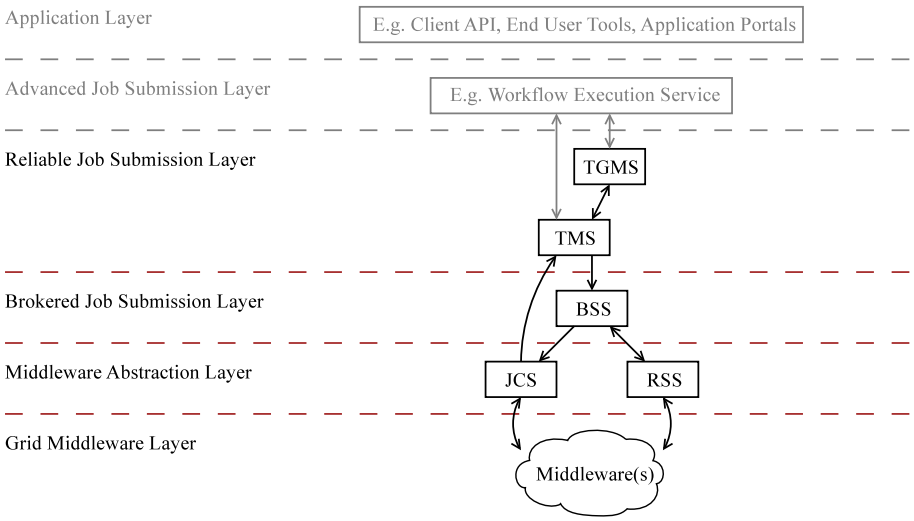


Figure 1. GJMF components mapped to their respective architectural layers.

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.

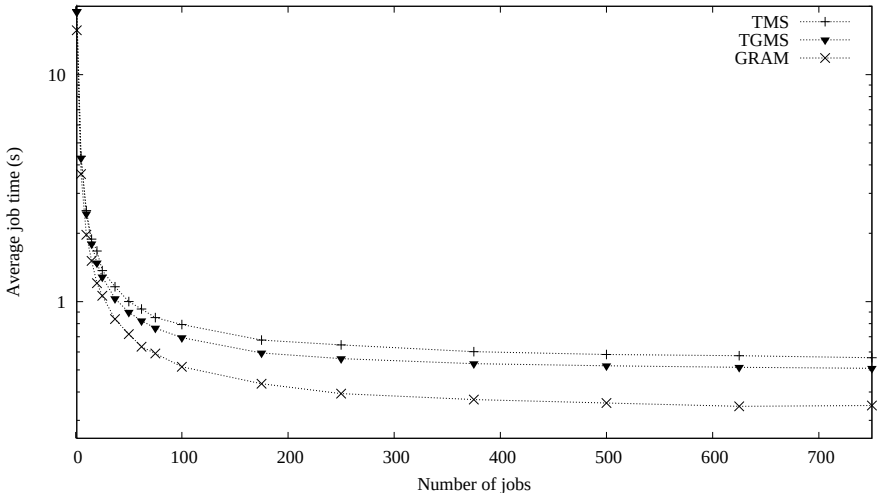


Figure 2. GRAM and GJMF job processing performance.

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.

IV

Paper IV

GJMF - A Composable Service-Oriented Grid Job Management Framework

Per-Olov Östberg and Erik Elmroth

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

Abstract: We investigate best practices for Grid software design and development, and propose a composable, loosely coupled Service-Oriented Architecture for Grid job management. The architecture focuses on providing a transparent Grid access model for concurrent use of multiple Grid middlewares and aims to decouple Grid applications from Grid middlewares and infrastructure. 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. A proof-of-concept implementation of the architecture is presented along with a discussion of architecture implementation details and trade-offs introduced by the service composition techniques used.

Key words: Grid computing, Grid job management, Grid ecosystem.

GJMF - A Composable Service-Oriented Grid Job Management Framework

Per-Olov Östberg and Erik Elmroth

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

Abstract

We investigate best practices for Grid software design and development, and propose a composable, loosely coupled Service-Oriented Architecture for Grid job management. The architecture focuses on providing a transparent Grid access model for concurrent use of multiple Grid middlewares and aims to decouple Grid applications from Grid middlewares and infrastructure. 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. A proof-of-concept implementation of the architecture is presented along with a discussion of architecture implementation details and trade-offs introduced by the service composition techniques used.

Key words: Grid computing, Grid job management, Grid ecosystem

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

1. Introduction

In this work, we extend the software design methodologies of [17, 19, 21], and investigate best practices for Grid software design and development in the context of Grid job management. As a result, we propose a composable Service-Oriented Architecture (SOA) for Grid job management constituted by layers of loosely coupled, composable, and replaceable Web Services, and illustrate the architecture with a prototype implementation called the *Grid Job Management Framework (GJMF)*.

There are many Grid applications and Grids in production use today, but as a result of the inherent complexity of Grid job management and the interoperability issues of current Grid middlewares, many Grid applications are tightly coupled to specific Grids and Grid middlewares. As 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, Grid job management tools should ideally virtualize the job management process to decouple Grid applications from Grid middlewares. The proposed architecture provides a set of middleware-agnostic job management interfaces that aim to allow applications to function seamlessly across Grid boundaries.

Grids are highly complex computational environments that exhibit high degrees of heterogeneity in resources and software, as well as in administrational practices. In these settings, software usability becomes a function of factors such as flexibility, scalability, and interoperability. To address these heterogeneity issues, we employ a model of software sustainability based on evolution of software in an ecosystem of Grid components (as described in Section 2). Here we identify a set of traits likely to promote survival for job management tools in a Grid ecosystem, and explore software design patterns and development methodologies that result in composable software that inhabit and interoperate between niches of such an ecosystem.

The architectural model used is a framework built on the principle of abstraction; functionality is stratified into layers of services that incrementally provide more advanced functionality through aggregation of underlying service capabilities. Designing software in abstractive layers facilitates component and system simplicity, robustness, automation, flexibility, and efficiency. This approach enables developers to build systems where individual components can be deployed stand-alone or as part of other architectures, while serving as part of the framework. Application developers can select parts of the framework to make use of based on current application needs, and

administrators are able to reconfigure framework deployments dynamically.

The contributed software prototype provides transparent and concurrent access to multiple Grid middlewares through a set of job management Web Services. All services of the prototype provide abstractive views of particular types of Grid job management functionality that help to decouple applications from Grid middlewares. Throughout the paper, facets of intended system behavior and implications of system design and architecture are discussed.

The rest of the paper is organized as follows: Section 2 discusses software requirements in the context of an ecosystem of Grid components, Section 3 outlines the proposed architectural model, and Section 4 describes technical details of the software prototype. In Section 5, the proposed architecture is discussed in more detail, and related and future work are presented in sections 6 and 7. Finally, the paper is concluded in Section 8.

2. 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 [19, 57] niches are defined by functionality required and provided by software components, end-users, and other Grid actors; and Grid applications and infrastructures are constituted by systems composed of components selected from the ecosystem. Here, software compete on evolutionary bases for ecosystem niches, where natural selection over time preserves components better at adapting to altered conditions. Adaptability is hence defined in terms of interoperability, 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, 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 [23], and

tend to result in tight coupling between Grid applications and Grid middlewares. To isolate Grid end-users and applications from details of underlying middlewares 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 an autonomous job management niche.

Furthermore, to promote interoperability components should build on 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 application 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 sustainability in the Grid ecosystem. 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 [21]. The idea to create composed and loosely coupled applications from aggregation of components is at the core of Service-Oriented Architecture (SOA) [49] methodology.

In the proposed architecture, components are realized as Web Services that contain customization points where third party plug-ins can be used to alter or augment system and component behavior. To promote deployment flexibility, architecture composition as well as component customization, can be dynamically altered via service configurations as described in [21]. Additionally, individual components of the architecture can be used as stand-alone services, or in other composed architectures, while concurrently serving as part of the architecture 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 [21]. This approach also provides a model for dis-

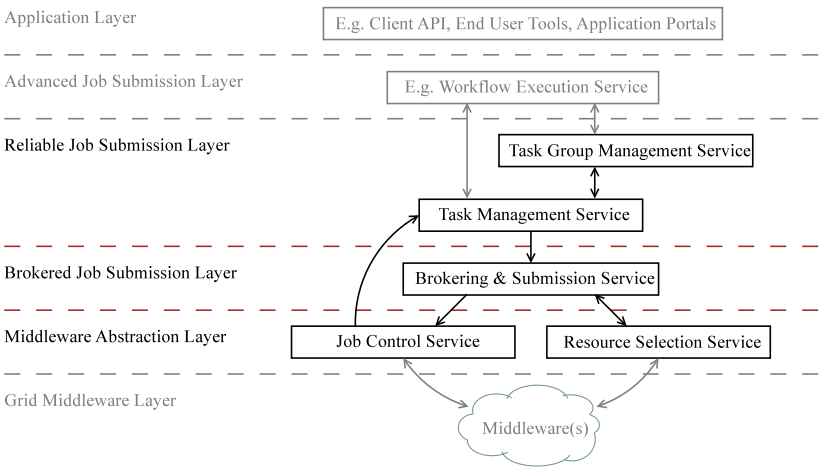


Figure 1: The proposed framework architecture. Services organized in hierarchical layers of functionality. Within the framework services communicate hierarchically, service clients are not restricted to this invocation pattern.

tributed software reuse, both on component and code level, and facilitates integration software development with a minimum of intrusion into existing systems [20]. Providing small, single-purpose components reduces component complexity and facilitates adaptation to standardization efforts [21].

The architectural model used has previously been briefly introduced in [17], and various aspects of the software development model are discussed in [19, 20, 21]. The software development model used in this work is a product of work in the Grid Infrastructure Research & Development (GIRD) multiproject [58] and is documented in [19, 21]. The models favor architectures built on principles of flexibility, robustness, and adaptability; and aim to produce software well adjusted for interoperability and use in the Grid ecosystem [57].

3.1. Architecture Layers

As illustrated in Figure 1, the framework architecture is divided into six layers of functionality, where each layer builds on lower layers and provides aggregated functionality to service clients. Stratification of the architecture into hierarchical layers provides several benefits, including

- Simplicity. By hierarchically segmenting the functionality of the framework, components can be kept task-oriented and implemented as small,

single-purpose modules that delegate functionality requirements to lower layers. Hierarchical ordering of the architecture also allows simplistic client-server models of communication between components.

- Robustness. Simplicity in implementation reduces complexity and increases robustness. Division of functionality into layers and definition of interfaces between components introduce natural failure management points, simplifies failure detection, and allows reactive failure recovery through, e.g., resubmission of tasks.

- Automation. Stratification of the architecture into a hierarchy of components allows abstraction of functionality in lower layers and facilitates introduction of automation of framework functionality in higher layers.

- Flexibility. Exposing individual components of the framework as services offers clients the choice of what levels of control and automation to make use of. Segmentation of the system into services also increases system deployment flexibility.

- Efficiency. Segmentation of system functionality into distributed components introduces communication overhead and synchronization issues. Keeping component segmentation hierarchical minimizes synchronization issues and facilitates parallel (pipeline) processing of framework tasks (masks communication overhead).

Stratification of the framework is based on identification of four fundamental types of Grid job management functionality: abstractive job control, job brokering, reliable job submission, and advanced job management. For each layer a core functionality set is identified and implemented as autonomous services in the proof-of-concept prototype (see Figure 1).

3.1.1. Grid Middleware Layer

In the architecture, the Grid middleware layer houses all software components concerned with abstraction of native job management capabilities. This includes traditional Grid middlewares abstracting batch systems, e.g., the Globus middleware (GT4) [33] abstracting the Portable Batch System (PBS) [40], standardized job dispatchment services, e.g., the OGSA BES [27], and desktop Grid approaches such as the Berkeley Open Infrastructure for Network Computing (BOINC) [7] and Condor [56] abstracting use of CPU

cycle scavenging and volunteer computing resources. Components in the Grid middleware layer are not part of the framework but are essential in providing native job submission, control, and monitoring capabilities to the framework.

3.1.2. Middleware Abstraction Layer

The purpose of the middleware abstraction layer is to abstract details of Grid middleware components and provide a unified Grid middleware interface. 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.

Services in the middleware abstraction layer are expected to be utilized by other softwares, e.g., third party job brokers and low-level job management tools, rather than end-users. Currently, the middleware abstraction layer contains services for targeted job submission and control, information system interfaces, and services concerned with translation of job descriptions. 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 maintain a unified job control interface.

3.1.3. 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. Services in this layer provides automated matching of jobs to computational resources, and are expected to primarily be used by simple job submission interfaces, e.g., command-line job submission tools.

Job submission performed by services in the brokered job submission 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) submitting 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.

3.1.4. 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, and are intended to be utilized (through job management tools) by end-users. The term reliable job submission refers to the ability of these services to 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.

3.1.5. Advanced Job Submission Layer

The advanced job submission layer is aimed towards advanced mechanisms for job management, e.g., workflow tools, Grid application components, and portal interfaces that by functionality requirements are coupled to components of the framework. Services of the advanced job submission layer are intended to aggregate services of the reliable job submission layer, provide aggregated job management contexts, and function as integration bridges and customized service interfaces to the framework. A number of functionality sets for advanced job management are identified and under investigation (see Section 7) for inclusion in the prototype implementation of the framework, e.g., management of data and sets of interdependent jobs.

3.1.6. 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, software in the application layer are not 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

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 are implemented and typically deployed using the Globus Toolkit [25], 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) [13], the Web Service Resource Framework (WSRF) [26], and the Job Submission Description Language (JSDL) [9]. The GJMF also conforms to the design of the Open Grid Service Architecture (OGSA) [28] and builds on the design of the OGSA Basic Execution Service (OGSA BES) [27], and the OGSA Resource Selection Services (OGSA RSS) [28].

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 [36], and expose these through service interfaces and WS-ResourceProperties [35], allowing clients to inspect state both explicitly and through subscription to WS-BaseNotifications [34].

4.1. Job Definitions

To facilitate abstraction of Gird middleware functionality, 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. Job descriptions typically consist of a JSDL file specifying an executable program, program parameters, computational resource references, file staging information, and optional JSDL annotations containing job processing hints.

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

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

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. Each service is capable of using multiple instances of other services, and supports a model of user-level isolation where unique service instances are created for each service user. Worker threads and contexts within individual services are shared among service instances and competition for resources between service instances occur as if services are 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 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. Within the GJMF, the LAS is used to record state transitions and job submission and processing information.

The LAS utilizes 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 LAS maintains internal storage queues and resource serialization mechanisms to minimize overhead for use of the service and provide an asynchronous log storage model. Customizable database support is provided through use of database accessor plug-in modules. Database accessors can be provided by third parties and boiler-plate solutions for accessor plug-ins supporting SQL and JDBC are provided. Currently, the LAS provides accessors for MySQL, PostgreSQL, and Apache Derby. Unlike other services of the GJMF, use of the LAS is optional and not required for any part of the GJMF to function. Both the LAS and LAS database accessors can 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. Within the framework, the JTS is typically used by the Job Control Service to provide translations of JSDL to native Grid middleware job description formats. To service clients, the JTS can 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).

The JTS employs a modularized architecture where translation semantics are provided by plug-ins. Support for new languages can be added by third parties without modification of the framework. Currently, the JTS supports translation between JSDL [9] and Globus Toolkit 4 Resource Specification Language (GT4 RSL) [25], NorduGrid Extended Resource Specification Language (XRSL) [16], and a custom dialect of XRSL presented in [20]. Translations of job descriptions are made based on the context of the job description representation created. Typically this means that existing job descriptions are queried 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)

The purpose of the Job Control Service (JCS) is to provide a uniform and middleware-transparent job submission and control interface. The JCS defines a set of generic job control functionality, as well as a job state model (illustrated in Figure 4c), 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 for job submission, and by the Task Management Service for job monitoring and control. Service clients can use the JCS directly as a targeted Grid job submission and control tool.

Internally, the JCS maintains a job control component that coordinates execution and monitoring of native Grid 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 [16], GT4 [33] middlewares, and Condor [56]. 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. The functionality of the JCS can also be altered by providing processing hints 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 [20]) or GT4 Globus Resource Allocation Manager (WS-GRAM) parameters for Condor-G [30] submissions. As these types of processing hints are completely orthogonal to standard service behavior, i.e. does not affect processing of other jobs or service functionality, they can be used to temporarily alter service behavior for a specific job without alteration of framework composition or configuration.

4.2.4. Resource Selection Service (RSS)

Built on the OGSA RSS [28] model, the GJMF Resource Selection Service (RSS) provides a service interface for resource brokering in Grid environments. Within the GJMF, the RSS is used by the Brokering & Submission Service as an execution planning and brokering tool. Service clients can use the RSS directly for job to resource matching or to inspect resource availability.

Internally, the RSS maintains a resource selector component that coordinates brokering of tasks to computational resources. Middleware-specific information system accessors are used to abstract middleware information systems and provide translations of middleware-specific record formats to an internal RSS format. The RSS also maintains mechanisms for resource information retrieval and caching, information system monitoring, and customization mechanisms that allow third parties to develop plug-ins to support new information sources.

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 sequentially submits jobs to each suitable computational resource identified (as ranked by the RSS) until a resource accepts the job or the list of resources is exhausted. Beyond this behavior, BSS failures are considered permanent.

Within the GJMF, the BSS is used by the Task Management Service for task submissions. Service clients can use the BSS directly as a 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

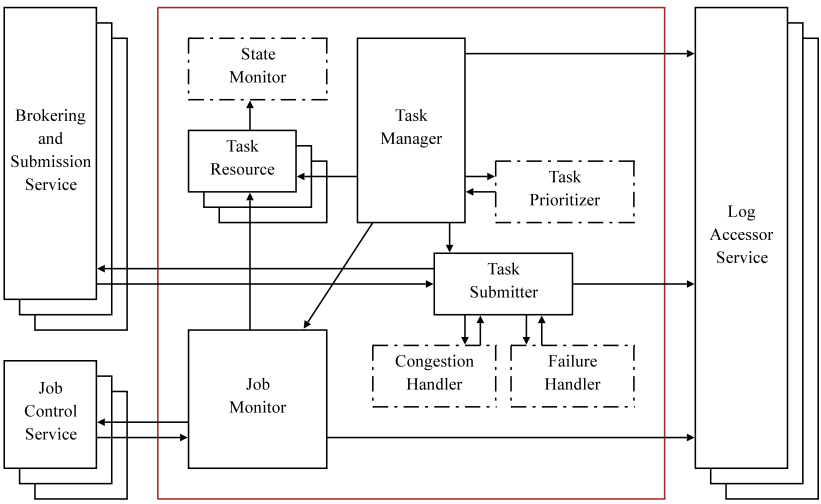


Figure 2: TMS architecture. The TMS architecture is typical for all GJMF services; interaction with other services are wrapped in access modules, and customization points are exposed through configurable plug-in structures. Customization points are illustrated using dotted lines.

assumed to be reported by the enacting JCS or detected and handled by service clients.

Internally, the BSS maintains components for job brokering and job submission. The job broker component interacts with RSSs to retrieve job execution plans. The job submission component is used by the job broker and interfaces with JCSs to submit jobs. Both components make use of LASs for log storage and are 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)

The Task Management Service (TMS) provides an interface for automated and fault-tolerant task management, and defines a task state model (illustrated in Figure 4b). 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 are made. Within the GJMF, the TMS is used by the Task Group Management Service for management of individual tasks.

Internally, the TMS maintains components for task management, task submission, and job monitoring, as illustrated in Figure 2. Task state is maintained and exposed through WS-ResourceProperties by task resources. The internal mechanisms of the TMS can be customized via configuration 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 task 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)

The Task Group Management Service (TGMS) provides an interface for automated management of groups of (mutually independent) jobs and tasks, and defines a task group state model (illustrated in Figure 4a). The TGMS is intended to be used by service clients and more complex task management systems, e.g., workflow and parameter sweep applications. The TGMS is currently not used by other services in the GJMF.

Internally, the TGMS maintains components for task group management, coordination of task and task group processing, and task monitoring. Task group state is maintained and exposed as WS-ResourceProperties by task group resources. The TGMS maintains state contexts for task groups, employs user-exclusive submission queues for both task groups and tasks, and 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 management, the TGMS does not contain a task execution failure handler. Task execution failures are by the TGMS assumed permanent, no error recovery or failover actions are taken by the TGMS. Task submission failures are considered temporary and result in task submission rescheduling.

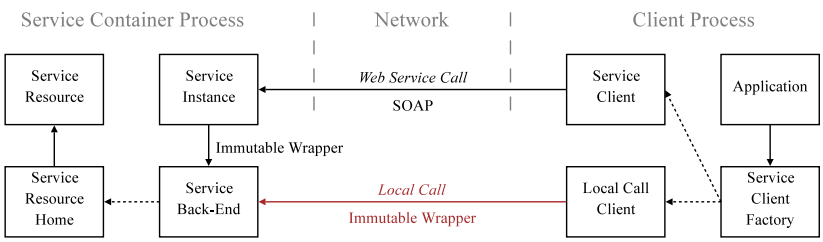


Figure 3: The GJMF service structure. The GJMF common library provides boiler-plate solutions for service instantiation, service back-end implementation, resource management, and client APIs. The GJMF client API abstracts use of the service invocation optimizations through use of service client factories and service back-ends. Dynamic invocation patterns illustrated using dotted lines.

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 are not further processed until explicitly resumed. Tasks in a suspended task group that are submitted to a TMS will be processed if possible, but no new task submissions are made until the task group is resumed. The TGMS can be configured to use a congestion handler to customize back-off behaviors in Grid congestion scenarios; 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. Common Library

The GJMF common library is a service development utility library that provides a common type set and boiler-plate solutions for, e.g., local call optimizations, service stubs, notification subscriptions, credentials delegation, security contexts, worker threads, state management, service client APIs, dynamic configuration, and resource 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 3, and details separation of service interface implementation from service back-end implementations, and service clients from service client factories. Service client factories are ex-

posed 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 [21].

The service development framework, in concert with the GJMF client API, handles all service invocation mechanics, including data type marshalling, service instantiation, and notification management. The framework 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, i.e. allowing 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, and also addressed in [21], is made fully transparent to service clients by the service structure of the common library and the client API. As also described in [21], 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.

4.2.9. 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 development. All GJMF functionality is accessible through both the GJMF services and the GJMF client API.

5. Architecture Discussion

The hierarchical architecture of the GJMF is intended to provide clients a versatile and flexible set of job management interfaces that offer an increasing range of automation of the job management process without sacrificing user

control. Services in lower layers offer fine-grained job management interfaces with explicit control, while services in higher layers attempt to automate the job management process and offer control through configuration of behavior and optional customization point modules.

To meet the flexibility and adaptability requirements discussed in Section 2, we extend the software development model previously presented in [21]. Key approaches in this model include use of Service-Oriented Architectures (SOAs) [49], design patterns, refactorization methods, and techniques to improve software adaptability such as dynamic configuration and customization points. Software is developed in Java and the Globus Toolkit [25] is employed to produce Grid-enabled Web Services compatible with established Grid security models.

5.1. Invocation Patterns

Services of the GJMF support two basic modes of method invocation; sequential and batch. In sequential invocations, service requests are transmitted in dedicated messages. In batch invocations, sets of service requests are bundled and transmitted in compound messages. Batch invocations allow service clients to, e.g., submit sets of tasks in single requests, significantly reducing service invocation makespan, network bandwidth requirements, and service invocation memory footprints. To simplify service invocation semantics, sets of requests sent using batch invocations are processed as transactions. If, e.g., a job submission in a batch request fails, other job submissions in the batch are canceled and rolled back.

When service clients are codeployed with GJMF services, i.e. deployed in the same service container as the GJMF, service invocations are by default routed through the GJMF local call optimizations. These mechanisms exploit that services hosted in the same container share the same process space, i.e. operate in the same Java Virtual Machine (JVM), and allow service clients to directly invoke methods in service implementation back-ends. By bypassing message serializations, this greatly reduces service invocation makespans and memory footprints, allow more fine-grained service communication models, and promotes a model of service aggregation where constituent services can function as local objects in aggregated services [21]. 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 lost. All services of the GJMF are designed to be distributed in separate service deployments, but are for performance reasons recommended to be codeployed.

As GJMF services can at any time be invoked directly by service clients, service invocation patterns are hard to predict and likely to vary over time. For this, as well as for performance reasons, all interservice communication is routed through the GJMF client API, which allows invocation modes and service communication optimizations to be ubiquitous and completely transparent.

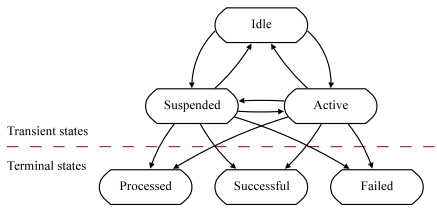
5.2. Deployment Scenarios

The construction of the framework as a loosely coupled SOA with invocation optimizations allows the framework great freedom in deployment. Envisioned usage scenarios for the framework include

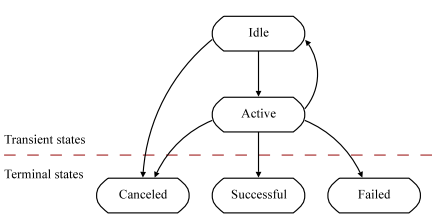
- Running the framework on Grid gateways to act as middleware-agnostic job submission interfaces.
- Running the framework on client computers to act as convenient personal Grid job management tools.
- Running multiple instances of the framework to provide partitioning and load balancing of large job submission queues and multiple Grids.
- Running multiple instances of the framework with different configurations to provide alternative job submission behaviors.

As natural overlaps between these usage scenarios exist, 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 combinations of multiple deployments of the framework, on top of multiple Grid middlewares and resource managers. To meet advanced application requirements, e.g., workflow enactment or parameter sweeps, the GJMF is expected to be utilized in combination with high-level tools such as the Grid Workflow Execution Engine (GWEE) [18].

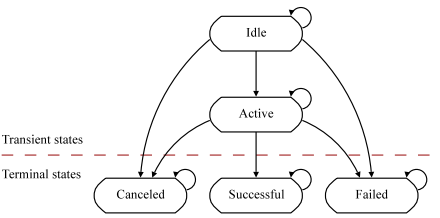
The GJMF deployment flexibility allows the framework to be employed in a number of computational settings, including high-performance computing (HPC) (requiring support from underlying middlewares for some functionality, e.g., MPI job execution), high-throughput computing (HTC), as well as the more recently defined many-task computing (MTC) [53] paradigm. In MTC, focus is placed on enactment of loosely coupled applications constituted by large numbers of short-lived, data intensive, heterogeneous tasks



(a) GJMF task group state model.



(b) GJMF task state model.



(c) GJMF job state model.

Figure 4: 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 based on the state model of the OGSA BES [27]. Recurring states in the GJMF job state model abstract state information from more fine-grained Grid middleware job state models.

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 are inherently complex. The GJMF employs a hierarchical event-driven model for distributed state updates where services hosting job description resources are responsible for propagating state updates to clients. The use of a hierarchical event-driven service state model allows service failure detection and recovery to be reactive and adaptive rather than predictive. Keeping individual service contexts simple (through delegation of functionality to lower layers) and reactive facilitates service implementation robustness.

The GJMF state update mechanisms are built on WS-BaseNotifications messages. To compensate for state notifications being dropped due to network failures or service container loads, all services implement a state monitoring mechanism that regularly polls for missing notifications. As both state

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.

coordination and monitoring mechanisms are encapsulated in the framework service structures and client APIs, service implementations consider state delivery transparent and reliable.

As illustrated in Figure 4, each type of GJMF job definition has a corresponding state model that drives processing of jobs, tasks, and task groups in the GJMF. In this model, jobs, tasks, and task groups are referred to as work units, and assigned individual work unit contexts that are exposed to clients through service interfaces and WS-ResourceProperties. A brief summary of state interpretations for GJMF work units is given in Table 1.

5.4. Data Management

To maintain middleware transparency, the GJMF does not actively participate in data transfers. The GJMF assumes that data files are available and can be transfered to and from computational resources by enacting Grid middlewares via file transfer mechanisms chosen by the middlewares. File staging information is conveyed as part of job descriptions, typically in the form of GridFTP [15] URLs, and JSDL annotations can be used to provide job brokering hints related to storage requirements for computational elements. 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.

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 investigation, see Section 7.

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 Grid environments are likely to contain multiple concurrent job submission and management systems, such brokering components will always operate on information that is to some extent deprecated [23].

The RSS is limited to provide computational resource recommendations (execution plans) without feedback from service clients. 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 to maintain an internal cache of prior execution plans and update resource load weights through speculation. As the RSS enforces user-level isolation of service capabilities, a unique cache would be created for each user and restricted to contain recommendations for that user.

To improve quality of resource brokering, it would be possible to interface the RSS with Grid accounting and load balancing systems, e.g., the SweGrid Accounting System (SGAS) [32], as well as provide the RSS with feedback from the JCS or job submission systems such as the Job Submission Service (JSS) [23]. To reduce system complexity and maintain a clean separation of concerns, the RSS does not implement speculative resource load prediction, but it does offer 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 [22]. Evaluation of RSS

brokering performance and quality of execution plans is considered out of scope for this work.

5.6. Security

The GJMF employs the Grid Security Infrastructure (GSI) [29] security model provided by the Globus Toolkit [25], and can be configured to use 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 client identities are verified for all service invocations from external clients. For service invocations using GJMF local call optimizations, i.e. from codeployed service clients, credential proxies are accepted without verification of caller identity. This relaxation of authentication is done for performance reasons and is deemed acceptable for situations where mutual trust is established between services and deployment environments. GJMF local call optimizations can be disabled or replaced with Apache Axis local call optimizations for situations where verification of caller identity is required to be enforced.

All types of job definitions, including task groups, are upon submission 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 GJMF service, a separate service implementation (back-end) is instantiated. This enforces user-level isolation of service capabilities and provides 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 construction of job submission proxies, a requirement in, e.g., Grid portal construction [20], clients may specify separate execution credentials for tasks. GJMF resources (including LAS logs) resulting from such submissions are owned by the the execution credentials identity. Authentication of caller credentials is performed in GJMF service invocations, but authentication of execution credentials may be deferred (by use of local call optimizations) until Grid middleware job submissions.

5.7. Performance Characteristics

While a detailed performance analysis is out of scope for this contribution, brief remarks regarding performance characteristics of the framework are included to motivate use of the framework. Implementation of the framework as a network of Web Services introduces overhead for service invocation, which can be mediated using invocation optimization mechanisms and masked by parallel processing of framework tasks.

Web Service invocation throughput is typically limited by two factors; service computational complexity and service invocation overhead. In the GJMF, services are designed to have low computational complexity, and framework invocation overhead is addressed by use of service invocation optimization mechanisms. The BSS and JCS are designed to make synchronous invocations to underlying systems, and are limited by the job submission performance of Grid middlewares. Other services of the GJMF employ asynchronous invocation processing models and do not suffer this limitation. To address invocation overhead issues, the GJMF employs two invocation optimization mechanisms; local calls and batch invocations.

To analyze performance of the framework prototype and evaluate impact of architecture design on system overhead, extensive job submission and processing performance tests have been undertaken. In tests, framework performance regarding service invocation capability (with and without invocation optimizations), job submission throughput, and job processing throughput (under ideal and realistic settings) have been quantified against baseline performance measurements of underlying Grid middlewares. All tests have been performed under realistic operational settings, using WS-Security SecureConversation and production use deployments of GT4, Torque, and Maui.

Results indicate that total system overhead can be subdivided into three components: submission overhead, processing overhead, and job execution overhead. Submission overhead is incurred during submission of tasks to the framework, and typically ranges from 1 to 30 seconds per invocation depending on factors such as class loading overhead (for Java-based service clients), encryption overhead (varies with security model), and transmission overhead (mainly consisting of network latency and XML message serialization overhead). Impact of submission overhead can be mediated by use of batch invocation modes (that reduce the number of invocations) and use of asynchronous processing mechanisms (that reduce service response time). In tests (using large task groups and full WS-Security), effective submission overhead contributions are reduced to fractions of seconds per job.

Processing overhead consists of service computation and invocation overhead. When framework services are co-deployed, invocation overhead can be reduced to the order of milliseconds by use of local call optimizations. Services using synchronous processing models (BSS and JCS) are limited by performance of underlying systems and produce the largest contributions to total system overhead. Concurrent use of local call optimizations and asynchronous invocation processing models reduce total service invocation overhead contributions to the order of milliseconds.

Job execution overhead is here defined to include factors such as middleware submission overhead, scheduling overhead, staging and execution makespan, i.e. external factors outside of framework control. For realistic scenarios, job execution overhead is typically several orders of magnitude larger than GJMF contributions to system overhead. Performance of GJMF local call optimizations are tested and found competitive when compared to similar mechanisms in GT4 and Apache Axis.

In summary, total overhead introduced by use of the GJMF for, e.g., job submission, brokering, and monitoring, can be limited to less than one second per job for realistic usage scenarios. Impact of framework overhead on system performance is mitigated by mechanisms such as asynchronous and parallel processing of tasks, batch invocation modes, and local call optimizations. For further and detailed performance evaluation information, see [51].

6. 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 [27], and OGSA RSS [28] 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 [26], WSDL [13], SOAP [38], and GSI [5] have outlined the architecture communication models, and Grid middleware and resource manager systems such as the Globus middleware [33], NorduGrid ARC [16], Condor [56], 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) [28], the Open Middleware Infrastructure Institute (OMII Europe) [50], and Grid Interoperation/Interoperability Now (GIN) [37], as well as contributions such as

[10, 44, 52, 60] have provided perspective, insight, and inspiration to architecture interoperability design.

The Grid resource management system survey presented in [45] 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.

Job management systems exhibiting similarities in design or intended use have been identified, and include

The GridWay Metascheduler [39], a framework for adaptive scheduling and execution of Grid jobs. Like the GJMF, GridWay builds on the Globus Toolkit and offers an abstractive 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 consistent performance levels and relies on Grid applications and middlewares to handle checkpointing and application preemption issues.

The Falkon [54] framework provides a fast and lightweight task execution framework focused on task throughput and efficiency. 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-transparency to service clients.

The Minimum intrusion Grid (MIG) [42] is a framework aimed at pro-

viding 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 abstracts use of Grid middlewares.

The Imperial College e-Science Networked Infrastructure (ICENI) [31] is a composable OGSA Grid middleware implementation based on Jini. 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 [63] 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) [23] is a resource brokering and job submission service developed in the GIRD [58] 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-agnostic 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, translation of job descriptions, and incorporation of more recent standardization efforts. Work on the GJMF 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 [55] 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 differences between eNANOS and the GJMF lie in the flexibility of the GJMF architecture, which provides dynamic composition of the framework and additional levels of abstraction of job management functionality. The GJMF also builds on standardization efforts such as JSDL, WSRF, and the OGSA BES.

The Community Scheduler Framework (CSF4) [62] 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, 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 toolkit for Grid application development. The fundamental ideas behind the GAT and the GJMF are similar, both aim to decouple Grid applications from middlewares by providing middleware-agnostic 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 job management functionality, and relying on underlying middlewares for job control and file staging capabilities.

GridSAM [46] 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 managers 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) [61] 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 framework, job description translation functionality,

job monitoring capabilities, and multiple job submission and control mechanisms.

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 offers customization points for these types of mechanisms in the RSS, and provides a flexible architecture that can incorporate usage pattern-specific adaptations with only local modifications.

The Gridbus [59] 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 reconfigure framework deployments.

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 [48], 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.

The Grid Meta-Broker Service (GMBS) [43] addresses Grid job management in a way similar to the GJMF. The goals of both projects include to provide interoperability between Grids through automation and virtualization of job management without modification of Grid middleware deployments. The GMBS defines an architecture for interoperability on a meta-broker level, defines languages for meta-broker scheduling and broker description, and performs brokering of jobs to resource brokers in a way similar to how the GJMF performs brokering of jobs to resources. Both architectures expose functionality through WSDL-based Web Service interfaces, build on standardization efforts, define translation components for JSDL documents, and provide broker- / resource-specific invocation components. While the GJMF and GMBS are similar in design, concept, and goals, they operate on different levels in the Grid job management stack. The GMBS provides Grid interoperability through high-level meta-brokering, whereas the GJMF defines a flexible architecture that provides interfaces to multiple types of job management functionality. With modification, the GMBS could make use of the lower layers of the GJMF for resource brokering, and the higher layers of the GJMF could be adapted to make use of the GMBS for job management.

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. Furthermore, there exists a number of workflow-based approaches to Grid job management, e.g., ASKALON [24], Pegasus [14], and GWEE [18]. These have been omitted here as the scope of this work is restricted to generic job management architectures. Naturally, with modifications, most of these could make use of the GJMF for middleware-transparent Grid access.

Finally, a few slightly different approaches have been identified, e.g., P-GRADE [41], 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 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 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 for such mechanisms. Compared to EMPEROR, the GJMF provides a more flexible architecture, greater standardization support, and multiple levels of job management abstractions.

7. Future Work

A number of possible future extensions to the framework have been identified and are under investigation.

- Data management. The GJMF is envisioned to be complemented with a service-based, middleware- and transport-agnostic 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, e.g., investigation of transport-agnostic mechanisms for integration with advanced job management mechanisms.

- Workflow management. While the GJMF currently integrates well with workflow management systems by offering transparent Grid access, the framework itself lacks support for execution of interdependent tasks. A middleware-agnostic tool for execution of static workflows would provide a management solution similar in functionality to the higher-order services of the GJMF for tasks and task groups.

- Adaptation of the framework to other environments. Currently, the GJMF builds and relies on GT4 for deployment and is therefor dependent on the Apache Axis SOAP engine. The GJMF service structure is designed to keep service core functionality independent of underlying

service engine, which should facilitate adaptations to alternative service environments, e.g., Apache Axis2 and Apache CXF. Preliminary investigation reveals that modification of the GJMF service security model and exposure of service state as WSRF resource properties are likely to be required for future adaptations. Adaptation of the framework to new Globus Toolkit versions is also of interest.

8. Conclusion

We have proposed a flexible and loosely coupled architecture for middleware-agnostic Grid job management. The architecture is designed as a composable framework of Web Services that abstracts resource and system heterogeneity on multiple levels. The framework is intended for use in Grid environments and makes no assumptions of centralized control of resources or omniscience in scheduling. Focus is placed on interoperability and maintaining non-intrusive coexistence and integration models. The architecture builds on standardization efforts such as JSDL, WSRF, OGSA BES, and OGSA RSS.

The architecture is organized in hierarchical layers of functionality, where services abstract and aggregate functionality from services in 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 from Grid middlewares and infrastructure components, and abstract Grid functionality through generic Grid job management interfaces. Applications built on the framework are loosely coupled to underlying Grids, gain portability and flexibility in deployment, and utilize heterogeneous Grid resources transparently.

In this work we have also presented a proof-of-concept implementation of the architecture that builds on Grid and Web Service standardization efforts and supports a range of Grid middlewares. Middleware transparency is provided through a set of 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 be dynamically altered and adapt to failures occurring in job submission or execution.

All services in the framework provide 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.

9. Acknowledgements

We extend gratitude to Peter Gardfjäll, Arvid Norberg, and Johan Tordsson who's prior work and feedback provides a foundation for this work. We acknowledge the Swedish Research Council (VR) who supports the project under contract 621-2005-3667, and the High Performance Computer Center North (HPC2N) on who's resources the research is 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, March 2010.

[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, March 2010.

[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, March 2010.

[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] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2010.

[14] 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.

[15] W. Allcock (editor). GridFTP: Protocol extensions to FTP for the Grid. http://www.ogf.org/documents/GFD.20.pdf, March 2010.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] E. Elmroth and J. Tordsson. A standards-based Grid resource brokering service supporting advance reservations, coallocation and cross-Grid interoperability. *Concurrency and Computation: Practice and Experience*, Vol. 25, No. 18, pp. 2298 - 2335, 2009.

[24] 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.

[25] 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.

[26] 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, March 2010.

[27] 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, March 2010.

[28] 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, March 2010.

[29] 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.

[30] 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.

[31] N. Furmento, W. Lee, A. Mayer, S. Newhouse, and J. Darlington. ICENI: an open grid service architecture implemented with Jini. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–10. IEEE Computer Society Press Los Alamitos, CA, USA, 2002.

[32] 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.

[33] Globus. http://www.globus.org. March 2010.

[34] 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, March 2010.

[35] 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, March 2010.

[36] 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, March 2010.

[37] Grid Interoperability Now. http://wiki.nesc.ac.uk/gin-jobs/. March 2010.

[38] 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/, March 2010.

[39] 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.

[40] Cluster Resources inc. Torque resource manager. http://www.clusterresources.com/pages/products/torque-resource-manager.php, March 2010.

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

[42] 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.

[43] A. Kertesz and P. Kacsuk. GMBS: A new middleware service for making grids interoperable. *Future Generation Computer Systems*, 26(4), pages 542–553, 2010.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] OASIS Open. Reference Model for Service Oriented Architecture 1.0. http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf, March 2010.

[50] OMII Europe. OMII Europe - open middleware infrastructure institute. http://omii-europe.org, March 2010.

[51] P-O. Östberg and E. Elmroth. Impact of Service Overhead on Service-Oriented Grid Architectures. *Submitted*, 2010. Preprint available at http://www.cs.umu.se/ds.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

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

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

[58] The Grid Infrastructure Research & Development (GIRD) project. Umeå University, Sweden. http://www.cs.umu.se/ds, March 2010.

[59] 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.

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

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

[62] 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.

[63] 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.

V

Paper V

Impact of Service Overhead on Service-Oriented Grid Architectures

Per-Olov Östberg and Erik Elmroth

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

Abstract: Grid computing applications and infrastructures build heavily on Service-Oriented Computing development methodology and are often realized as Service-Oriented Architectures. Current Service-Oriented Architecture methodology renders service components as Web Services, and suffers performance limitations from Web Service overhead. The Grid Job Management Framework (GJMF) is a flexible Grid infrastructure and application support component realized as a loosely coupled network of Web Services that offers a range of abstractive and platform independent interfaces for middleware-agnostic Grid job submission, monitoring, and control. In this paper we a present a performance evaluation aimed to characterize the impact of service overhead on Grid Service-Oriented Architectures and evaluate the efficiency of the GJMF architecture and optimization mechanisms designed to mediate impact of Web Service overhead on architecture performance.

Key words: Grid computing, Grid job management, Grid ecosystem, Performance Analysis, Service-Oriented Architecture.

Impact of Service Overhead on Service-Oriented Grid Architectures

Per-Olov Östberg
Umeå University
SE-901 87
Umeå, Sweden
p-o@cs.umu.se

Erik Elmroth
Umeå University
SE-901 87
Umeå, Sweden
elmroth@cs.umu.se

***Abstract*—Grid computing applications and infrastructures build heavily on Service-Oriented Computing development methodology and are often realized as Service-Oriented Architectures. Current Service-Oriented Architecture methodology renders service components as Web Services, and suffers performance limitations from Web Service overhead. The Grid Job Management Framework (GJMF) is a flexible Grid infrastructure and application support component realized as a loosely coupled network of Web Services that offers a range of abstractive and platform independent interfaces for middleware-agnostic Grid job submission, monitoring, and control. In this paper we a present a performance evaluation aimed to characterize the impact of service overhead on Grid Service-Oriented Architectures and evaluate the efficiency of the GJMF architecture and optimization mechanisms designed to mediate impact of Web Service overhead on architecture performance.**

***Keywords*-Grid computing; Grid job management; Grid ecosystem; Performance Analysis; Service-Oriented Architecture**

I. Introduction

This paper presents a performance evaluation aimed to characterize and quantify impact of Web Service overhead on Grid Service-Oriented Architectures. The Grid Job Management Framework (GJMF) [13], an infrastructure component for computational Grid environments realized as a loosely coupled network of layered Web Services, is used as a testbed for experimentation and the efficiency of GJMF mechanisms designed to mediate impact of Web Service overhead are evaluated against the functionality offered by the framework.

Grid computing is an approach to distributed computing designed to provide scalable computational infrastructures through federation and aggregation of existing resources into virtual systems. Resources are typically aggregated through middlewares that abstract underlying resource systems and organize Grid user bases in Virtual Organizations. To facilitate application integration and provide desirable middleware features such as platform and language independence, many Grid middlewares utilize Service-Oriented Computing methodology and expose interfaces in the form of Web Services.

A number of Grid middlewares have evolved and are in production use. Despite being designed to alleviate the interoperability issues that exist between native resource systems, Grid middlewares exhibit interoperability issues due to heterogeneity in, e.g., security models, job description formats, and job control interfaces. The GJMF is designed to abstract Grid middleware heterogeneity and provide middleware-agnostic, flexible, and intuitive job management interfaces [13].

To provide platform independence and interoperability, the GJMF exposes functionality through SOAP WSRF Web Services and is subject to well-known Web Service invocation overhead issues. In this paper we investigate the impact of service invocation overhead on framework performance, and evaluate effectiveness of mechanisms, e.g., asynchronous message processing, batch invocations, and local call invocation optimizations, designed to mediate service overhead impact.

The rest of this paper is organized as follows. Section II provides an overview of the GJMF architecture and components. Section III presents a performance evaluation categorizing and quantifying framework overhead and the efficiency of a set of overhead mediation mechanisms. Section IV presents a brief sampling of related work, and Section V concludes the paper.

II. The Grid Job Management Framework

The Grid Job Management Framework (GJMF) [13], is a hierarchical network of services designed to provide intuitive and middleware-agnostic interfaces to Grid job management. The framework offers concurrent and transparent access to multiple middlewares, is designed to be flexible in deployment, and can be tailored through dynamic reconfiguration and customization points to alter component and framework behavior. The framework design is based on a software development methodology inspired by the notion of an ecosystem of Grid infrastructure components [4], [17], and supports a model for distributed software reuse where components can be discovered, replaced, or updated dynamically.

Implemented as a Globus Toolkit [5] Java SOA, the GJMF is platform and language independent, and provides a Grid access model that abstracts Web Service development complexity. Framework design is based on a number of Grid and Web Service standards, e.g., the Job Submission Description Language (JSDL) [2], the Web Service Description Language (WSDL) [3], the Web Service Resource Framework (WSRF) [6], and the OGSA Basic Execution Service (OGSA BES) [7] and Resource Selection Services (OGSA RSS) [8].

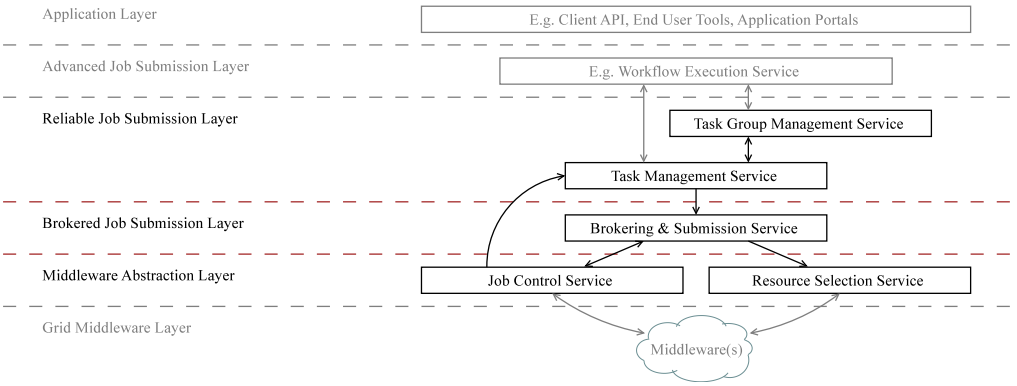


Fig. 1: The GJMF framework architecture. Services organized in hierarchical layers of functionality. Within the framework services communicate hierarchically, service clients are not restricted to this invocation pattern. Illustration from [13].

A. Architecture

As illustrated in Figure 1, the GJMF architecture stratifies a set of typical Grid job management mechanisms into six layers. Framework layers are populated by Web Services that implement well-defined interfaces to common types of Grid job management. Detailed information about the design and implementation of the GJMF is available in [13], here we give a brief overview of the architecture and components of interest to the performance evaluation. The bottom layers of the architecture abstract Grid-specific interfaces and technology, and isolate the rest of the framework from middleware-specific dependencies. Services in higher layers aggregate functionality from services in lower layers and provide increasing levels of (customizable) automation. The GJMF architecture is designed to provide deployment flexibility and abstract Grid infrastructure development and administration complexity.

B. Components

The core of the GJMF architecture consists of five services. The middleware abstraction layer is populated by the Job Control Service (JCS), which provides a middleware agnostic interface for job submission, control, and monitoring based on the OGSA BES, and the Resource Selection Service (RSS), a middleware information system monitoring and job execution plan formulation service based on the OGSA RSS.

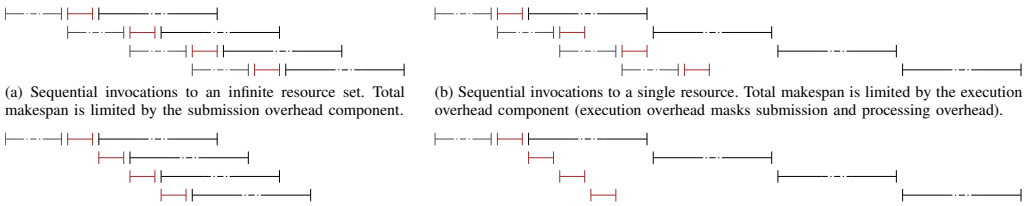
For brokered job submission, the GJMF defines the Brokering and Submission Service (BSS), which employs the RSS and the JCS for job to resource matching and job submission respectively. Atop the BSS, the Task Management Service (TMS) defines a high-level ”fire and forget” type of job management interface for Grid application integration. Internally, the TMS makes use of the BSS for job brokering and submission, and the JCS for job monitoring and control. The TMS defines job submission and execution failure handler policies and provides a high-level interface for job management. Similar in structure to the TMS, the Task Group Management Service (TGMS) provides a high-level interface for managing groups of tasks in a single context.

In addition to the core functionality set, the GJMF provides services for job description translation and log management. The JSDL Translation Service (JTS) defines an interface for translation of job description documents between JSDL and Grid middleware job description formats, and is in GJMF employed to decouple the JCS from middleware dependencies. Service clients may employ the JTS to facilitate migration of Grid applications by translating middleware-specific job descriptions to standardized JSDL documents. The Log Accessor Service (LAS) provides a log database service interface that stateful (i.e. task processing) GJMF services can use to store task and job processing information. The LAS can be utilized by service clients to track task progress and state.

All high-level GJMF services employ asynchronous communication patterns and support batch invocation modes. Services expose plug-in customization points and employ dynamic reconfiguration mechanisms to allow third parties to tailor component and architecture functionality to deployment environments. Services can be deployed in distributed patterns and employ local call optimizations for increased invocation performance when co-deployed.

III. Performance Evaluation

To measure framework efficiency, and characterize Web Service overhead impact on performance, we define GJMF overhead as the time penalty imposed by use of the framework and use it as a cost function for efficiency. To gain a generalized model for Grid overhead and correlate GJMF overhead to it, we define Grid overhead as time spent performing any job enactment task other than execution of a job binary. This model defines Grid tasks such as brokering, job submission, and file staging as Grid overhead. To isolate GJMF contributions to Grid overhead, we model Grid overhead as a



(a) Sequential invocations to an infinite resource set. Total makespan is limited by the submission overhead component.

(b) Sequential invocations to a single resource. Total makespan is limited by the execution overhead component (execution overhead masks submission and processing overhead).

(c) Batch invocation to an infinite resource set. Total makespan is limited by the processing overhead component.

(d) Batch invocation to a single resource. Total makespan is limited by the execution overhead component (execution overhead masks submission and processing overhead).

Fig. 2: A model for Grid job enactment overhead. Submission overhead, processing (GJMF) overhead, and execution overhead (illustrated in gray, red, and black, respectively) are sequential components of the makespan of a job. When jobs outnumber computational resources, the execution overhead component dominates the total makespan of a set of jobs.

process divided into three sequential components; submission overhead, processing overhead, and execution overhead.

Submission overhead is defined as overhead incurred prior to a job being present in a GJMF service and consists of factors such as Java class loading and Web Service invocation time. Processing overhead is the GJMF overhead contribution 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. In this model, submission and execution overhead are external to the GJMF, and imposed by service clients, middlewares, and native resource systems.

As illustrated in Figure 2, total overhead for enactment of a group of jobs depends on the saturation level of the resource set used for job enactment. As the GJMF is constructed in hierarchical layers, the GJMF services can process jobs in pipelines, facilitating parallel processing of tasks that allows the GJMF to mask overhead contributions through temporal overlaps between task processing and job executions. Total system overhead imposed by the GJMF is constituted by the sum of GJMF overhead contributions subtracted by overhead the GJMF is able to mask by parallel processing of tasks.

When the number of available computational hosts exceeds the number of jobs, the GJMF ability to mask overhead is limited and system overhead is bound by the submission and processing overhead components, as illustrated in Figure 2a and Figure 2c. When the number of jobs exceeds the number of available computational hosts, total system overhead is bound by the job execution overhead component. The GJMF ability to mask overhead contributions from individual jobs in these situations is illustrated in figures 2b and 2d.

To quantify overhead incurred by the GJMF, we configure a GJMF deployment to operate on top of a Grid middleware and compare job submission and processing performance to using the middleware directly. Total overhead imposed by the GJMF is calculated as the makespan of processing a group of jobs subtracted by a known theoretical minimum time required to execute all jobs in the group on an ideal system, i.e. a system that does not impose job execution overhead.

To isolate individual contributions to 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.

A. Test Environment

As the tests of the performance evaluation focus on investigating overhead imposed by the GJMF, a limited test environment is sufficient as these performance limitations are independent of the number of computational resources used.

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 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, 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 memory allocation pools range in size from 512 MB to 1 GB. As GJMF overhead is independent of middleware overhead, and the purpose of the performance evaluation is to investigate GJMF overhead contributions to total system overhead, a test setup using an older middleware version is acceptable.

We employ GT4 WS-GRAM as a 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 impact of GJMF overhead when testing ideal jobs, we utilize the GT4 Fork job dispatch mechanism. For tests of realistic deployment scenarios we use the GT4 PBS job dispatch module, and submit jobs to a local cluster using Torque. To minimize the impact of stochastic network behaviors in our overhead measurements we do not use jobs that involve file transfers. Not using file transfers constitutes a worst case scenario for GJMF overhead as it reduces GJMF

ability to mask overhead (GJMF relies on middlewares for file staging). The known length of the test jobs are used as a theoretical minimum time for executing a job. A single middleware is used in tests as the evaluation aims to quantify GJMF overhead rather than illustrate middleware independence. More information about the Globus Toolkit can be found in [5].

In the tests, we use the GT4 WS-SecureConversation [1] security mechanism with client and service security descriptors in all Web Service invocations, including communication with the underlying Grid middleware. This mechanism performs authentication and encryption of communication channels, increases communication overhead and reduces invocation throughput for Web Service invocations. The selected security setup is used in some high security production Grid deployments, and is used for these tests as it represents a worst case scenario for invocation throughput.

B. Performance Tests

The purpose of the evaluation experiments is to investigate individual overhead contributions to total system overhead, relate overhead imposed by the GJMF to the functionality offered by the framework, and characterize service overhead contributions to GJMF overhead. In the evaluation, we perform a set of tests of service invocation capabilities, job submission performance, and job throughput to quantify and evaluate impact of the GJMF overhead on total system overhead. The tests performed are based on the overhead model illustrated in Figure 2 and designed to illustrate individual aspects of the framework overhead. The five types of tests performed are:

a) Job submission tests (Section III-B1): Investigate GJMF service client overhead associated with job submission and illustrate the impact of, and the trade-offs between, different service deployment and invocation methods.

b) Job throughput tests for ideal computational settings (Section III-B2): Investigate service overhead for scenarios illustrated in figures 2a and 2c, where computational resources outnumber jobs. This constitutes a worst-case scenario for GJMF overhead and quantifies an upper bound for overhead imposed by use of the framework.

c) Job throughput tests for realistic computational settings (Section III-B3): Investigate service overhead for scenarios illustrated in figures 2b and 2d, where jobs outnumber computational resources. These tests illustrate GJMF ability to mask overhead through task parallelization.

d) Service invocation capability tests (Section III-B4): Investigate invocation throughput for the GJMF auxiliary services to quantify their contributions to total system overhead and illustrate trade-offs between service communication overhead and service complexity.

e) Service invocation optimization tests (Section III-B5): Investigate performance trade-offs for different types of invocation optimization mechanisms and illustrate the impact of local call optimizations on service communication overhead.

1) Job Submission: To evaluate GJMF job submission overhead we measure the framework's submission throughput and quantify it against a baseline measurement of 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.

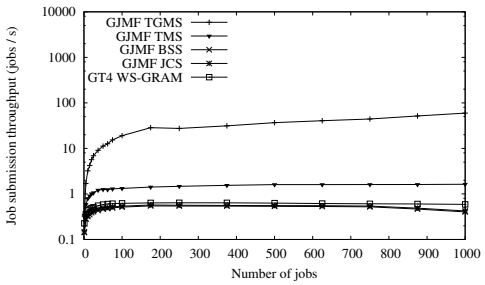
For all tests, job, task, and task group submission performance is measured as turn-around time for submission in service clients using realistic job descriptions. Average job submission makespan is used as a direct measurement of the overhead incurred by the GJMF for job submission.

As can be seen in Figure 3, JCS and BSS job submission throughput is slightly lower than that of GT4 WS-GRAM. This 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 asynchronous message processing. The TGMS exhibits the highest throughput as it submits multiple tasks in single service invocations. As can be seen in Figure 3b, 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.

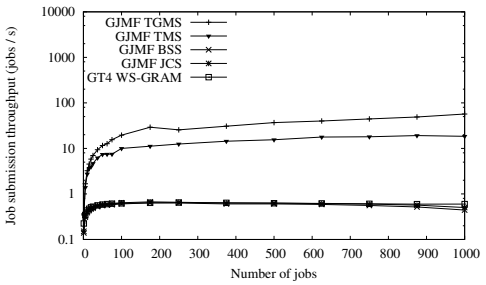
When using local call optimizations, as illustrated in Figure 3c and 3d, submission overhead can be reduced for all GJMF services. The TGMS and TMS achieve very high submission throughput due to their ability to perform asynchronous job submissions. Use of local call optimizations reduce invocation overhead to a range where impact of this overhead component becomes almost negligible.

2) Job Throughput for Ideal Computational Settings: To evaluate and get an upper bound for the processing overhead component of total system overhead, we measure framework job processing throughput when the GJMF's ability to mask overhead is minimized, and quantify it against a baseline measurement of GT4 WS-GRAM job processing performance. As indicated in Figure 2, this occurs when the number of available computational resources exceeds the number of jobs. To simulate this, and isolate and maximize impact of the GJMF overhead, we submit jobs with zero execution time, i.e. */bin/true* executions, to the GT4 middleware using the Fork dispatcher, which starts all jobs in parallel on the same machine without delay. As this setting minimizes GJMF ability to mask processing overhead through task parallelization, it constitutes a worst-case scenario for GJMF overhead and is used to quantify an upper bound for 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.

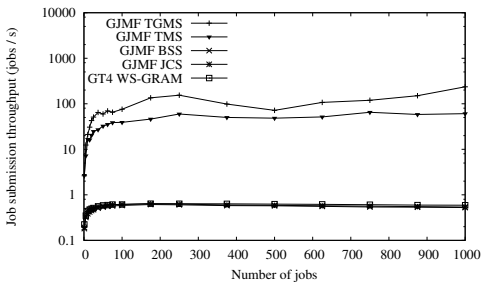
As can be seen in Figure 4, the GJMF incurs an average performance penalty of less than one second per job for ideal



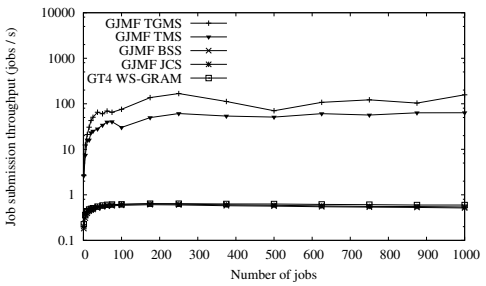
(a) GRAM and GJMF job submission throughput. Sequential job submissions using Web Service invocations.



(b) GRAM and GJMF job submission throughput. Batch job submissions using Web Service invocations.



(c) GRAM and GJMF job submission throughput. Sequential job submissions using local call optimization invocations.



(d) GRAM and GJMF job submission throughput. Batch job submissions using local call optimization invocations.

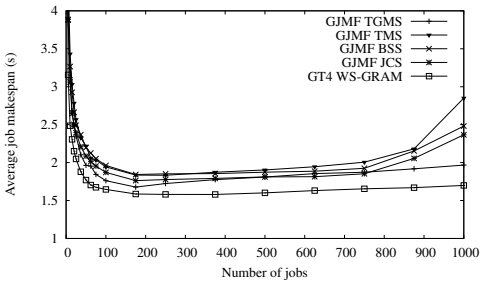
Fig. 3: GJMF job submission performance. Job submission throughput as a function of number of jobs. Vertical axis logarithmic.

(zero execution time) jobs. This overhead includes factors such as job submission, interservice communication, job brokering, and distributed state management. Batch invocation Web Service invocation job submissions (Figure 4b) mediate the incurred overhead somewhat. Particularly, JCS overhead is reduced to a level close to that of using GT4 WS-GRAM directly. As the BSS performs task to resource matching, the BSS and services using the BSS, i.e. TGMS and TMS, suffer overhead from the brokering process that, as illustrated in figures 2a and 2b, is partially masked by the submission overhead when using sequential invocation modes (Figure 4a).

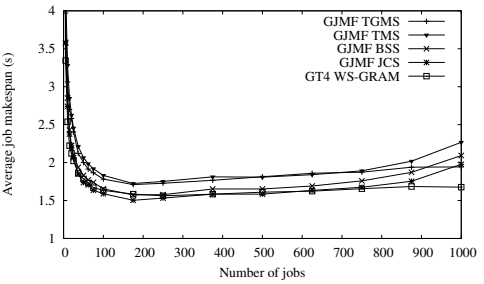
When using service clients co-deployed with the GJMF, as illustrated in figures 4c and 4d, GJMF local call optimizations allow JCS overhead to be reduced to close to GT4 WS-GRAM performance regardless of invocation mode. Local call optimizations do not greatly affect the throughput of the other GJMF services as these are still bound by 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 JCS in figures 4a and 4b. 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. Use of batch invocation modes alleviates this effect for all services, but does not eliminate it as back-end invocations still marshal requests and create job and task resources.

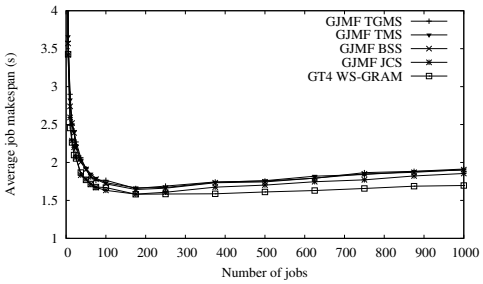
3) Job Throughput for Realistic Computational Settings: To evaluate the processing overhead component of total system overhead under more realistic circumstances, we measure framework job processing throughput and quantify overhead incurred by the GJMF when deployed with a production environment system (GT4 and PBS Torque) against a baseline measurement of GT4 WS-GRAM job processing performance. In these tests, the number of jobs exceeds the number of available computational resources, allowing the GJMF to mask overhead through parallel task processing. To establish a



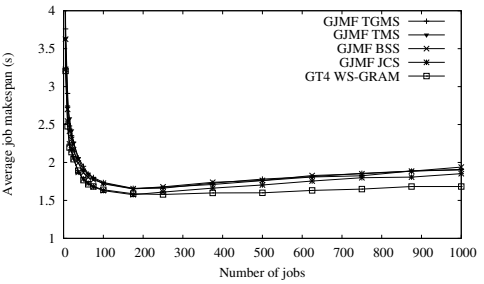
(a) GRAM and GJMF job makespan. Sequential job submissions using Web Service invocations.



(b) GRAM and GJMF job makespan. Batch job submissions using Web Service invocations.



(c) GRAM and GJMF job makespan. Sequential job submissions using local call optimization invocations.



(d) GRAM and GJMF job makespan. Batch job submissions using local call optimization invocations.

Fig. 4: GJMF job processing performance, ideal computational settings. Job makespan as a function of number of jobs.

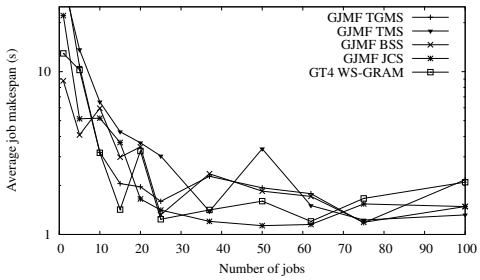
theoretical minimum time required to execute a set of jobs, we employ */bin/sleep* jobs of a known, non-zero execution length in the tests. Job, task, and task group throughput are measured using sequential and batch invocation modes for Web Service invocations and local call optimization invocations.

In Figure 5, 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) is subtracted from each measurement to better illustrate overhead components. As illustrated, a stochastic element is 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 60 and 120 seconds). 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 is in the tests partially masked by job execution times and is, independently of invocation mode and mechanism, small enough to be masked by the PBS component.

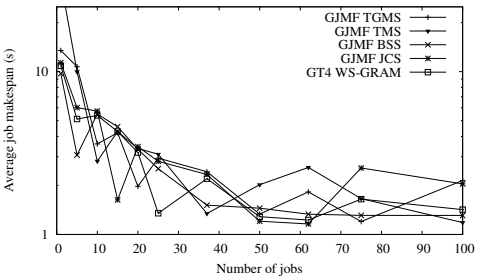
The term realistic computational settings is here used to denote job management components operating in a setting where non-zero job execution overhead and duration allow the GJMF to mask individual service overhead contributions. In realistic scenarios, job execution durations are typically orders of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

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 throughput of the LAS, JTS, and RSS using sequential and batch invocation modes for Web Service and local call optimization invocations. For all tests, typical GJMF tasks containing full JSDL documents are used as service invocation parameters. In the LAS tests tasks are stored in logs, for the JTS tests task JSDLs are translated to GT4 RSL, and in the RSS tests tasks are matched to resources.

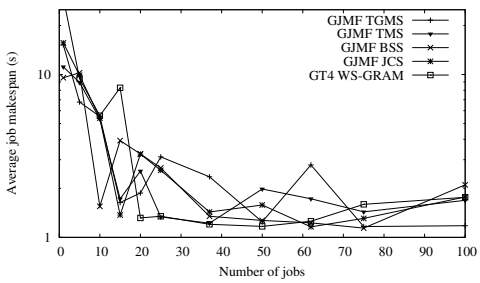
As can be seen in Figure 6, local call optimizations allow greater invocation throughput than Web Service invocations and batch invocations increase invocation throughput for the auxiliary services. The LAS implements an asynchronous communication model that allows service client invocation throughput to be bound by communication overhead (Fig-



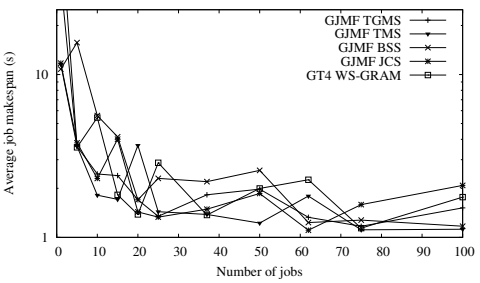
(a) GRAM and GJMF job makespan. Sequential job submissions using Web Service invocations.



(b) GRAM and GJMF job makespan. Batch job submissions using Web Service invocations.



(c) GRAM and GJMF job makespan. Sequential job submissions using local call optimization invocations.



(d) GRAM and GJMF job makespan. Batch job submissions using local call optimization invocations.

Fig. 5: GJMF job processing performance, realistic computational settings. Job makespan as a function of number of jobs. A stochastic element is introduced to the execution overhead component by the batch system. Vertical axis logarithmic.

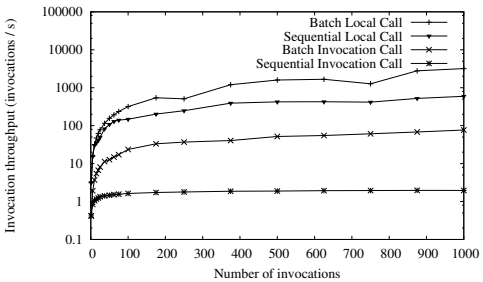
ure 6a). The JTS and RSS provide synchronous request processing models, and are performance bound by the processing capacity of the service implementation as well as the communication overhead (figures 6b and 6c, respectively).

The JTS is able to process requests efficiently 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 be the limiting factor for invocation throughput. In this case, use of local call optimizations does not affect invocation throughput. As these measurements are made using the same setup as the job submission and throughput tests of sections III-B1, III-B2, and III-B3, the values for local call optimization tests can be used as rough estimates of the individual overhead contributions of these services to GJMF processing overhead.

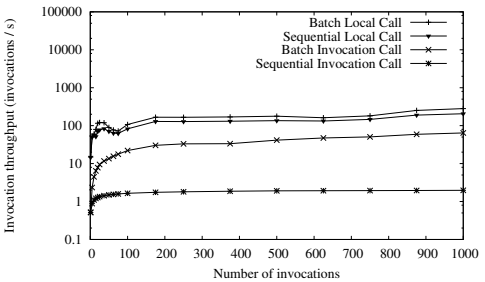
5) Local Call Optimizations: As framework services are likely to be co-deployed, local call optimizations are expected to heavily affect framework performance. To evaluate performance and impact of the GJMF local call optimization mechanisms, we measure invocation throughput for a reference service using 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. As the GJMF services are implemented in Java, measurements of Java method invocations will constitute a measurement of the maximal possible invocation throughput in the test environment, and serve as a baseline for comparison. To minimize impact of memory starvation effects in the tests, invocations are made sequentially and in parallel (with a multithreaded service client) using small messages.

The different types of service invocation mechanisms used in the tests are illustrated in Figure 7. GJMF local call optimizations identify service implementation back-ends based on class name and perform marshaling of service invocation data using immutable wrapper types. Globus Local Invocations perform service implementation lookup through a Java Naming and Directory Interface (JNDI) [16] 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.

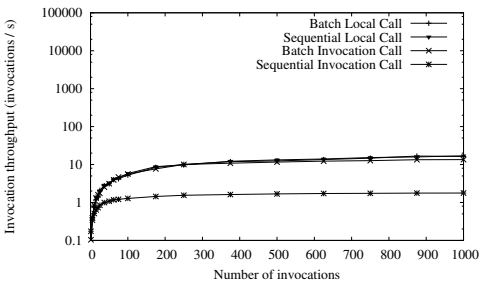
As illustrated in Figure 8, local call optimizations greatly improve service invocation throughput. GJMF local call mechanisms provide invocation performance comparable to existing Axis and Globus optimizations. All invocation optimizations



(a) The Log Accessor Service.



(b) The JSDL Translation Service.



(c) The Resource Selection Service.

Fig. 6: Auxiliary service performance. Invocation throughput as a function of number of invocations. Vertical axis logarithmic.

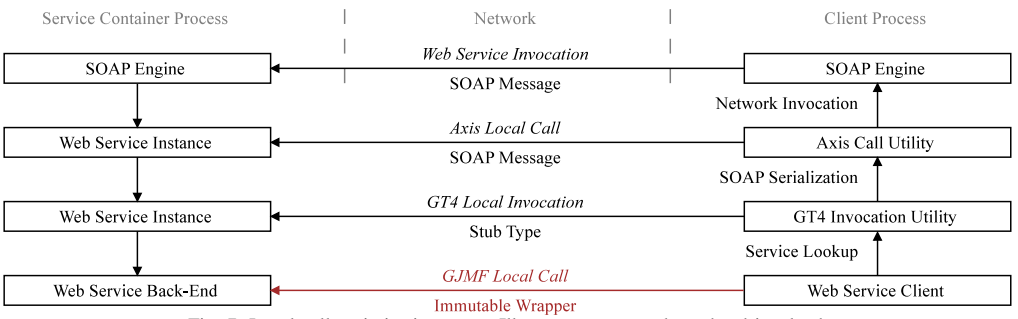
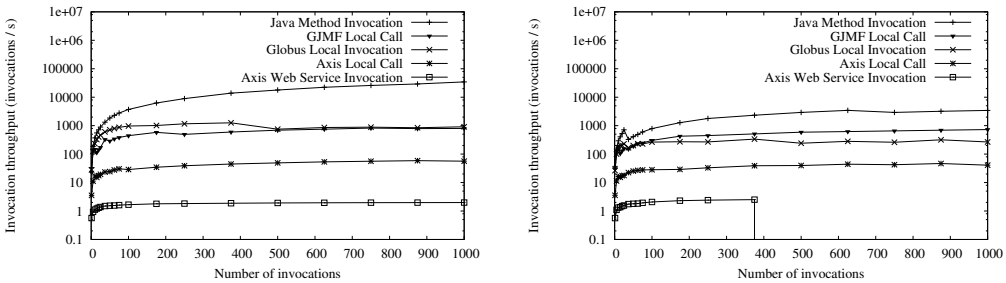


Fig. 7: Local call optimization types. Illustrates actors and overhead involved.

scale well for large numbers of parallel invocations, while Axis Web Service invocation throughput drops due to service container memory and thread pool exhaustion issues. GJMF local call optimizations provide local call capabilities for state notification delivery with comparable performance. This is not evaluated in the tests as Globus Local Invocations and Axis Local Calls do not support this functionality.

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 does not directly affect service invocation performance, it reduces service container memory load when using WS-BaseNotification based notification schemes. As can be seen in Figure 8a, the Globus local 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, and second, the GJMF performs context-based type validation of job description data in immutable wrapper types. For the parallel invocation case illustrated in Figure 8b, GJMF local call optimizations outperforms the Globus local invocations mechanism, which is attributed to the lower memory usage of the GJMF local call optimizations.



(a) Sequential invocations.

(b) Parallel invocations.

Fig. 8: Web Service invocation throughput using different types of invocation optimization mechanisms. Service invocation throughput as a function of number of invocations. Vertical axis logarithmic.

C. Discussion

In the GJMF, task processing is parallelized and, as illustrated in Figure 2, overhead 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 resources available. When the number of jobs exceeds the number of nodes available for immediate job submission (as illustrated in 2b and 2d), GJMF job processing is performed in parallel with job executions, and job durations mask impact of overhead incurred by the GJMF. For realistic scenarios, e.g. use of co-deployed GJMF services in computational Grids, job durations are typically several orders of magnitude larger than overhead incurred by the GJMF and help mask GJMF overhead even when large numbers of computational resources are available. Combined with the typically high utilization rates of Grids, this effect is expected to effectively mediate impact of GJMF overhead.

As the greater bulk of the GJMF job processing overhead is constituted of service invocation, co-hosting GJMF services allows 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. For distributed clients, job submission overhead for groups of jobs can be mediated to a one-time cost by use of batch invocation modes (illustrated in 2c and 2d). In most cases, submission overhead impacts 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 effect. GJMF processing overhead can be mediated by GJMF deployment and configuration options, e.g., by use of co-deployment of services and local call optimizations. Use of batch invocation modes for service invocations conserve network bandwidth and reduce memory footprints of GJMF services and clients. Use of local call optimizations eliminates network bandwidth requirements and reduces memory and CPU used for service invocations to a minimum.

The overhead model used here (illustrated in Figure 2) is somewhat simplified as the GJMF incurs 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 [12], the GJMF is designed 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 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 not evaluated in tests.

IV. Related Work

Alternative approaches to Grid job management include, e.g., The GridWay Metascheduler [10] and the Community Scheduler Framework (CSF4) [19]. Both of these contributions are Grid meta-schedulers built using the Globus Toolkit and designed to abstract complexity in job management. GridWay provides a metascheduler framework for adaptive scheduling and execution of Grid jobs focused on reliable and autonomous execution of jobs. CSF4 is constructed as a framework of WSRF Web Services and based on OGSA.

Falkon [15] is a lightweight task execution framework designed for Many Task Computing [14]. Falkon and GJMF are both service-based frameworks, but designed for different use cases using different technologies. Falkon achieves high job submission throughput rates through optimizations such as custom state update protocols, while GJMF provides, e.g., multiple middleware-agnostic job management interfaces.

GridSAM [11] is a standards-based job submission system that abstracts underlying resource managers through a Web Service interface. GridSAM employs a staged event-driven architecture (SEDA) [18] job submission pipeline designed to reduce job submission response times very similar to the asynchronous job processing of the GJMF.

The Simple API for Grid Applications (SAGA) [9] is an API standardization initiative that like the GJMF aims to provide a unified interface to Grid integration. SAGA features separation of functionality in layers, but the design philosophy of the SAGA API differs from the GJMF in, e.g., that asynchronicity is handled via polling rather than publish/subscribe notification schemes. Performance evaluations comparing the GJMF to implementations of the SAGA API are subject for future work.

Compared to these approaches GJMF provides, e.g., concurrent access to multiple Grid middlewares, dynamic reconfiguration and recomposition of the framework, and job description translation functionality. More exhaustive treatment of Grid job management mechanisms is available in [13].

V. Conclusion

In this paper we evaluate the architecture of the Grid Job Management Framework, and investigate the impact of service invocation overhead on framework performance. Investigation indicates that framework overhead is mainly constituted by Web Service invocation latencies, and a number of tests are undertaken to evaluate the effectiveness of mechanisms designed to mediate overhead, e.g., asynchronous message processing, batch invocations, and local call optimizations.

Service components are likely to be co-hosted, and the GJMF is equipped with local call invocation optimization mechanisms that reduce invocation overhead and mediate container load. For distributed deployments, the GJMF is equipped with batch invocation mechanisms that reduce invocation and message processing overhead. Asynchronous message processing mechanisms and message buffers reduce invocation latencies and facilitate task parallelism in the framework.

As the GJMF utilizes Web Services as a distributed component platform the framework suffers non-neglibile overhead from service communication. Construction of the framework as layers of loosely coupled services affords the GJMF deployment flexibility and the ability to partially mask overhead imposed by service communication through task parallelization. Use of techniques such as local call optimizations (co-deployed services), asynchronous message processing, and batch invocation modes (distributed services) allow services to effectively minimize service invocation overhead. Combined with sound architecture design and coarse-grained communication patterns, these techniques effectively mitigate the impact of Web Service overhead on architecture performance. The effective overhead imposed by use of the GJMF for Grid job management is less than one second per job, demonstrating that Web Services are a viable realization platform for Service-Oriented Grid Architectures.

VI. Acknowledgements

The authors extend their gratitude to Keith Jackson for valuable feedback and interesting discussions. This work is done in collaboration with the High Performance Computing Center North (HPC2N) and is funded by the Swedish Research Council (VR) under Contract 621-2005-3667 and the Swedish Government's strategic research project eSSENCE. The authors acknowledge the Lawrence Berkeley National Laboratory (LBNL) for supporting the project under U.S. Department of Energy Contract DE-AC02-05CH11231.

References

[1] 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, February 2011.

[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, February 2011.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, February 2011.

[4] 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.

[5] 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.

[6] 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, February 2011.

[7] 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, February 2011.

[8] 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, February 2011.

[9] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. Von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications. High-level application programming on the Grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006.

[10] 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.

[11] 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.

[12] 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.

[13] P-O. Östberg and E. Elmroth. GJMF - A Composable Service-Oriented Grid Job Management Framework. Preprint available at http://www.cs.umu.se/ds, submitted, 2010.

[14] 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.

[15] 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.

[16] Sun Microsystems. Java Naming and Directory Interface (JNDI). http://java.sun.com/products/jndi/, February 2011.

[17] The Globus Project. An "ecosystem" of Grid components. http://www.globus.org/grid_software/ecology.php, February 2011.

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

[19] 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.

VI

Paper VI

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.cs.umu.se/ds

[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.

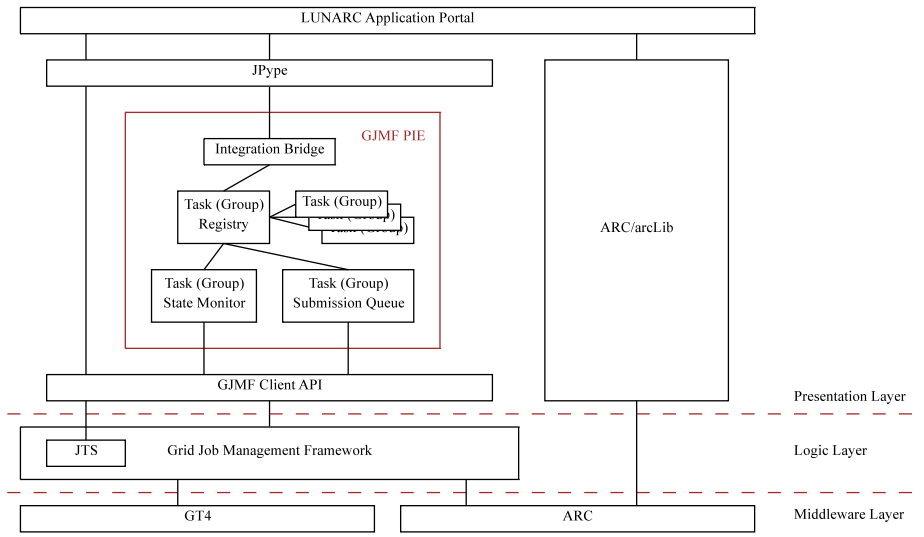


Fig. 1. Overview of the LAP-GJMF integration architecture. Integration components are presented without detailing the internal workings of the LAP or the GJMF.

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.

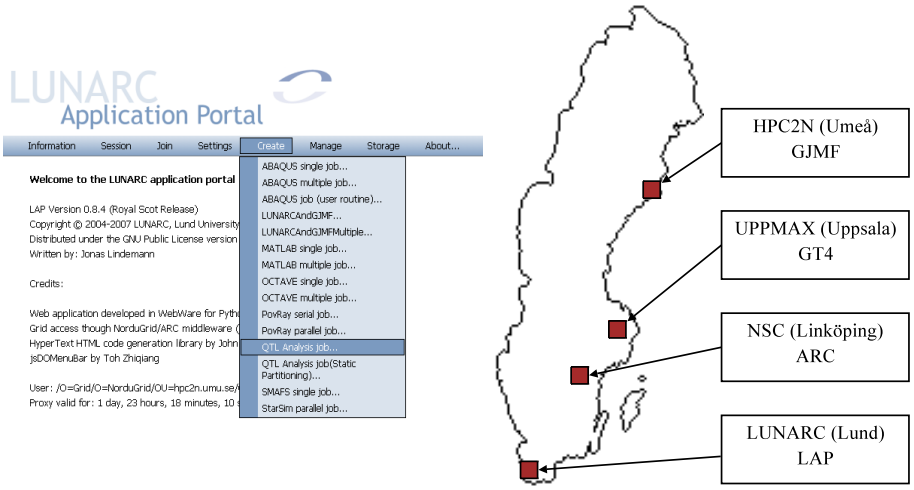


Fig. 2. The LAP and the production environment deployment configuration.

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.

VII

Paper VII

Decentralized, Scalable, Grid Fairshare Scheduling (FSGrid)

Per-Olov Östberg, Daniel Henriksson, and Erik Elmroth

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

Abstract: This work addresses Grid fairshare allocation policy enforcement and presents FSGrid, a decentralized system for Grid-wide fairshare job prioritization. The presented system builds on three contributions; a flexible tree-based policy model that allows delegation of policy definition, a job prioritization algorithm based on local enforcement of distributed fairshare policies, and a decentralized architecture for non-intrusive integration with existing scheduling systems. The system supports organization of users in virtual organizations and divides usage policies into local and global policy components that are defined by resource owners and virtual organizations. The architecture realization is presented in detail along with an evaluation of system behavior in an emulated environment. The system is shown to meet scheduling objectives and convergence noise (mechanisms counteracting policy allocation convergence) are characterized and quantified. System mechanisms are shown to be scalable in tests using realistic policy allocations.

Key words: Grid scheduling, Fairshare scheduling, Fair share scheduling, Grid allocation policy enforcement.

Decentralized, Scalable, Grid Fairshare Scheduling (FSGrid)

Per-Olov Östberg and Daniel Henriksson and Erik Elmroth

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

Abstract

This work addresses Grid fairshare allocation policy enforcement and presents FSGrid, a decentralized system for Grid-wide fairshare job prioritization. The presented system builds on three contributions; a flexible tree-based policy model that allows delegation of policy definition, a job prioritization algorithm based on local enforcement of distributed fairshare policies, and a decentralized architecture for non-intrusive integration with existing scheduling systems. The system supports organization of users in virtual organizations and divides usage policies into local and global policy components that are defined by resource owners and virtual organizations. The architecture realization is presented in detail along with an evaluation of system behavior in an emulated environment. The system is shown to meet scheduling objectives and convergence noise (mechanisms counteracting policy allocation convergence) are characterized and quantified. System mechanisms are shown to be scalable in tests using realistic policy allocations.

Keywords: Grid scheduling, Fairshare scheduling, Fair share scheduling, Grid allocation policy enforcement

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

1. Introduction

The core idea of fairshare scheduling is to schedule jobs with respect to what fraction of preallocated resource capacity job owners have consumed within a finite time window [12]. Existing schedulers such as Maui [11] and Simple Linux Utility for Resource Management (SLURM) [19] have built-in mechanisms for fairshare, but are not designed to support Grid environments that span multiple administrative domains, utilize heterogeneous schedulers, and require support for site autonomy in allocation policies. This work addresses a need for a global mechanism for Grid allocation policy enactment and presents FSGrid, a system for decentralized fairshare job prioritization that operates on global (Grid-wide) usage data and provides fairshare support to resource site schedulers.

Much work on Grid infrastructure have been directed towards virtualization of job and resource management, but many state of the art Grids still lack adaptability and flexibility in usage policy enactment. Rigidity in allocation mechanisms can effectively restrict many of the main use cases for Grids and, e.g., force end-users to perform manual resource selection to meet usage allocation criteria not supported by automated brokers. To facilitate the Grid vision of transparency in end-user resource utilization, policy enactment mechanisms that virtualize Grid-level usage allocation are required. To facilitate scalability in system deployment and administration, Grid fairshare enactment systems should also allow delegation of policy administration, use scalable fairshare calculation algorithms, and support management of Grid-scale volumes of usage data.

The proposed system provides a flexible capacity allocation policy model that maps organizational structures directly to policy specifications. The policy model separates policy specifications into local and global components, and delegates policy component administration to policy actors, e.g., Virtual Organizations (VOs) [10] and projects. Resource sites mount global policy components onto local policies to form policy trees, which allows global policy allocation updates (performed by policy actors) to be transparently propagated to resource sites.

The fairshare algorithm operates on usage data and compares consumed resource capacity to policy-defined capacity allocations. Fairshare is calculated for each level in the tree-based policy model and enforced top-down, ensuring fairshare balance between policy subgroups to take precedence over balance within subgroups. As local policy components are defined by site administrators and form top levels of policy trees, site owners retain full control over site resources.

FSGrid employs an architecture for distributed storage of usage data, and maintains periodic summaries at resource sites. With minimal demands on the content of usage data, the system integrates seamlessly with accounting systems, facilitating automated tracking of Grid-level usage data. FSGrid places a component close to resource site schedulers that local scheduling mechanisms invoke to replace existing fairshare calculations, imposing minimal changes to existing deployment environments. The main building blocks of FSGrid are:

- A Grid usage policy allocation model that supports recursive delegation of policy administration.
- An algorithm for efficient calculation of job prioritization from usage data and allocation policies.
- A decentralized and distributed architecture for dynamic fairshare policy enactment that implements the proposed fairshare algorithm and integrates with minimum intrusion into existing high-performance computing resource environments.

The evaluation presented in this work assumes a general model of Grid environments built on High-Performance Computing (HPC) resource sites, where jobs are fed from a batch system into a (cluster) scheduler. While this model is representative for many current Grid environments, the proposed system is not limited to HPC deployment Grids. The proposed system can be utilized by any system that performs execution order prioritization of jobs. The proposed system contains no functionality for advanced scheduling mechanisms, e.g., job preemption, and is to be viewed as an independent job prioritization component rather than a full policy enforcement or job scheduling mechanism.

Ordering of jobs with respect to differences between usage allocation and resource consumption allows schedulers to achieve a fairshare job prioritization semantic of "least favored first". This creates a global self-adjusting policy enactment mechanism that helps users receive resource capacity as defined by policy allocations. By definition of an allocation policy model for VOs, a fairshare algorithm operating on the policy model, and an architecture for distribution and decentralization of policy enforcement, we extend an existing fairshare mechanism to Grid level.

In FSGrid, we define fairness in terms of convergence of resource consumption to policy-defined preallocations over time. As a point of departure, this work builds on earlier efforts [9] where preliminary versions of the

policy model and algorithm are presented. A comprehensive differentiation and discussion of new and prior results is given in Section 6.2.

The rest of the paper is structured as follows. In the first sections we present the building blocks of the FSGrid system; a tree-based policy model (Section 2), an algorithm for efficient calculation of fairshare vectors (Section 3), and a decentralized architecture for scheduler-based Grid allocation policy enactment (Section 4). These are followed by a performance evaluation and a discussion of the proposed system in sections 5 and 6, and a survey of related work in Section 7. Finally, Section 8 outlines possible directions for future work, and the paper is concluded in Section 9.

2. A Tree-Based Usage Policy Model

Grids are typically formed through joint collaborations of autonomous resource sites. The amount of resources contributed to a specific collaboration normally differs between sites, and may vary over time. Grid policy models, i.e. mechanisms for mapping user identities to resource allocations, must allow site administrators to specify resource allocations on multiple levels, e.g., between local and Grid jobs, or different Grid collaborations (e.g., VOs). As Grid user bases are usually formed as VOs, Grid policy mechanisms are required to adapt to dynamic changes in VO structure.

As illustrated in Figure 1, FSGrid employs a model for specification of usage allocations in *policy trees*. Policy tree nodes contain tuples of *VO identity strings*, i.e. strings uniquely identifying a VO entity (e.g., a user or a project), and *usage share values*. A usage share value expresses a relative usage preallocation of resource capacity within a *policy group* (a set of VO identities that are policy tree siblings). The user $U4$ allocation of 0.2 in Figure 1 is interpreted as $U4$ being allocated 20 percent of whatever resource capacity (e.g., monthly CPU hours) is allocated to project $P1$.

This model allows VOs to map internal structure directly onto policy trees, and express both organizational hierarchy (tree structure) and relationships between and within policy groups (node share values) in a single structure. There are no limitations on policy organization other than VO identities being unique within tree levels, i.e. within policy groups or projects.

Expression of allocation quotas in relative usage metrics (e.g., share percentages) rather than absolute capacity metrics (e.g., CPU hours) virtualizes both the currency used in the system and allocation of resource site capacity. Separation of allocation quotas from resource capacity metrics allows policy quota allocations to be mapped to custom metrics, provides a semantic for reallocation of unused policy allocations, and insulates allocation enforcement mechanisms from volatility in resource site capacity.

As FSGrid policy trees express relative share ratios and make no assumptions of tree structure, policy trees can be constructed from multiple sources by mounting subtrees onto leaf nodes in a policy tree (see Figure 1). FSGrid makes a semantic distinction between *local* and *global share policies*. Local share policies are root policies defined by resource site administrators for individual resource clusters. Global share policies are independent policy trees defined by VOs, and are mounted onto local policy trees by resource site administrators (also illustrated in Figure 1). Local policies express what global policies to enact and relative resource allocations between them. Local share policies may have local queue components that allow site administrators to reserve resource capacity for local (non-Grid) jobs. Global policy trees express structure and allocations for VO components, e.g., groups, projects, and users.

As policy tree construction can be distributed and performed recursively, FSGrid delegates policy component (subtree) administration to policy actors. Delegation of policy specification allows policy actors, e.g., individual projects in a VO, to define policy components (subtrees) and mount these onto (leaf) nodes in parent policy trees, i.e. updating usage policy allocations without involving resource site administrators. Mounting global policy components to local policy trees allows resource site owners to subdivide and allocate resource site capacity shares to virtual organizations, which can further subdivide and allocate resource site capacity shares recursively within their organization.

Mounting of policy components onto policy trees does not violate the node peer uniqueness criteria of the policy model as subtree root nodes are overwritten by policy tree nodes in the mounting process. Paths in policy trees uniquely qualify both VO identities (the bottom path node) and chains of relationships between VO identities and policy ancestors.

3. A Grid Fairshare Algorithm

Fairshare scheduling relies on prioritization of jobs with respect to consumption of resource capacity preallocations. In FSGrid, job prioritization is performed through comparison of *fairshare vectors*, vectors of fairshare balance values calculated from paths in *fairshare trees*. Fairshare trees inherit structure from policy trees and are calculated from comparisons of policy trees and

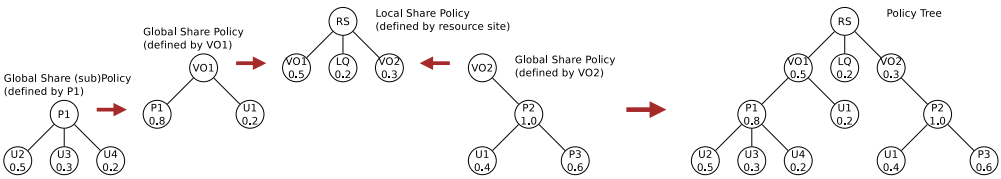


Figure 1: Delegation of policy specification to policy actors. Resource capacity allocations are subdivided recursively in usage shares. Resource site *RS* local share policy contains preallocated usage shares for virtual organizations (*VO*1 and *VO*2) and local job queue (*LQ*). Administration of policy components is delegated to organization and project administrators.

historical usage data. As paths in policy trees define ancestries of VO identities, comparison of fairshare vectors offer a computationally efficient way to simultaneously perform scheduling prioritization on multiple levels in policy trees.

The FSGrid fairshare algorithm performs calculation of fairshare vectors in two steps. First, a fairshare tree is calculated (once per resource site, illustrated in Figure 3) from an FSGrid policy tree and historical usage data. Second, fairshare vectors representing each VO identity in the system are calculated from the fairshare tree (once per VO identity, illustrated in Figure 4), and associated to jobs.

3.1. Fairshare Tree Calculation

Calculation of fairshare trees is done in two steps. First, a *usage tree* is constructed by recursively (bottom-up) replacing all node values in a policy tree with a cumulative usage sum. This value is calculated as the sum of all usage data found in the usage time window for the node VO identity and the sum of all child node values. To facilitate comparison of usage and policy data, node values are normalized to $[0, 1]$. Normalization is performed by replacing each node value with the node's relative share of the sum of all node values on the tree level. If no usage data is found (i.e. all sibling nodes have value zero), all sibling nodes in the tree level receive equal shares. Like in policy trees, all tree level node values sum to 1 after normalization.

Second, a fairshare tree is calculated by node-wise application of a *fairshare operator* on the policy and usage trees (illustrated in Figure 3). Fairshare operators compare share values from policy and usage trees and quantify a distance from the current to the ideal system fairshare balance state (where all users utilize resource capacity according to policy capacity preallocations). The policy and usage trees are identical in structure, and have node values in $[0, 1]$. Node values in the resulting fairshare tree are in $[-1, 1]$, and quantify a difference between policy usage preallocation and actual resource consumption (as defined by the fairshare operator used). Node value sign indicates direction (positive values underuse, negative overuse), and magnitude quantifies distance to policy-usage balance. All tree level node values in fairshare trees sum to 0. Like a policy tree contains all information required for policy enactment for a VO or a resource site, a fairshare tree contains all information required to perform fairshare prioritization of jobs on a resource site.

3.2. Fairshare Vector Calculation

Once a fairshare tree has been calculated, individual VO identity fairshare vectors are calculated (illustrated in Figure 4). As paths in fairshare trees uniquely define ancestries of VO identities, combining fairshare tree node values (top-down) along a tree path creates vectors that contain fairshare information for hierarchies of VO identities. After vector calculation, node values (x) are transformed to integer elements (y) as

$$y = floor((\frac{x+1}{2}) * 9999) \quad (1)$$

where

$$x \in [-1, 1]$$

$$y \in [0, 9999]$$

This results in integer vectors that can be serialized to strings and compared lexicographically. The value 9999 is an upper limit constant determining the numerical resolution of vector element integer representations.

For arithmetic comparison of vectors, where vectors are projected to one-dimensional value spaces, vectors are required to be of uniform length. Therefore, vectors are appended zero value elements until they reach maximum vector length (defined by fairshare tree depth). Zero is chosen as pad value as it expresses policy-usage

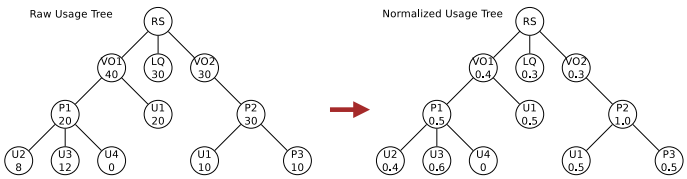


Figure 2: Construction of usage trees from (distributed) usage data is done in two steps: 1. Raw usage trees inherit structure from policy trees and node values are defined by cumulative summation of usage data for all usage identities at or below the current node. In the illustration, project *P*2 consumes 10 usage credits. 2. Usage trees are normalized to enable policy comparison through recalculation of node values as relative shares of node tree level usage data.

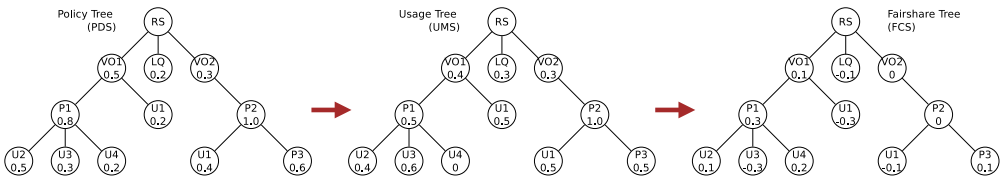


Figure 3: The FSGrid fairshare calculation algorithm. Fairshare trees are calculated by node-wise application of a fairshare distance measure operator on the policy tree and the usage tree, in the illustration the absolute fairshare operator $p - u$. Done once per Grid site and scheduling step.

balance in fairshare trees. As illustrated in Figure 4, padding is performed prior to transformation to integer vectors in the vector extraction algorithm.

Prioritization of jobs based on fairshare vector comparison results in hierarchical ranking of VO identities. Vector comparisons express differences in policy-defined preallocations and actual resource capacity consumption on multiple policy levels. As comparison is done on job ownership VO identity level, all jobs owned by the same VO identity receive the same priority.

In this framework, fairshare scheduling can be viewed as an optimization problem where the distance from each VO identity's usage state and the system balance axis are sought to be minimized simultaneously. By prioritizing jobs by fairshare distances, a scheduling policy of "least favored first" is enacted. The term *convergence* is in this context defined to refer to VO identities' resource capacity consumptions approaching policy-defined usage preallocations over time. Conversely, any mechanism counteracting system convergence in this context is defined to be *convergence noise*.

3.3. Fairshare Distance Measure Operators

For fairshare job prioritization, a mechanism to quantify differences between usage share preallocations and resource usage is required. To construct a metric for comparison, FSGrid defines a two-dimensional value space spanned by unit basis vectors for policy share preallocations (p) and resource capacity consumption (u). The system balance state, where resource consumptions equal policy allocations, forms an axis ($u = p$) transecting the value space diametrically.

By ordering VO identities by the distance from their current usage state (a function of p and u) to the system balance axis ($u = p$), a fairshare job prioritization order is established. For distance measurement, FSGrid defines a fairshare operator (d) constituted by an absolute (d_a) and a relative (d_r) component. To increase system configurability, relative operator component influences are regulated by a weight (k).

$$d = kd_a + (1 - k)d_r \tag{2}$$

where

$$d_a = p - u \tag{3}$$

$$d_r = \begin{cases} \left(\dfrac{p-u}{p}\right)^2 & \text{for} \quad u < p \\ 0 & \text{for} \quad u = p \\ -\left(\dfrac{p-u}{u}\right)^2 & \text{for} \quad u > p \end{cases} \tag{4}$$

$$k, p, u \in [0, 1]$$

$$d, d_a, d_r \in [-1, 1]$$

While any arbitrary operator (with arbitrary value

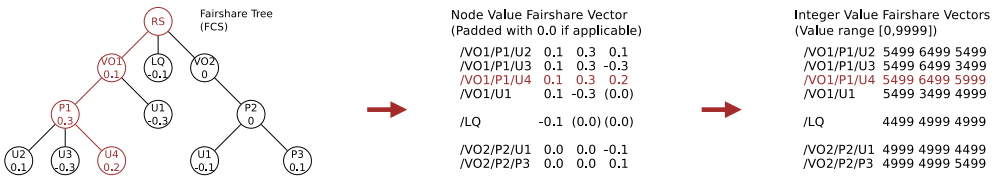


Figure 4: VO identity fairshare vectors are calculated and padded to uniform length. Node values ($[-1, 1]$) are converted to integer values ($[0, 9999]$). Fairshare vectors are calculated once per VO identity in fairshare trees.

space) may be chosen for fairshare distance measurement, operator selection impacts complexity and design of the system. For example, uniform and symmetric value spaces make distance interpretation intuitive, zero distance balance points facilitates padding of fairshare vectors, and unit distance magnitude facilitates scaling of fairshare balance values. Conceptually the absolute fairshare operator can be seen as a geometrical measurement of the distance between resource consumption and policy allocations in usage credits. The relative fairshare operator expresses a ratio between resource capacity consumption and policy preallocations.

The requirement for a combined operator stems from the behavior of the individual operator components. In situations where a VO identity does not utilize allocated capacity, the absolute operator degenerates and divides unused allocations evenly among VO identity peers. In situations where no usage data is available (e.g., at startup) the absolute operator favors users with large usage shares. In situations where zero policy allocations are assigned VO identities with reported usage, the relative operator yields a maximum distance regardless of differences in usage consumptions. By design, the relative operator has a higher resolution far from balance, and a lower resolution near balance. Combining the two operator components allows FSGrid to operate more robustly, and provides administrators the ability to customize the fairshare operator.

3.4. Combining Job Prioritization Mechanisms

As defined here, fairshare scheduling implies only a prioritization order for jobs. Jobs with low fairshare values may be scheduled if there are resources available and no jobs with higher fairshare prioritization value in queue. Jobs are by this mechanism not preempted or stalled, and fairshare scheduling is to be considered a soft scheduling mechanism. If policy fairness is more important than resource utilization, schedulers may combine fairshare prioritization with external mechanisms that, e.g., reject jobs with fairshare values below a certain threshold.

Some schedulers, such as Maui and SLURM, calculate a linear combination of multiple scheduling factors to determine job prioritization order. In these cases, a scalar fairshare rank value computed by the FSGrid algorithm can be used as a fairshare component in the linear combination. If so, the fairshare vector must be projected onto a limited value range to restrict the final prioritization value's range, which may affect the numerical stability of fairshare prioritization. To avoid this, projection of the fairshare balance values (fairshare vector elements) to a more restricted value range may be replaced with an algorithm that assigns values to vector elements according to group-wise sort order. This will project the fairshare vector to a truncated value range, preserving vector sort order while truncating distances between vectors uniformly. Typically, schedulers that use linear combinations of scheduling factors allow site administrators to configure weights to determine to what extent fairshare factors influence job prioritization.

4. A Decentralized Grid Fairshare Architecture

The policy model and algorithm of sections 2 and 3 provide a mechanism for fairshare prioritization of jobs based on usage allocation and resource consumption. As usage allocation policies are constructed from distributed policy components, and the algorithm operates on usage data from multiple distributed resource sites, an architecture managing distribution of data and computations is required.

As illustrated in Figure 5, the architecture of FSGrid is designed as a distributable Service-Oriented Architecture (SOA) where blocks of functionality in the FSGrid fairshare algorithm are identified and exposed as services. The FSGrid architecture contains three major blocks of functionality; policy administration, usage data monitoring, and fairshare vector calculation; which also constitute integration points between FSGrid and the deployment environment.

To facilitate computational efficiency and reduce communication overhead of the system, a number of ob-

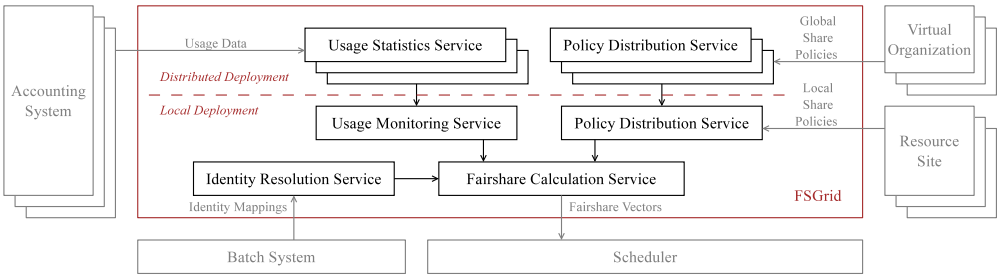


Figure 5: The FSGrid architecture. System functionality is segmented into distributable services. The system integrates with cluster schedulers, requires policy definitions from organizations, usage data from Grid accounting systems, and (optionally) identity mappings from batch systems. Deployment patterns are expected to be site-dependent.

servations about the interaction patterns of the functionality blocks can be made. Fairshare vectors are required for job prioritization and should be recalculated whenever updated policy allocations or usage data are available. As schedulers require access to fairshare vectors whenever scheduling decisions are made, e.g., when job queues change or periodic scheduling cycle events occur, the fairshare vector calculation block should be located close to the scheduler. The policy administration and usage data monitoring blocks are by nature distributed, but should for reduction of communication overhead have a cache component close to the fairshare vector calculation block.

The computational complexities of computing fairshare trees and vectors are low, and both operations can be precomputed and results cached, making them well suited for implementation in Web Services. Calculation of the fairshare tree is performed once per resource site and scheduling step, and calculation of fairshare vectors is performed once per VO identity owning a job in the scheduling queue.

Note that the design of the system does not assume coordination of component actions, or synchronization of distributed state, but rather realizes a set of autonomous components that combined form a decentralized fairshare architecture. Global fairshare resource allocation is enacted through concurrent, asynchronous local computations on distributed data.

To minimize the system deployment footprint, all services are designed to integrate non-intrusively with existing infrastructure and minimize network traffic required by the system. Service deployment patterns are expected to vary from site to site, but are recommended to be based on the pattern illustrated in Figure 5 to minimize communication overhead.

4.1. Architecture Components

As illustrated in Figure 5, FSGrid is constituted by five services and a set of plug-ins for scheduler prioritization, usage data submission, and identity resolution. To facilitate seamless integration into existing HPC deployments, the architecture is implemented in Java and exposes service functionality through WSDL SOAP Web Services deployed in Apache Axis2 service containers. Integration with HPC cluster schedulers (currently Maui and SLURM) is done through injection of FSGrid clients into scheduler exposed prioritization customization points.

4.1.1. Policy Distribution Service (PDS)

The Policy Distribution Service (PDS) provides a service interface to FSGrid usage policy allocations. Internally, the PDS collates policy components from multiple sources, e.g., XML files, HTTP web resources, other PDSs; assembles a policy tree; and publishes policies through the service interface. As multiple PDS may be chained, and data read remotely, the PDS provides a flexible mechanism for delegating policy definition to VO and site administrators. To FSGrid and FSGrid clients, the PDS provides an easy to use interface for policy retrieval, and can be to, e.g., monitor updates in policy allocations.

4.1.2. Usage Statistics Service (USS)

The Usage Statistics Service (USS) is designed to provide time-resolved histograms of usage data on a per-user basis. To reduce the amount of data, the service interface accepts updates in a format semantically equivalent to summaries of Open Grid Forum (OGF) Usage Records [17], and exposes usage summaries for requested time windows. Internally, the USS stores

usage histograms for known users in a database, and maintains a usage summary cache to minimize invocation response time. The USS is the only required part of FSGrid that receives input data from the surrounding system environment. As usage data constitutes the currency that drives FSGrid fairshare, it is vital to FSGrid system coherency that each job usage record is only reported to a single USS. As the USS provides a histogram-based view of historical usage data, it can be used by FSGrid services and clients to assess usage statistics for individual VO identities on individual resource sites.

4.1.3. Usage Monitoring Service (UMS)

The main task of the Usage Monitoring Service (UMS) is to provide a service interface for computation of (normalized) usage trees from policy trees. Internally the UMS compiles data from a set of known USSs, maintains a database of USS usage summaries, a time-resolved per-user usage cache, a cache of previously known policy trees, and agents to monitor USSs and precompute usage trees. The UMS also maintains a customization point for moderation of usage data influence through a time window and usage decay function plug-in. The UMS provides an interface for summarizing usage records from multiple (USS) data sources and mapping these to (provided) usage policies, and can be used by FSGrid services and clients to get normalized usage data views.

4.1.4. Identity Resolution Service (IRS)

Key to enabling fairshare scheduling of jobs in FSGrid is to be able to access historical usage records for VO identities. As VO identities may be translated to local cluster or site users when jobs are dispatched to batch queues, schedulers may lack access to VO identities. The IRS exposes an interface for storing and accessing VO identity to job associations, and is primarily used to resolve job ownerships. Use of the IRS in FSGrid is optional. If a scheduler has access to VO identity job ownership data, these may be used directly when requesting scheduling prioritization information.

4.1.5. Fairshare Calculation Service (FCS)

The Fairshare Calculation Service (FCS) offers a flexible service interface that provides access to the FSGrid policy-based fairshare tree, fairshare vectors for specified VO identities (or jobs), and preformatted fairshare tuples that contain VO identities, fairshare vectors, and scalar fairshare prioritization values. The rich interface of the FCS is designed to facilitate flexibility in implementation of scheduler integration plug-ins. Internally, the FCS maintains caches for job identifier to VO identity maps, policy, usage, and fairshare trees, as well as agents for monitoring services (PDSs and UMSs) and precomputing fairshare trees. The FCS allows configuration of UMS and PDS connections, PDS deployments, and monitoring scheduling intervals.

4.1.6. Integration Plug-Ins

In addition to the services of FSGrid, a set of integration plug-ins is also considered part of the FSGrid architecture. Depending on the FSGrid deployment environment, integration plug-ins for scheduler job prioritization, usage data submission, and VO identity resolution may be required. Design of scheduler plug-ins depend on scheduler architecture, but typically consist of an FCS client implemented in the same language as the scheduler and possibly routines for calculation, transformation, and caching of fairshare prioritization data. Design of plug-ins for usage data submission and VO identity resolution depend on accounting system and scheduler architecture, and will typically consist of USS and IRS clients.

As many Grid computing environments build on existing HPC deployments, which typically are required to maintain HPC interfaces (e.g., batch systems) in coexistence with Grid interfaces, it is vital to design Grid systems to impose a minimum intrusion level when integrating Grid components with existing HPC deployments. The FSGrid architecture is designed to have as few and simple integration points as possible while still maintaining compatibility with a general model for HPC deployment based Grid environments.

Typical Grid FSGrid integrations include

- Replacement of a local scheduler (fairshare) job prioritization mechanism with an FCS invocation client.
- Injection of a mechanism for submission of usage data to the USS. This can be done in multiple ways, e.g., through a scheduler job monitoring plug-in, or a resource site or Grid accounting system.
- Optional injection of a job ownership resolution component. If VO identity job ownership data is not available to the scheduler, a job ownership mapping between job and original VO identity can be stored in the IRS. This data can be submitted at any point prior to invocation of the FCS. Submission is typically expected to be done by the system responsible for translation of VO identities to local resource site users, e.g., a batch system.

4.2. (Concurrency in) Data and Control Flow

Data and control flows of an FSGrid deployment consist of five autonomous and concurrent processes:

1. A set of PDSs monitors a set of data sources and periodically compiles policy trees.
2. A set of USSs receives (summarized) usage reports for jobs and builds time-resolved usage histograms.
3. A UMS monitors a set of (local or remote) USSs, periodically retrieves updates, and assembles usage summaries. The UMS precomputes usage trees for known policy trees, and on demand for unknown policy trees (which are added to the cache structure).
4. An IRS receives VO identity job ownership data and maintains a directory for ownership resolution.
5. An FCS monitors a PDS and periodically retrieves policy trees and calculates usage (via a UMS) and fairshare trees. The FCS maintains a cache of precomputed fairshare vectors (based on precomputed fairshare trees), and does not compute fairshare vectors for unknown VO identities.

The data required to drive the system, usage data and usage policy allocations, are provided by accounting systems and VO, project, and resource site administrators respectively. FSGrid assumes that jobs are scheduled in an order influenced by fairshare prioritization and that usage costs for all jobs are reported to the system. Should resource sites utilize FSGrid to prioritize jobs without reporting usage data, resource consumption costs for jobs running on such sites do not contribute to fairshare calculation results and imbalances in global resource consumption may occur. Conversely, should resource sites report usage data without utilizing FSGrid as a job prioritization mechanism, global fairshare convergence will suffer oscillations correlated in size to resource site capacity. As the core balancing mechanism of FSGrid is self-adjusting, global fairshare balance will converge over time.

Specification of usage policies can be seen to be a largely manual process, while usage data submissions are expected to be fully automated. Through these five processes, the FSGrid system provides an automated, decentralized, and self-adjusting mechanism for Grid-wide fairshare enactment of usage policy allocations.

4.3. Time Window and Decay Function

As illustrated in Figure 6, FSGrid defines a finite usage data time window (typically a configurable amount of days into the past) to restrict the influence of historical usage data on the fairshare mechanism. As also illustrated, FSGrid employs a customizable usage decay function to modulate how usage statistics influence the fairshare mechanism. The time window width limits the scope of usage statistics influence (data outside the time window does not affect FSGrid behavior). The granularity of the time window histogram slots affect the resolution of the fairshare mechanism. The usage decay function modulates usage statistics by, e.g., increasing or decreasing influence of more recent usage statistics on system behavior. In the FSGrid architecture, both time window parameters are configurable, and the usage decay function is exposed as a customization point in the UMS. Further study of the impact of usage decay functions in this context is subject for future work.

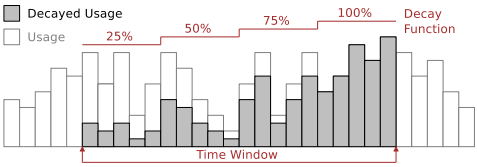


Figure 6: Usage data histogram time window. Usage decay functions modulate influence of usage data.

5. Evaluation

To evaluate core system functionality and isolate noise sources (i.e. mechanisms counteracting system convergence), a number of tests designed to quantify aspects of FSGrid's technical performance are employed. These tests are run in an emulated system environment and are designed to introduce and illustrate system mechanics. As the purpose of this evaluation is to evaluate system ability to enact policy allocations in a distributed environment rather than demonstrate system integration in a production deployment, use of an emulated system environment is sufficient. In the evaluation, the follow tests are performed:

- Noise characterization tests (Section 5.1). Investigate and characterize FSGrid noise mechanisms.
- Noise interaction tests (Section 5.2). Investigate interaction between different noise types and illustrate impact of system deployment patterns on system performance.
- Policy enactment tests (Section 5.3). Investigate FSGrid ability to enact policy allocations in decen-

tralized multi-site deployments employing multiple asynchronized concurrent schedulers. Quantify and evaluate FSGrid ability to adapt to dynamic changes in policy allocations and distributed system failures.

- Scalability tests (Section 5.4). Investigate FSGrid ability to cope with realistically sized policy allocations and quantify system scalability in the presence of large amounts of usage data and updates.

All evaluation tests are performed on a set of four identical 1.8 GHz quad core AMD Opteron CPU, 4 GB RAM machines, interconnected using a Gigabit Ethernet network. For functionality tests, an additional set of four identical 2 GHz AMD Opteron CPU, 2 GB RAM machines, interconnected with a 100 Mbps Ethernet network are used. All machines are running Ubuntu Linux and Axis2 1.5. The Java version used in tests is 1.6, and Java memory allocation pools range from 512 MB to 1 GB in size. For system integration tests SLURM 2.1.2 is employed as batch system and cluster scheduler.

Functionality tests are performed using a discrete-event simulator emulating an execution environment consisting of a batch system, a cluster scheduler, a cluster, and an accounting system. The batch system registers (in the IRS) and feeds the scheduler sets of jobs. The scheduler invokes the FCS to prioritize jobs and allocates them to cluster hosts. The accounting system submits usage reports to the USS upon job completions. Job start and end timestamps are used to evaluate FSGrid ability to enact resource capacity allocations.

Simulation job arrival models saturate scheduling queues in the sense that schedulers have access to at least one job for each usage policy VO identity at all times. Single-site system emulations are run on a single host, multi-site system emulations as a set of non-communicating systems run concurrently on multiple hosts. Cross-site synchronization is performed exclusively by UMSs, which have access to USSs for all sites, emulating a distributed Grid configuration.

All tests are, unless stated otherwise, run using identical parameter sets and the policy tree illustrated in Figure 1. USS and UMS update intervals are set to 1 second, usage time windows are 10 slots wide and set to a granularity of 1 hour (system wall clock time), all clusters have a single host, and job lengths are either fixed to 1 or stochastic and uniformly distributed between 1 and 5000 time units long. To eliminate them as parameters in measurements, absolute and relative fairshare operators are equally weighted ($k = 0.5$). Usage decay is disabled (i.e. usage decay function is constant $y = 1$), and the usage cost metric used is job length (CPU time).

Job failures do not affect FSGrid convergence rates as failed jobs do not get reported to the accounting system and appear as diminished resource capacity.

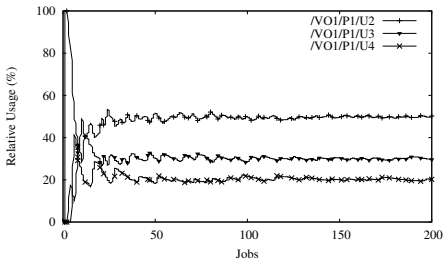
5.1. Noise Characterization

As we refer to system ability to over time enact policy-defined resource capacity allocations as system convergence (to policies), we define any mechanism counteracting this process as convergence noise. To illustrate system convergence to policy allocations, we isolate policy (sub)groups, i.e. groups of nodes with a common parent, and render cumulative resource consumption for individual VO identities as a function of number of jobs run in the group. To maximize the influence of noise in measurements, we isolate the policy subgroup containing the VO identity with the lowest total usage share ($P1$ in Figure 1).

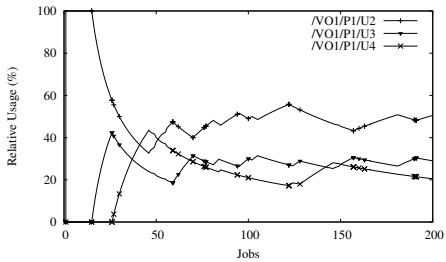
In FSGrid, there are two primary mechanisms counteracting system convergence, *variance in job usage costs* and *usage data update latencies*. To isolate impact of variance in job usage cost, we emulate a single-site FSGrid deployment with stochastic job usage costs drawn from a uniform [1,5000] probability distribution. To eliminate impact of usage data update latencies on system convergence, each scheduling step is delayed to allow usage data updates from prior jobs to propagate to the FCS between scheduling steps. As illustrated in Figure 7a, usage cost variance amplifies oscillations in system convergence. When compared to ideal convergence, differences in usage costs manifest as additive noise in convergence adjustments. In Figure 7a, job usage cost variance noise is illustrated as vertical offsets in convergence oscillations.

To isolate impact of usage data update latencies, we emulate a single-site deployment with uniform job usage cost (cost = 1) and UMS and FCS update delays designed to allow approximately 10 jobs to be scheduled between FCS usage data updates. As the FSGrid job prioritization mechanism operates on usage data for completed jobs, i.e. has no memory for recent scheduling decisions or prediction mechanism for costs of running jobs, usage data update latencies result in multiple subsequent scheduling decisions being taken on the same usage data. As illustrated in Figure 7b, this results in amplifying convergence oscillations and significantly lowering system convergence rate. When compared to ideal convergence, usage data update latencies manifest as multiplicative noise in convergence adjustments and a divisible reduction in convergence rate.

Prior work [9] suggests that including cost for scheduled and running jobs in prioritization calculations reduce impact of usage data update latency noise on sys-



(a) Job usage cost variance noise. Variations in job usage costs cause convergence adjustments to overshoot and amplify convergence oscillations.



(b) Usage data update latency noise. Update latencies reduce granularity of convergence adjustments and delay system convergence.

Figure 7: Noise characterization. Cumulative resource consumption as function of scheduled and run jobs. Convergence to usage policy allocations for VO identities designated in legend. Illustration capped to region of interest.

tem performance. Future work includes evaluation of different strategies for inclusion of this approach in multi-site FSGrid deployments.

5.2. Noise Interaction

Under realistic FSGrid operational settings both job usage cost variance and usage data update latencies are likely to be present. To investigate noise interaction, we emulate a single-site FSGrid deployment with stochastic job lengths and usage update latencies. As illustrated in Figure 8a, usage data update latencies add a multiplicative component to usage cost variance noise. Impact of noise is amplified by lowered convergence rate.

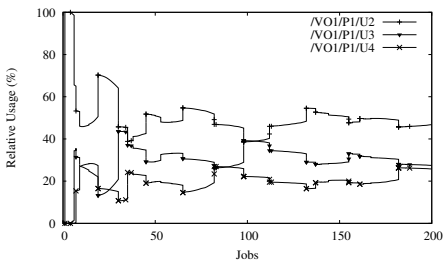
To evaluate noise interaction in decentralized Grid environments, we emulate a four-site FSGrid deployment with stochastic job lengths and usage data update latencies. As illustrated in Figure 8b, parallelism of concurrent scheduling amplifies update latency noise, in this experiment delaying system convergence by a factor of 10. As the number of jobs scheduled between usage data updates determine impact of update latency noise, large numbers of computational resources per scheduler skew system convergence at startup. As job lengths constitute lower bounds for usage data update latencies, excessive job lengths amplify update latency noise. For multi-site settings, concurrent scheduling with synchronized update schedules amplify update latency noise. Conversely, asynchronicity in multi-site update schedules allow parallel processing of usage updates to increase update frequencies and mediate impact of usage data update latency noise.

These experiments are run in an artificial environment, but outline a few interactions between mechanisms in the FSGrid fairshare job prioritization system. Scheduling jobs after the principle of "least favored first" creates a self-adjusting system that over time distributes resource capacity after policy allocations. Noise from job usage cost variance and update latencies lower system rate of convergence by affecting the convergence adjustments (i.e. order in which jobs are run). Number of hosts, sites, job lengths, as well as frequency and synchronicity of usage data update schedules may serve to amplify convergence noise. Over time, relative impact of each noise source and type lessen, as more usage data affect scheduling prioritization. As long as usage data time windows are large enough to contain enough data for the system to converge, the system remains stable.

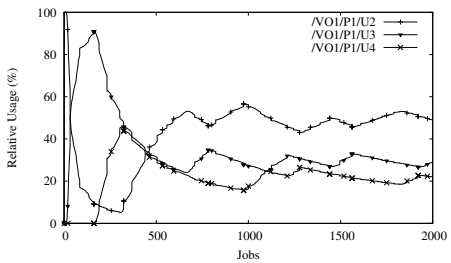
5.3. Policy Enactment

To evaluate system ability to respond to external events such as dynamic changes in site availability or policy allocations, we emulate an eight-site FSGrid deployment with stochastic job lengths and usage data update latencies over an extended period of time. To study impact of site volatility, four sites are removed after approximately 25000 jobs are scheduled. After approximately 50000 jobs are scheduled, the local allocation policy RS is altered to transfer 10 percent from each of the allocations for $VO1$ and LQ to $VO2$.

As illustrated in Figure 9a, eight concurrent schedulers cause significant initial convergence noise. At approximately 25000 jobs four schedulers are removed, and convergence noise is reduced (also visible at approximately 10000 jobs in Figure 9b). At approximately 50000 jobs the usage policies are updated and all schedulers adapt to new scheduling priorities. It takes approximately the same amount of jobs currently in the time window to reach the level of convergence achieved

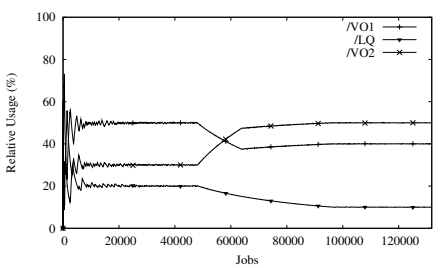


(a) Interaction of usage cost variance and update latency noise. Impact of usage cost variance noise amplified by delayed convergence.

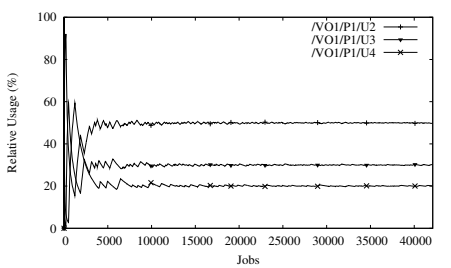


(b) Concurrent scheduler noise augmentation. Impact of usage data update latency noise amplified by parallelism in scheduling.

Figure 8: Noise interaction. Cumulative resource consumption as function of scheduled and run jobs. Convergence to usage policy allocations for VO identities designated in legend. Illustration capped to region of interest.



(a) Adaptation to site failures and updated usage policy allocations.



(b) Adaptation to site failures and subgroup isolation.

Figure 9: Policy enactment. Cumulative resource consumption as function of scheduled and run jobs. Convergence to usage policy allocations for VO identities designated in legend. Illustration capped to region of interest.

before the policy shift. As also illustrated in Figure 9a, convergence rate is a function of the relative share ratio, the VO identity with the lowest policy allocation (LQ) converges slowest.

As illustrated in Figure 9b, which illustrates policy group $P1$ of Figure 9a, altering the policy allocation of an individual policy group does not affect other groups in the same policy tree. Note that this simulation contains multiple shifts of the usage data time window, which do not visibly affect system convergence.

5.4. Scalability Tests

To evaluate FSGrid ability to function in production environments, we run large scale tests over longer periods of time using realistically sized policy allocations and system configurations. System convergence is validated for tests using policy allocations with thousands of users running millions of jobs.

For tests using a balanced policy tree (symmetrically distributed users with equal allocation shares) containing 1000 users, 100 projects, and 10 VOs, the system is shown to converge and exhibit stable performance consistent with the behavior illustrated in tests using smaller policy trees. System behavior is shown to be deterministic and stable in tests using more than 4 million jobs. First 5000 jobs of such a test are shown in Figure 10. Tests using symmetric policy trees with 1000 users and equal allocation shares show faster convergence rates than tests using small asymmetrical policy trees. Exact system convergence rate depends on a number of factors including, e.g., policy tree shape, usage allocation share variance, job length variance, update delays, and site synchronicity, and is considered out of scope for this work. Further scalability and integration tests in production environments, as well as analysis of convergence factors and formulation of convergence rate formulas are subject for future work.

6. Discussion

The system evaluation of Section 5 demonstrates technical aspects of FSGrid and shows how Grid-wide fairshare job prioritization can be realized. While this evaluation is performed in an emulated environment, the evaluation demonstrates key aspects of system functionality, scalability of system mechanisms, and system ability to enact policy usage allocations. Full-scale testing and evaluation of the system in production environments is subject for future work.

The remained of this section discusses fairness in scheduling, differences between global and local fairshare scheduling, and relates this work to earlier efforts.

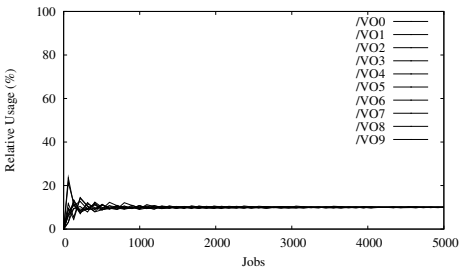


Figure 10: Convergence of total resource consumption for 10 different VOs. The system comprises a total of 1000 users symmetrically distributed with equal allocation shares over 100 projects and 10 VOs. Over 4 million jobs are run with stable convergence behavior. Illustration capped to region of interest (first 5000 jobs).

6.1. Global and Local Fairshare

FSGrid provides a framework for global (Grid-wide) fairshare scheduling. Existing schedulers (such as SLURM and Maui) contain fairshare prioritization mechanisms, but are designed for local (resource site) fairshare scheduling, using a single share policy, a common scheduler technology, and consider only local usage data. FSGrid offers a model for global fairshare where sites may have different share policies, use different schedulers, and operate on global usage data.

Local fairshare is often sufficient to manage job prioritization for HPC, as user identities are normally associated with a single site. In Grids, computational resources from several sites are aggregated into a common pool of resources available to all users of that Grid. The jobs of Grid users should be given the same prioritization regardless of which site in the Grid the job is submitted to, and the combined use across the Grid should be used for fairshare. For this, a global fairshare mechanism is required.

Although FSGrid is designed for global fairshare, the system can also be used for local fairshare job prioritization. Compared to local fairshare, global fairshare has additional challenges that include:

- Operation across administrative domains.
- Heterogeneity in technology, performance, availability, scheduling models, and allocation models.
- Greater usage data volumes.

- Usage and policy data updates has to be propagated to all participating sites, and each update may trigger a fairshare recalculation.
- Many different actors (site schedulers) depend on the same fairshare values simultaneously.

FSGrid may be deployed and configured in many different ways to suit individual environments. Any scheduler capable of calling Web Services can be integrated with FSGrid, and the policy model is based on site-specific local policies under full control of local administrators. To cope with Grid data volumes, fairshare values for known users are precomputed and cached. Similarly, as summaries of all usage for each slot in the time window are maintained at user level, usage updates only trigger recalculation of summaries for affected slots. Updated summaries are used in computation of fairshare values in following iterations.

Summaries and precomputed fairshare values are maintained at each FSGrid installation (normally one per site), and each scheduler can be served by a local FSGrid instance. As summaries are stored at each FSGrid instance, each site can have differently sized time windows (see Section 4.3) and purge usage data outside of the time window without affecting other sites.

6.2. Prior Work

This work builds on earlier efforts presented in [9], where preliminary versions of the policy model and simulations of the algorithm are presented. The main contributions of this work are a proposed architecture for realization of a decentralized system based on this algorithmic model, adaptations of the policy model and algorithm to facilitate distribution of the system, and a technical evaluation and analysis of the system. The architecture is designed for use in large scale environments, and focused on scalability through distribution and parallelization of data management and computations. Modifications of preliminary results presented in [9] include:

- Realization of the system. Prior work presented a simulation of the algorithm. This work presents a realization of a distributed system that is evaluated in an emulated environment.
- Extension of the fairshare algorithm with a framework for more fine-grained differentiation of resource consumption (fairshare operators).
- Increased precision of vector elements to allow a greater resolution in fairshare vectors.
- Reformulation of policy formats and interpretations to allow for dynamic updates of allocation policies.

7. Related Work

The fair Share scheduler [12] introduces the concept of user-level fair resource allocation in uni-processor sharing environments. The work introduces concepts such as fairness over time, support for different entitlements for different users, hierarchical policy structures, and sub-group isolation.

An evaluation of fair share in clusters or HPC systems is presented in [15]. Applicability of previous work on uni-processor sharing [12] to Grids or HPC systems is analyzed and simulated using logged data for thousands of real jobs. Effects of fair share on job prioritization are found to be small, partly because average system utilization is not high enough to cause enqueueing of jobs and partly because other factors (e.g. CPU requirements) are more deciding than differences in job priority.

Buyya et al. present a variation of the original FSGrid resource allocation strategy of [13]. Sub-groups (such as a sub-VO) may have dedicated resource allocations that can be used in conjunction with allocation of ancestor nodes. Consumption cost is used to select which allocation to use if several suitable alternatives are available. The aim is to maximize resource utilization, and fair allocation of resources between siblings in a hierarchy is not taken into consideration. An extension that also provides fair resource sharing is presented in [14]. Node job arrival rates are assumed to be known for all nodes in the system and the problem is formulated as a waiting time minimization problem. Jobs that cannot be immediately scheduled are rejected, and as jobs arrive with an assumed Poisson distribution, minimizing waiting time affects job acceptance rate. In [14] fairness is measured by job acceptance rate for different users.

Fair Execution Time Estimation (FETE) scheduling [2] is another take at Grid fair scheduling, where jobs are scheduled according to expected completion time as if running on a time-sharing system instead of a space-sharing system. Focus of this approach is to minimize risk of missing deadlines for submitted jobs.

Another hierarchical model presented in [8, 6], is used to allot resources from different sites to VOs and from VOs to users. Each sub-allocation includes both a burst allocation and an epoch allocation to control resource consumption in short- and long-term, respectively. GRUBER [7], is an architecture of this model that acts as a broker for resource usage Service Level Agreements (SLAs).

DI-GRUBER [5] extends GRUBER and adds support for distributed VO policy decision points. In these systems, VO polices are analogous to global policies in FSGrid and manage suballocation of resources within a VO. In contrast to FSGrid (where each site loads and enforces global policies), DI-GRUBER calls external decision points for VO policy decisions.

An evaluation of Grid resource allocation mechanisms is presented in [16]. Three different mechanisms considered are *volunteer*, *agreement-based*, and *economic* resource allocation. The agreement-based allocation mechanism used in the evaluation is based on earlier FSGrid work ([9]). The agreement based method was shown to have better overall resource utilization and suffer less degradation from high numbers of users compared to alternative approaches.

A comprehensive study on share scheduling mechanisms is presented in [3]. The study includes a thorough mathematical analysis of different strategies for share scheduling in uniprocessor, multiprocessor, and distributed systems.

More algorithms for fair scheduling in Grids are presented in [4]. The primary objective is to adhere to task deadlines. All tasks receive an equal share of resources regardless of number of jobs submitted. Excess resources not required by a task are divided equally among tasks that require more resources. Tasks may also be weighted to receive more than their equal share of available capacity.

A game-theoretic approach to fair Grid resource management is presented in [18]. This work considers the case where local scheduling decisions may be taken to optimize the system from the local schedulers point of view, and evaluates consequences of different levels of local scheduler autonomy in terms of (fair) scheduling.

Fair decentralized scheduling for Desktop Grids is presented in [1]. Fairness in this case is defined as minimizing the overhead of running each task on a shared infrastructure compared to a dedicated one. FSGrid defines fairness differently, and measures fairness as the difference between the expected and actual share of total resource consumption.

8. Future Work

A number of possible directions for future work are identified. Further investigation of trade-offs between FSGrid convergence parameters is expected to increase understanding of system behavior. Evaluation of impact of update latencies, usage decay functions, and job scheduling patterns are likely to influence parameterization and further development of the system. Evaluation of experiences from integration of the system in production use Grid deployments are expected to be of interest for further development of the system. Incorporation of scheduled and running jobs in fairshare job prioritization is likely to reduce impact of usage data update latency noise. Integration of the FSGrid job prioritization mechanism with additional cluster scheduling systems is expected to be of interest for system adoption.

9. Conclusion

In this work we present FSGrid, a decentralized system for fairshare-based Grid usage policy enactment built on three main contributions; a flexible policy model, a scalable fairshare calculation algorithm, and a decentralized architecture for parallelized fairshare prioritization of jobs. The system design is presented in detail, along with a performance evaluation and a discussion of the system.

The policy model supports mapping of VO structures onto policies, delegation of policy specification, and virtualization of usage credits. The fairshare calculation algorithm is self-adjusting and noise-stable, virtualizes resource site capacity, provides subgroup isolation within policy allocations, and adapts to changes in usage data and policy allocations. The architecture of the system is designed to facilitate decentralization of system deployments, precomputation and caching of scheduling data, and integrates non-intrusively with existing scheduling systems. The presented system can be utilized for job prioritization and scheduler-based policy enactment in Grid and HPC environments.

The performance evaluation illustrates the FSGrid policy allocation mechanism, demonstrates the feasibility of the approach, and identifies factors that manifest as noise in system convergence. The evaluation investigates trade-offs between convergence noise factors, and suggests how impact of these factors may be reduced. The discussion relates the proposed system to similar work and systems, and outlines the role of the system in Grid environments.

Acknowledgments

The authors extend their gratitude to Peter Gardfjäll for prior work, Lars Karlsson and Lars Larsson for feedback and discussions, and Raphaela Bieber-Bardt for work related to the project. The authors also acknowledge Tomas Ögren and Åke Sandgren for technical assistance to the project.

This work has been done in collaboration with the High Performance Computing Center North (HPC2N)

and has been funded in part by the Swedish Research Council (VR) under Contract 621-2005-3667, the Swedish National Infrastructure for Computing (SNIC), and the Swedish Government's strategic research project eSSENCE. The authors also acknowledge the Lawrence Berkeley National Laboratory (LBNL) for supporting the project under U.S. Department of Energy Contract DE-AC02-05CH11231.

References

[1] J. Celaya and L. Marchal. A Fair Decentralized Scheduler for Bag-of-Tasks Applications on Desktop Grids. In *CCGRID '10: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 538–541, Washington, DC, USA, 2010. IEEE Computer Society.

[2] E. Dafouli, P. Kokkinos, and E. A. Varvarigos. Fair Execution Time Estimation Scheduling in Computational Grids. In P. Kacsuk, R. Lovas, and Z. Nmeth, editors, *Distributed and Parallel Systems*, pages 93–104. Springer US, 2008.

[3] J. De Jongh. *Share scheduling in distributed systems*. PhD thesis, Delft Technical University, 2002.

[4] N. Doulamis, E. Varvarigos, and T. Varvarigou. Fair Scheduling Algorithms in Grids. *IEEE Transactions on Parallel and Distributed Systems*, 18:1630–1648, 2007.

[5] C. Dumitrescu, I. Raicu, and I. Foster. DI-GRUBER: A Distributed Approach to Grid Resource Brokering. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 38, Washington, DC, USA, 2005. IEEE Computer Society.

[6] C. L. Dumitrescu and I. Foster. Usage Policy-Based CPU Sharing in Virtual Organizations. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 53–60, Washington, DC, USA, 2004. IEEE Computer Society.

[7] C. L. Dumitrescu and I. Foster. GRUBER: A Grid Resource Usage SLA Broker. In J. C. Cunha and P. D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 465–474. Springer Berlin / Heidelberg, 2005.

[8] C. L. Dumitrescu, M. Wilde, and I. Foster. A model for usage policy-based resource allocation in grids. *Policies for Distributed Systems and Networks, 2005. Sixth IEEE International Workshop on*, pages 191 – 200, jun. 2005.

[9] 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*, pages 221–229. IEEE CS Press, 2005.

[10] 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):200–222, 2001.

[11] D. Jackson, Q. Snell, and M. Clement. Core Algorithms of the Maui Scheduler. In D. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 87–102. Springer Berlin / Heidelberg, 2001.

[12] J. Kay and P. Lauder. A fair Share scheduler. *Commun. ACM*, 31(1):44–55, 1988.

[13] K. H. Kim and R. Buyya. Policy-based Resource Allocation in Hierarchical Virtual Organizations for Global Grids. In *SBAC-PAD '06: Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing*, pages 36–46, Washington, DC, USA, 2006. IEEE Computer Society.

[14] K. H. Kim and R. Buyya. Fair resource sharing in hierarchical virtual organizations for global grids. In *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 50–57, Washington, DC, USA, 2007. IEEE Computer Society.

[15] S. D. Kleban and S. H. Clearwater. Fair Share on High Performance Computing Systems: What Does Fair Really Mean? In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 146, Washington, DC, USA, 2003. IEEE Computer Society.

[16] S. Krawczyk and K. Bubendorfer. Grid resource allocation: allocation mechanisms and utilisation patterns. In *AusGrid '08: Proceedings of the sixth Australasian workshop on Grid computing and e-research*, pages 73–81, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[17] R. Mach, R. Lepro-Metz, S. Jackson, and L. McGinnis. Usage Record - Format Recommendation, 2007.

[18] K. Rzadca, D. Trystram, and A. Wierzbicki. Fair Game-Theoretic Resource Management in Dedicated Grids. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 343–350, Washington, DC, USA, 2007. IEEE Computer Society.

[19] A. Yoo, M. Jette, and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In D. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. Springer Berlin / Heidelberg, 2003.

VIII

Paper VIII

Increasing Flexibility and Abstracting Complexity in Service-Based Grid and Cloud Software

Per-Olov Östberg and Erik Elmroth

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

Abstract: This work addresses service-based software development in Grid and Cloud computing environments, and proposes a methodology for Service-Oriented Architecture design. The approach consists of an architecture design methodology focused on facilitating system flexibility, a service model emphasizing component modularity and customization, and a development tool designed to abstract service development complexity. The approach is intended for use in computational eScience environments and is designed to increase flexibility in system design, development, and deployment, and reduce complexity in system development and administration. To illustrate the approach we present case studies from two recent Grid infrastructure software development projects, and evaluate impact of the development approach and the toolset on the projects.

Key words: Service-Oriented Architecture, Grid Computing, Cloud Computing, Service-Oriented Component Model.

INCREASING FLEXIBILITY AND ABSTRACTING COMPLEXITY IN SERVICE-BASED GRID AND CLOUD SOFTWARE

Per-Olov Östberg, Erik Elmroth
Department of Computing Science, Umeå University, SE-901 87, Umeå, Sweden
p-o@cs.umu.se, elmroth@cs.umu.se

Keywords: Service-Oriented Architecture, Grid Computing, Cloud Computing, Service-Oriented Component Model.

Abstract: This work addresses service-based software development in Grid and Cloud computing environments, and proposes a methodology for Service-Oriented Architecture design. The approach consists of an architecture design methodology focused on facilitating system flexibility, a service model emphasizing component modularity and customization, and a development tool designed to abstract service development complexity. The approach is intended for use in computational eScience environments and is designed to increase flexibility in system design, development, and deployment, and reduce complexity in system development and administration. To illustrate the approach we present case studies from two recent Grid infrastructure software development projects, and evaluate impact of the development approach and the toolset on the projects.

1 INTRODUCTION

In this paper we discuss service-based software development, propose a methodology for Service-Oriented Architecture (SOA) design, and present a toolset designed to abstract service development complexity. To illustrate the approach we present case studies from two recent Grid infrastructure software development projects, and evaluate impact of the development approach and the toolset on the projects.

Grid and Cloud computing alter and introduce new software requirements for computational eScience applications. Increasingly, eScience software now require the ability to be flexible in deployment, dynamically reconfigured, updated through modules, customized, and have the ability to integrate non-intrusively in heterogeneous deployment environments. At the same time, software size and complexity is growing in multiple dimensions (Kephart and Chess, 2003), and limitations and complexity in current service development tools increase development overhead. Software development projects include more developers, require more coordination, and result in more complex systems. Software administration is growing in complexity and new mechanisms for software administration are required.

This work addresses an identified need for scalability in more dimensions than just performance, and builds on prior efforts in service composition-based software design (Elmroth and Östberg, 2008) and a model of software sustainability based on the notion of an ecosystem of software components (Elmroth et al., 2008). We explore an approach to service-based software development based on separation of service functionality blocks, reduction of software complexity, and formulation of architectures as dynamically reconfigurable, loosely coupled networks of services. To support the approach, we present a toolset designed to abstract complexity in service description and development. The approach is illustrated in two case studies from recent development projects.

The rest of this paper is structured as follows: Section 2 overviews related and prior work. Section 3 discusses software development in Grid and Cloud computing environments, and illustrates the need for flexibility in design, development, and deployment of eScience software. Section 4 proposes a methodology for service-based software design, and Section 5 presents a toolset for service development. To illustrate the methodology and use of the toolset, Section 6 presents case studies of two recent software development projects, and the paper is concluded in Section 7.

2 RELATED AND PRIOR WORK

This work builds on a service composition model and a set of architectural design patterns presented in (Elmroth and Östberg, 2008), and further refines a software design and development model based on the notion of an ecosystem of software components (Elmroth et al., 2008). The approach is designed to facilitate software flexibility and adaptability, and promote software survival in natural selection settings. While the approach does not define explicit self-* mechanisms, it does adhere to the line of reasoning used in Autonomic Computing (Kephart and Chess, 2003), and defines a component model well suited for construction of self-management mechanisms, e.g., self-healing architectures and self-configuration components. Contributions of this paper include refinement of a software development model for flexible components and architectures, and presentation of a toolset designed to abstract service development complexity.

(Lau and Wang, 2007) provides a taxonomy for software component models that identifies a set of key characteristics, e.g., encapsulation and compositionality, for service-based component models. In (Lau et al., 2005), the authors also propose a component model based on exogenous connectors designed to facilitate loose coupling in aggregation control flow.

(Curbera et al., 2005) outlines a component-based programming model for service-oriented computing. This model focuses on goals similar to our approach and, e.g., identifies a need for flexibility and customization in SOA systems, and employs a view of Service-Oriented Architectures (SOAs) as distributed, abstractive architectures built on service-oriented components. While (Curbera et al., 2005) outlines requirements and structures for service-oriented component models and classifies component composition models, we propose a development methodology consisting of best practice recommendations for architecture design and component development.

Similar to the Spring Java application development framework (Spring Framework, 2011), iPOJO (Escoffier et al., 2007) provides a service-oriented component model where logic is implemented as POJOs and service handlers are injected through byte code modification. iPOJO emphasizes separation of service logic and interface implementations and provides a component model built on OSGi (OSGi, 2011) that provides both component- and application-level containers. The approach of this work aims to facilitate service-based application development, and presents a toolset to abstract service development complexity, while iPOJO provides a full and extensible software component model.

In addition to these component models, a number of service integration models exist (Peltz, 2003), and be categorized as, e.g., service composition, service orchestration, and service interaction models. In this work, we build architectures based on a model where we focus on component abstraction, and define component interactions in programming language terms rather than system-level orchestrations. Note that components developed using our development model are still compatible with service discovery and orchestration techniques, while we aggregate components in configuration and programming languages.

The Service-Oriented Modeling and Architecture (SOMA) (Arsanjani et al., 2010) approach outlines a development methodology for identification, design, development, and management of service-oriented software solutions. SOMA constitutes a complete service lifecycle development model that addresses modeling and management of business processes in addition to software development tasks. Compared to our approach, SOMA is a mature development model that provides guidelines for modeling and development tasks in large software projects. Our approach targets smaller development projects and aims to simplify service and component development by abstracting development complexity.

In addition to these efforts, a number of commercial service-based software development tools and environments, e.g., Microsoft .NET (Lowy, 2005), exist. In comparison to open source and scientific projects, commercial development tools are in general mature, well documented, and more continuously maintained. Commercial enterprises do however have business incentives for restricting development and integration flexibility in products, and commercial products are often associated with license cost models that discourage use in eScience application environments.

3 GRID AND CLOUD SOFTWARE DEVELOPMENT

Grid and Cloud eScience systems are distributed and designed for asynchronicity, parallelism, and decentralization. Grid environments organize users in Virtual Organizations (VOs) mapped onto virtual infrastructures built through resource site federations. As Grids build and provide abstract interfaces to existing resources through middlewares, Grid infrastructures focus heavily on integration of existing resources and systems, and have adapted a number of tools well suited for system integration. For these reasons, many Grid architectures are designed as SOAs and implemented using Web Services.

Cloud computing evolved from a number of distributed system efforts and inherits technology and methodology from Grid computing. To applications, Clouds provide the illusion of endless resource capacity through interface abstraction, and run virtual machines on resources employing hardware-enabled virtualization technologies. As the notion of a service (an always-on, network-accessible software) fits well in the Cloud computing model, many Cloud environments build on, or provide, service-oriented interfaces. In effect, Grids provide scalability through federation of resources, while Clouds provide computational elasticity through abstraction of resources.

In service-based software development, a number of trade-offs and development issues exist.

Software reuse. Development of Grid and Cloud computing infrastructure components and applications consists, at least in part, of integration of existing systems. Integration projects tend to produce software specific to integration environments, and limit software reuse to component level. To enable software reuse, components should be kept flexible and customizable (Elmroth and Östberg, 2008; Elmroth et al., 2008), and SOA programming models should emphasize construction of modules that developers can customize without source code modification (Curbera et al., 2005).

Software flexibility. In SOA environments, component interactions are specified in terms of service interfaces, orchestrations, and compositions. Services define interfaces in terms of service descriptions (SOAP style Web Services), or via exposure of resources through HTTP mechanisms (REST services). As SOAs are typically designed to abstract underlying execution environments and dynamically provision functionality, services may be deployed in dynamic and heterogeneous environments. To facilitate integration between components, SOAP Web Service platforms provide Application Programming Interfaces (APIs) and employ code generation tools to provide boilerplate solutions for component interaction. REST architectures define resource representations in documentation and often encapsulate component invocation in APIs. By providing mechanisms for dynamic recomposition of architectures and reconfiguration of components, SOAs facilitate system deployment and administration flexibility.

Multi-dimensional scalability. There are many types of scalability in Grid and Cloud Computing. Within performance, scalability can be categorized in dimensions such as horizontal or vertical scalability, i.e. ability to efficiently utilize many or large resources, or in time, e.g., in terms of computation throughput, makespan, and response time. In Clouds, hardware virtualization enabled resource elasticity describes system ability to vary number and size of hardware resources contributing to Cloud resource and infrastructure capacity.

While achieving performance scalability is central to computational systems (and the explicit focus of many Grid and Cloud Computing efforts), there are also other types of scalability likely to impact software sustainability in Grid and Cloud Computing environments. Scalability in, e.g., development, deployment, and administration, are becoming limiting factors as software projects scale up. In development, scalability is often limited by system complexity and problem topology. Deployment flexibility can be measured in terms of adaptability and integrability, and is typically limited by restrictions imposed by architecture design or implementation choices. Increasingly, as software projects grow in size and complexity, configuration and administration scalability is becoming a factor. Administration scalability can be measured in terms of automation, configuration complexity, and monitoring transparency. Computational and storage scalability are often limited by problem topology, while development and deployment scalability tend to be limited by solution complexity.

Development complexity. Limitations in current service engines, frameworks, and development tools often result in increased component implementation complexity and reduced developer productivity (when compared to non-service-based software development). Service development APIs expose low level functionality and service engine integration logic leaving, e.g., parts of message serialization tasks to service and client developers. For services defined with explicit service interface descriptions, e.g., Web Service Description Language (WSDL) documents, a number of code generation tools exist. These typically extract type systems and service interface information from service interface descriptions, and generate code to integrate and communicate with services.

Current service APIs and code generators tend to be service platform specific and tie service and service client implementations to specific service engines. Tool complexity often leads to complex interactions with and within service implementations, resulting in service interface implementations being mixed with service functionality logic. In addition, service description and type validation languages are often complex. WSDL and XML Schema are examples of widely used languages with great expressive power and steep learning curves that lower developer productivity and obstruct component reuse.

Complexity and ambiguity in service description formats lead to steep learning curves, high develop-

ment overhead, and incompatibilities in service interface and message validation implementations. As generated code is intended for machine interpretation, it is typically left undocumented, unindented, and hard to read, reducing toolset transparency. Tool complexity results in vendor lock-in, reduced development productivity, and decreased software stability.

4 DESIGN METHODOLOGY

To address service software reuse, flexibility, and scalability issues, we propose a SOA development methodology consisting of two parts: an architecture design methodology and a service component model. The methodology emphasizes modularity and customization on both architecture and component level. Architectures isolate functionality blocks in services and define architectures as loosely coupled networks of services that can be customized through recomposition mechanisms. Services separate component modules and offer customization through exposure of plug-in customization points. To support this methodology we provide a service development toolset (presented in Section 5) designed to abstract service development complexity. The overall goal of this methodology is to raise service development abstraction levels and produce systems that are flexible in development, deployment, and administration.

4.1 Design Perspective

To design modular and reusable software, we employ a model of software evolution based on the notion of an ecosystem of infrastructure and application components (Elmroth et al., 2008). Here systems form niches of functionality and components are selected for use based on current client or application needs. Over time, software are subject to evolution based on natural selection. In conjunction with this model, we observe the line of reasoning used in autonomic computing (Kephart and Chess, 2003), and address scalability through modularity and reduction of software complexity. The proposed methodology provides component and architecture models well suited for construction of software self-* mechanisms.

In architecture design, we combine the top-down perspective of structured system design with the modeling of objects and relationships between objects of object-oriented programming. Similar to the reasoning of (Lau and Wang, 2007), we design components that expose functionality through well-defined Web Service interfaces and compose architectures as SOAs. System composition takes place in the component design (through interface, dependency, and communication design) and deployment (through runtime configuration) phases, and is determined through a system de- and recomposition approach (Elmroth and Östberg, 2008). As encapsulation (modularity) counters software complexity (Lau and Wang, 2007), we utilize interface abstraction and late binding techniques to construct loosely coupled and location transparent components.

The design approach defines architectures as flexible, dynamically reconfigurable, and loosely coupled networks of services. Autonomous blocks of functionality are identified and isolated, and components are modeled to keep component interactions coarse-grained and infrequent. Functionality likely to be of interest to other components or clients is exposed as services or customization points.

For applications, this approach provides flexibility in utilization and customization of system functionality, and increases system task parallelization potential. Construction of software as SOAs emphasizes composition of new systems from existing components, allows a model of software reuse where applications dynamically select components based on current needs, and facilitates replacement and updates of individual components. On component level, this approach results in increased modularity and a greater focus on interface abstraction, benefiting component and system flexibility, adaptability, and longevity.

While architectures designed as networks of services may be distributed, components are likely to (at least partially) be co-hosted for reasons of performance and ease of administration. Co-hosted components are able to make use of local call optimizations, which greatly reduce service container memory and CPU load, as well as system network bandwidth requirements. Use of local call optimizations results in less network congestion issues, fewer network stack package drops, fewer container message queue drops, and reduced impact of component communication overhead and errors. Local call optimization mechanisms allows design of systems that combine the component communication efficiency of monolithic systems with the deployment flexibility of distributed service-based systems.

4.2 Architecture Design

Our design approach is summarized in three steps.

Identification of autonomous functionality blocks. Similar to how objects and object relationships are modeled in object-oriented programming, autonomous functionality blocks are identified and block interactions are modeled using coarse-grained

service communication patterns. Key to this approach is to strike a balance between architecture fragmentation and the need to keep components small, single-purpose, and intuitive. Component dependency patterns are identified to illuminate opportunities for parallelization of system tasks.

Identification of exposure mechanisms for functionality blocks. Functionality of interest to components in neighboring ecosystem niches, with clear levels of abstraction, and where well-defined interfaces can be defined is exposed as services. Functionality not of interest to other systems, but where component customization would increase system flexibility is exposed as customization points, e.g., through configuration and plug-ins. Service-exposed functionality is generally identified at architecture level, while customization points are typically identified at component level.

Design of service interactions and interfaces. Formulation of interfaces and service communication patterns are essential to performance efficiency in SOAs. Key to our approach is to design architectures that allow services to function efficiently as both local objects and distributed services. As exposure of components as services may lead to unexpected invocation patterns, defensive programming techniques and design patterns are employed to keep service interfaces unambiguous, simple, and lean.

4.3 Component Design

In component design, we adhere to the general principles for a service-oriented component model presented in (Cervantes and Hall, 2004) and organize service components in a way similar to classic three-tier architectures. To enable a software development model facilitating loose coupling of service components, we emphasize separation of modules in component design (Yang and Papazoglou, 2004). Separation of service client, interface, logic, and storage components facilitates flexibility in distributed system development and integration. Separation of service clients and interface implementations is a key mechanism in Service-Oriented Computing (SOC) that facilitates loose coupling in systems, and is here extended to provide (optional) flexibility in logic implementation.

Use of language and platform independent techniques for data marshaling and transportation allows service clients to be implemented using application-specific languages and tools, facilitating system integrability and adoptability. Development of service components using this pattern can be used to, e.g., create lightweight Web Service integration interfaces for existing components running in component environments such as the Common Object Model (COM) or Enterprise Java Beans (EJB).

Separation of service interface and logic implementation enables use of alternative wire protocols and communication paradigms, and facilitates implementation and deployment of service logic in component model environments. Encapsulation of platform-specific code, i.e. service client and interface implementations, facilitates migration and porting of service logic to alternative service platforms.

Implementation of local call optimizations allow logic components to function as local Java objects in service clients (Elmroth and Östberg, 2008), reducing component communication overhead to the levels of monolithic architectures (Östberg and Elmroth, 2011). Embedding local call optimizations in component APIs allows optimizations to be transparent to developers and ubiquitous in service implementations (see Section 5), combining the deployment flexibility of distributed SOAs with the communication performance of monolithic architectures.

Implementation of service logic in component models introduce additional requirements for software development. Best practices for Web Service and component-based software development overlap partially. Web Service components should, e.g., be implemented to be thread safe, communicate asynchronously, consume minimal system resources, and minimize service response times. Separation of service logic from storage layers enables loose coupling between component models and storage mechanisms, and facilitates migration of service logic between component environments.

To provide component-level flexibility, we define structures for customization points as plug-ins. Component interfaces are defined for advisory and functionality provider interfaces, and customization point implementations are dynamically loaded at runtime. Through this mechanism, third parties can replace, update, and provide plug-in components for internal structures inside services without impacting design of application architectures.

5 DEVELOPMENT TOOLSET

Software reuse in highly specialized, complex, and low maturity environments such as emerging Grid and Cloud computing eScience platforms is limited and inefficient. Code generation tools target automation of software development and can facilitate software reuse by providing boiler-plate solutions for service communication and isolate service logic. To address software reuse and abstraction of develop-

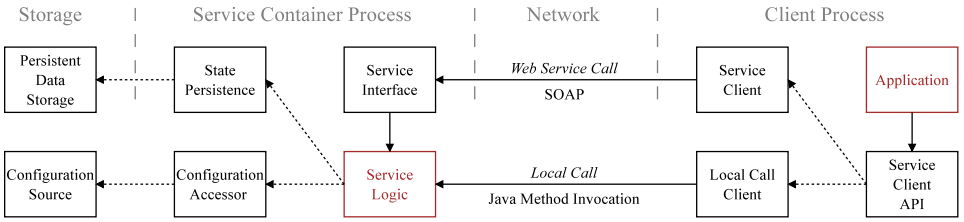


Figure 1: SDAT service structure. Manually developed components (application and service logic) decoupled from generated components (service invocation, configuration management, and state persistence). Transparent local call optimizations reduce service invocation overhead and container load.

ment complexity, we present a service development toolset called the Service Development Abstraction Toolset (SDAT). SDAT builds on the component design model of Section 4.3 and is designed to raise service development abstraction levels through a simplified service description language and a code generation tool that separates service components and provides boiler-plate solutions for, e.g., data representation, validation, and communication.

5.1 Simplified Service Description

To promote loose coupling of service clients and implementations, enable service code generation, and facilitate service discovery, we define a simplistic XML-based service interface description format called the Abstractive Service Description Language (ASDL). The format specifies service interfaces in terms of tree-based data types and call by value operations on defined types. Data types are defined in a schema language defined as a subset of XML Schema. Conceptually, ASDL can be seen as a minimalistic subset of the Web Service Description Language (WSDL), where the expressive power of XML Schema and WSDL are reduced in favor of simplicity in interface interpretation and data representation.

The ASDL type schema language consists of a restricted set of XML and XML Schema tags. Data field types are defined using `simpleType` tags, and organized in hierarchical records using `complexType` tags. For message validation schema completeness `element` tags are inserted, and service interfaces are defined using `service` and `operation` tags. All schemas define a target namespace, use explicitly referenced namespaces, and all tags are qualified. XML Schema mechanisms for `include` and `import` tags are supported to facilitate type definition reuse. ASDL is designed to provide a service interface design model semantically equivalent to Java interfaces using immutable Java classes.

Through ASDL, SDAT exploits XML Schema document validation without encumbering interface designers with the full complexity of XML Schema and WSDL. For translation to WSDL interface descriptions and generation of SOAP Web Service implementations, the following assumptions are made. All service communication is kept document-oriented and use literal encoding of messages. Interfaces are designed to have single-part messages and define a single service per service description. Services define a single type set per service description and all data fields are encoded in XML elements. Exceptions are exposed as SOAP faults and serialized as messages defined in the service type schema.

5.2 Code Generation

To facilitate design of architectures as loosely coupled networks of services, SDAT defines a service structure (illustrated in Figure 1) that isolates service functionality blocks and employs a local call optimization mechanism for co-hosted services. Local call optimizations transparently bypass network serialization and reduce invocation overhead and container load (Elmroth and Östberg, 2008). Immutable data types are exposed in service interfaces and used for service invocation. Optional message validation is performed in service client and interface components. The service structure defines modules for:

- Data type representations. Flat record structures are extracted from service description schemas and data type representation components are defined as immutable and serializable Java classes.
- Service interfaces. A service invocation framework abstracting call optimizations is built into service implementations and client APIs, making optimizations transparent to service clients.
- Message data validation. XML Schemas are extracted from service interface descriptions and used to generate message validation components.

- Service configuration management. A configuration accessor and monitor API is defined for service components. Configuration modules are available to service client and logic components.
- Persistent state storage. A framework for persistent service state storage is defined and accessible to service logic components. Persistence modules are customizable and can be extended to support, e.g., additional databases or serialization formats.

Separation of service interface and logic implementation allows compartmentalization of service platform specific code and facilitates abstraction of service interface and invocation implementation.

As illustrated in Figure 1, the SDAT service structure isolates manually coded components (application and service logic) from generated service components. Typical service development using SDAT consists of specification of a ASDL interface, generation of service components and interfaces, and implementation of a service logic interface. The goal of the tool is to abstract service development complexity to the level of implementation of a Java interface while providing optional customization of service components.

In addition to service components, SDAT generates deployment information (WSDL service descriptions, deployment descriptors, etc.), security code (WS-Security implementations, policy files, etc.), and a (Apache Ant) build environment. By default, SDAT generates compiled and deployable service packages where developers only need to add service logic implementations to services. To facilitate transparency, all source code generated is designed to be well formatted, easy to read, and documented.

Configuration of SDAT services is segmented into separate container and service configuration. As SDAT services expose customization points in the form of plug-ins for, e.g., service logic implementation, some configuration of the generated SDAT framework must be done on container level. A typical example of this is service plug-ins, which are specified in container Java runtime property settings. Configuration of service logic components is typically done through service configuration files accessed through the SDAT configuration API.

To enforce user-level isolation of service capabilities, service interface implementations instantiate unique service logic components for each invoking user. User-level isolation is implemented through a singleton factory that caches service instances based on invocation credentials, i.e. user certificates. This mechanism is designed to coexist with native mechanisms for component-level isolation of services.

For platform independence, clear representations of service interfaces, strong Web Service support, and in-memory compilation, SDAT is built and produces services in Java. For separation of service interface specification and service development, SDAT primarily supports SOAP style Web Services. Currently, SDAT integrates with the Apache Axis2 SOAP engine, but as service interface components are decoupled from service logic components, support for additional service engines, client languages, or communication patterns can be extended without affecting other service mechanisms. In extension, this model is expected to be of interest for creating components that can be hosted in different service engines, or even (simultaneously) support multiple types of service communication (e.g., REST, TCP/IP, and SOAP).

The aim of SDAT is to provide a development tool that abstracts complex and error-prone service communication development and allows service developers to focus on service logic. The tool is designed to provide a simplistic service model where service interfaces are kept simple, but still have expressive power enough to create efficiently communicating services. SDAT is designed as a prototype development tool that aims to integrate with existing service containers and development environments. While current code generators are tied to particular service environments or languages, SDAT is designed to support a development methodology rather than a specific platform or toolset.

6 CASE STUDIES

To illustrate our design approach, we present case studies from two recent software development projects. In these projects, which target development of Grid infrastructure components, emphasis is placed on development of flexible architectures capable of seamless integration into existing Grid and High-Performance Computing (HPC) environments.

6.1 GJMF

The Grid Job Management Framework (GJMF) (Östberg and Elmroth, 2010) illustrated in Figure 2 is a Grid infrastructure component built as a loosely coupled network of services. Designed to provide middleware-agnostic and abstractive job management interfaces, the GJMF offers concurrent access to multiple Grid middlewares through components organized in hierarchical layers. Services in higher layers aggregate functionality of lower layers, and form a rich middleware-agnostic Grid job management interface. Through a set of integration plug-ins, the framework can be customized to, e.g.,

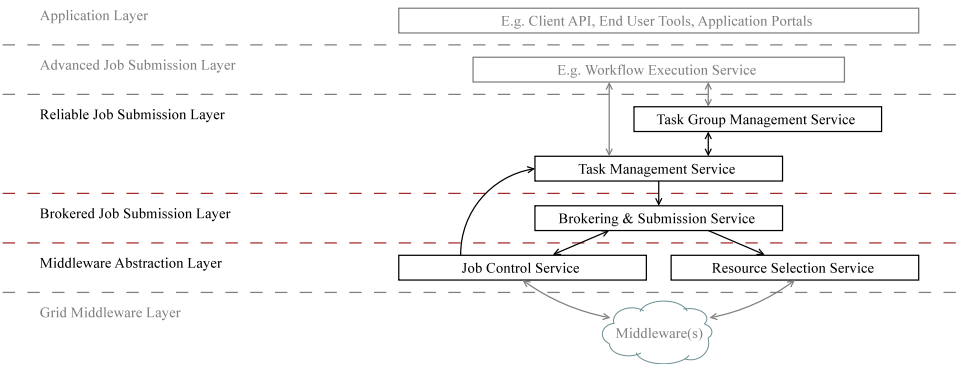


Figure 2: The Grid Job Management Framework (GJMF). A hierarchical framework of services offering abstractive Grid job management. Illustration from (Östberg and Elmroth, 2010).

support additional Grid middlewares, replace job brokering algorithms, and define job failure management policies. Framework composition, plug-in selection, and component configuration are configurable through dynamic configuration modules.

The GJMF predates the service development toolset of Section 5 and has served as a testbed for the development methodology. The framework is developed using the Globus Toolkit v4 (Foster, 2005) and employs WSRF notifications for service monitoring and coordination. Due to the flexibility of the framework architecture, the GJMF is deployable in multiple concurrent settings as, e.g., a gateway job management interface, an alternative job brokering mechanism, or a personal client-side job monitoring tool.

6.2 FSGrid

FSGrid (Östberg et al., 2011) is a job prioritization mechanisms for scheduler-based Grid fairshare policy enforcement. Designed as a distributed architecture consisting of a network of services, FSGrid is segmentable and can be tailored to resource site deployments. The system virtualizes usage metrics and resource site capacity, and is capable of collaboration between different types of FSGrid configurations.

As illustrated in Figure 3, FSGrid deploys components to be close to data and computation, and enforces VO and resource site allocation policies simultaneously. The system mounts VO allocation policies onto resource site policies, assembles and operates on global usage data, and injects a fairshare job prioritization mechanism into local scheduling decisions.

The flexibility of the architecture allows FSGrid to be deployed in different configurations on different sites, to be dynamically reconfigured, and adapt to dynamic changes in usage data and allocation policies. FSGrid exposes customization points through dynamic service configuration modules, and plug-ins for usage decay functions and fairshare metrics.

In FSGrid, all service interfaces are describe using ASDL and all service components developed using SDAT. The code generation tools of SDAT help to isolate service implementations from service service container and communication dependencies. The system is deployed using Apache Axis2 (Apache Web Services Project - Axis2, 2011) service containers and integrates into cluster schedulers using custom service clients. As GJMF and FSGrid are designed using the same methodology but different tools and platforms, they make a suitable platform for evaluation.

6.3 Evaluation

The GJMF and FSGrid are developed using the same approach to SOA design, and designed with the same goal: to provide flexible architectures for Grid infrastructure. Both systems are designed as loosely coupled and reconfigurable networks of services, expose customization points for tailoring of component functionality, and provide APIs to facilitate integration into deployment environments.

Experiences from integration of GJMF with the LUNARC application portal (Lindemann and Sandberg, 2005) and a problem-solving environment in R are documented in (Elmroth et al., 2011) and (Jayawardena et al., 2010) respectively. These projects illustrate benefits of construction of infrastructure components as flexible networks of service SOAs. The GJMF exposes a range of job submission, monitoring, and control interfaces that can be utilized to integrate in heterogeneous deployment en-

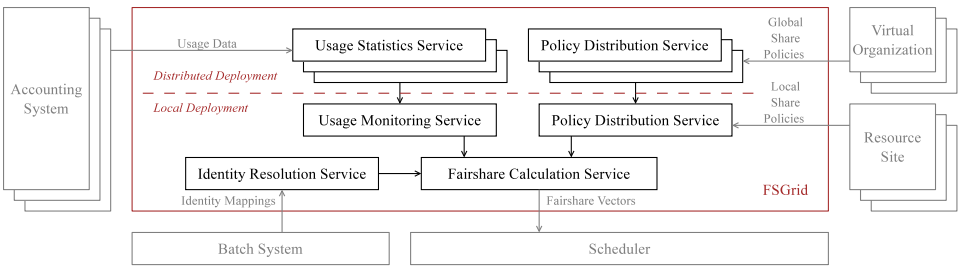


Figure 3: FSGrid, a scheduler-based fairshare job prioritization framework for Grid environments built as a distributable set of services. Illustration from (Östberg et al., 2011).

vironments. The deployment flexibility of the framework allows parts of the framework to fulfill different job management roles and be hosted separately.

A recent performance analysis (Östberg and Elmroth, 2011) illustrates that organization of services in hierarchical layers allows the GJMF to mask service communication overhead through parallel task processing. Local call optimizations significantly reduce communication overhead and container load for inter-service communication. The service structure abstracts use of local call optimizations and allows optimizations to be ubiquitous and transparent.

Compared to GJMF development, the FSGrid project benefits from use of SDAT in reduction of service interface implementation complexity. FSGrid development cycles are shorter, and use of ASDL and SDAT facilitates experimentation in architecture design. FSGrid benefits from use of SDAT in compartmentalization and reduction of complexity in service logic implementation. The proposed development methodology provides GJMF and FSGrid architecture and component level reconfigurability, adaptivity, and flexibility. The SDAT abstracts service development and facilitates porting of core system functionality to alternative service platforms.

Impact of the proposed methodology on the internal quality of produced software can be evaluated through, e.g., evaluation of the maintainability and cohesion of service interfaces (Perepletchikov et al., 2010). Study of the impact of SDAT on the quality of GJMF and FSGrid is subject for future work.

7 CONCLUSION

In this paper we address software development practices for eScience applications and infrastructure. We discuss service-based software development in Grid and Cloud computing environments and identify a set of current software development issues, e.g., complexity and lack of flexibility in service development tools. To address these issues, we propose a SOA-based software design methodology constituted by a set of architecture design guidelines, a component design model, and a toolset designed to abstract service development complexity. The approach is illustrated and evaluated in a case study of experiences from two recent software development projects.

Our design approach aims to produce software architectures that are flexible in deployment, reduce need for complex distributed state synchronization, and facilitate distribution and parallelization of system tasks. The component model isolates service components in standalone modules, exposes functionality as services and customizable plug-ins, and compartmentalizes platform-specific service interface and invocation code. The toolset is designed to support the design methodology through abstraction of development complexity and facilitation of flexibility in service development, deployment, and utilization. A simplified service description language abstracts service interface and type description complexity.

Experiences from recent software development projects illustrate the need for structured development models for Service-Oriented Architectures. The hierarchical structure of the GJMF allows the framework to dynamically function as several types of job management interfaces simultaneously as well as mask service invocation overhead. Building the system as a network of services allows FSGrid to deploy components close to data and computations as well as provide more flexible interfaces for scheduler integration. The design approach provides both systems with increased flexibility in development and deployment. Use of the SDAT toolset abstracts service development complexity and provides FSGrid a component model that supports dynamic reconfiguration and encapsulation of platform-specific service functionality.

ACKNOWLEDGMENTS

The authors extend their gratitude to the anonymous reviewers, Deb Agarwal, and Vladimir Vlassov for valuable feedback that has contributed to the quality of this paper. This work is funded in part by the Swedish Research Council (VR) under Contract 621-2005-3667, the Swedish Government's strategic research project eSSENCE, and the European Community's Seventh Framework Programme (FP7/2001-2013) under grant agreement 257115 (OPTIMIS). The authors acknowledge the Lawrence Berkeley National Laboratory (LBNL) for supporting the project under U.S. Department of Energy Contract DE-AC02-05CH11231.

REFERENCES

Apache Web Services Project - Axis2 (2011). http://ws.apache.org/axis2, February 2011.

Arsanjani, A., Ghosh, S., Allam, A., Abdollah, T., Ganapathy, S., and Holley, K. (2010). SOMA: A method for developing service-oriented solutions. *IBM Systems Journal*, 47(3):377–396.

Cervantes, H. and Hall, R. (2004). Autonomous adaptation to dynamic availability using a service-oriented component model. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 614 – 623.

Curbera, F., Ferguson, D., Nally, M., and Stockton, M. (2005). Toward a programming model for service-oriented computing. In Benatallah, B., Casati, F., and Traverso, P., editors, *Service-Oriented Computing - ICSOC 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin / Heidelberg.

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

Elmroth, E., Holmgren, S., Lindemann, J., Toor, S., and Östberg, P.-O. (to appear, 2011). 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*.

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

Escoffier, C., Hall, R. S., and Lalanda, P. (2007). iPOJO: an Extensible Service-Oriented Component Framework. *Services Computing, IEEE International Conference on*, 0:474–481.

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

Jayawardena, M., Nettelblad, C., Toor, S., Östberg, P.-O., Elmroth, E., and Holmgren, S. (2010). A Grid-Enabled Problem Solving Environment for QTL Analysis in R. In *In Proceedings of the 2nd International Conference on Bioinformatics and Computational Biology (BICoB)*, pages 202–209. ISCA.

Kephart, J. O. and Chess, D. M. (2003). The Vision of Autonomic Computing. *Computer*, 36:41–50.

Lau, K.-K., Velasco Elizondo, P., and Wang, Z. (2005). Exogenous connectors for software components. In Heineman, G. T., Crnkovic, I., Schmidt, H. W., Stafford, J. A., Szyperski, C., and Wallnau, K., editors, *Component-Based Software Engineering*, volume 3489 of *Lecture Notes in Computer Science*, pages 90–106. Springer Berlin / Heidelberg.

Lau, K.-K. and Wang, Z. (2007). Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709 –724.

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

Lowy, J. (2005). *Programming .NET Components, 2nd Edition*. O'Reilly Media, Inc.

OSGi (2011). http://www.osgi.org, February 2011.

Östberg, P.-O. and Elmroth, E. (submitted, 2010). GJMF - A Composable Service-Oriented Grid Job Management Framework. Preprint available at http://www.cs.umu.se/ds.

Östberg, P.-O. and Elmroth, E. (submitted, 2011). Impact of Service Overhead on Service-Oriented Grid Architectures. Preprint available at http://www.cs.umu.se/ds.

Östberg, P.-O., Henriksson, D., and Elmroth, E. (submitted, 2011). Decentralized, Scalable, Grid Fairshare Scheduling (FSGrid). Preprint available at http://www.cs.umu.se/ds.

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

Perepletchikov, M., Ryan, C., and Tari, Z. (2010). The impact of service cohesion on the analyzability of service-oriented software. *IEEE Transactions on Services Computing*, 3:89–103.

Spring Framework (2011). http://www.springsource.org, February 2011.

Yang, J. and Papazoglou, M. P. (2004). Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97 – 125. The 14th International Conference on Advanced Information Systems Engineering (CAiSE*02).